ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Θέματα υλοποίησης μεταφραστή για το πρότυπο παράλληλου προγραμματισμού OpenMP

Άλκης Γεωργόπουλος

Επιβλέπων Καθηγητής: Βασίλειος Δημακόπουλος

Ιωάννινα, Σεπτέμβριος 2004

Πίνακας περιεχομένων

Περίληψη

Η εργασία αυτή έχει σαν θέμα την υλοποίηση ενός μεταφραστή της γλώσσας C, ο οποίος υποστηρίζει τις προδιαγραφές OpenMP για παράλληλο προγραμματισμό κοινής μνήμης. Σύμφωνα τις προδιαγραφές αυτές, ένας προγραμματιστής μπορεί να αναπτύξει ένα σειριακό πρόγραμμα και στη συνέχεια να το παραλληλοποιήσει προσθέτοντας στον πηγαίο του κώδικα ειδικές οδηγίες προς τον μεταγλωττιστή. Ο μεταγλωττιστής λαμβάνοντας υπ' όψη του αυτές τις οδηγίες παράγει το αντίστοιχο παράλληλο πρόγραμμα, το οποίο είναι σε μορφή πολυνηματικού πηγαίου κώδικα C.

Στα πλαίσια της εργασίας δημιουργήσαμε μία ενιαία διεπαφή για πολυνηματικό κώδικα, η οποία είναι αρκετά γενική ώστε να υποστηρίζει διαφορετικές υλοποιήσεις νημάτων. Έχουν υποστηριχθεί με επιτυχία τα νήματα POSIX και τα νήματα Solaris. Επίσης, δημιουργήσαμε μία βιβλιοθήκη η οποία επιτρέπει την παρακολούθηση των επιδόσεων του παραγόμενου κώδικα (profiling), υλοποιώντας το πρωτόκολλο POMP και κάνοντας χρήση των εργαλείων ανάλυσης EXPERT και γραφικής αναπαράστασης CUBE. Τέλος, η υλοποίηση περιέλαβε σημαντικές βελτιστοποιήσεις που αφορούν στον παραγόμενο κώδικα, στην runtime υποστήριξη και στην εκμετάλλευση της υποκείμενης αρχιτεκτονικής.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή μου, κ. Βασίλειο Δημακόπουλο για την αρωγή του σ' αυτήν την προσπάθεια, την γυναίκα μου και τα παιδιά μου για την κατανόηση και την υποστήριξή τους, και την Δ/νση Β/θμιας Εκπαίδευσης Ν. Ιωαννίνων καθώς και το Ίδρυμα Κρατικών Υποτροφιών που μου έδωσαν την δυνατότητα να ασχοληθώ απερίσπαστα με αυτή την εργασία.

Κεφάλαιο 1

Εισαγωγή

1.1 Παράλληλοι υπολογιστές

Ένας παράλληλος υπολογιστής μπορεί να οριστεί ως ένα σύνολο επεξεργαστών οι οποίοι είναι δυνατόν να συνεργαστούν για την επίλυση ενός υπολογιστικού προβλήματος [Fost95]. Αυτός ο ορισμός είναι αρκετά γενικός ώστε να περιλαμβάνει παράλληλους υπερυπολογιστές με εκατοντάδες χιλιάδες επεξεργαστές, δίκτυα σταθμών εργασίας, σταθμούς εργασίας με πολλούς επεξεργαστές, και ενσωματωμένα συστήματα.

Με την διαρκή αύξηση στις επιδόσεις των υπολογιστικών συστημάτων, μπορεί κάποιος να υποθέσει ότι κάποια στιγμή θα γίνουν «αρκετά γρήγορα» και η ανάγκη για συνεχώς μεγαλύτερη επεξεργαστική ισχύ θα καταλαγιάσει. Όμως η ιστορία δείχνει ότι καθώς η τεχνολογία βελτιώνεται και καλύπτει τις ανάγκες των υπαρχουσών εφαρμογών, καινούργιες εφαρμογές εμφανίζονται και απαιτούν την ανάπτυξη νεότερων τεχνολογιών. Χαρακτηριστική είναι η φράση που αποδίδεται στον Bill Gates «640Kb μνήμης (RAM) θα πρέπει να είναι αρκετά για οποιονδήποτε», τον καιρό που έβγαιναν τα πρώτα XT στην αγορά. Οι απαιτήσεις των προσωπικών υπολογιστών σε μνήμη και επεξεργαστική ισχύ έχουν χιλιαπλασιαστεί μέσα στις δύο τελευταίες δεκαετίες.

Η αύξηση στην ταχύτητα των επεξεργαστών όμως δεν είναι δυνατόν να συνεχιστεί επ' αόριστο. Φυσικοί νόμοι (ταχύτητα ηλεκτρονίων) αλλά και περιορισμοί ολοκλήρωσης (μέγιστος αριθμός transistor για συγκεκριμένη επιφάνεια) διαγράφουν

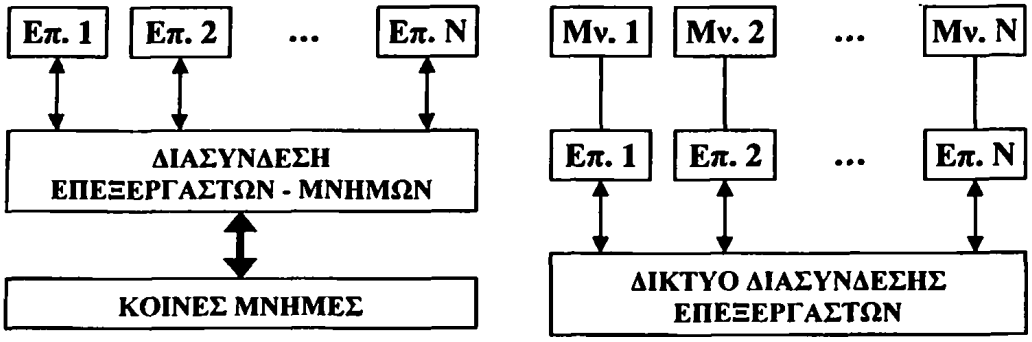
ένα πάνω όριο στην αύξηση της ταχύτητας. Τα παράλληλα συστήματα φαίνεται ότι μπορούν να δώσουν μια λύση σε αυτό το πρόβλημα. Η ιδέα είναι απλή: αντί να προσπαθούμε να φτιάξουμε πιο γρήγορους επεξεργαστές μοιράζουμε την δουλειά σε περισσότερους από έναν. Βέβαια στην πράξη λίγα είναι τα προβλήματα που παραλληλοποιούνται πλήρως, και συνήθως υπάρχει και επιπλέον φόρτος για τον συντονισμό των εργασιών.

Υπάρχουν κάποια θεμελιώδη επιστημονικά προβλήματα τα οποία έχουν χαρακτηριστεί ως «μεγάλες προκλήσεις» [HPCC94]. Η λύση τους απαιτεί τεράστια υπολογιστική ισχύ, την οποία μπορούν να παρέχουν μόνο μαζικά παράλληλοι υπολογιστές, και αναμένεται να επιφέρει μεγάλες οικονομικές και επιστημονικές αλλαγές. Περιλαμβάνουν προβλήματα από τις παρακάτω ενδεικτικές κατηγορίες:

- Επιστήμη υπολογιστών: ανάπτυξη υλικού και λογισμικού με απώτερο σκοπό οι υπολογιστές να μπορούν «να δουν, να κινηθούν και να μιλήσουν»
- Παρακολούθηση και πρόβλεψη περιβαλλοντολογικών αλλαγών: υλοποίηση μοντέλου για την αναπαράσταση του κλίματος σε παγκόσμια κλίμακα, σε μαζικά παράλληλους υπολογιστές με ταχύτητες της τάξης των teraFLOPs, σεισμικά μοντέλα
- Ενεργειακά: ανάπτυξη μαζικά παράλληλων μοντέλων για την μελέτη πυρηνικών αντιδραστήρων
- Αεροδιαστημική: επίλυση προβλημάτων των πεδίων της δυναμικής ρευστών και μελέτη των στροβιλισμών που απαντώνται στην δυναμική ρευστών της γεωφυσικής και της αστροφυσικής
- Μοριακή βιολογία και επεξεργασία βιοϊατρικής εικόνας: αναπαράσταση των συνδέσμων του ανθρωπίνου μυοσκελετικού συστήματος με προηγμένα υπολογιστικά μοντέλα
- Βελτιστοποίηση της διαδικασίας παραγωγής προϊόντων: ανάπτυξη λογισμικού για την προσομοίωση σε επίπεδων ατόμων του σχεδιασμού πρώτων υλών και εφαρμογή σε κρίσιμα βιομηχανικά προβλήματα
- Διαστημική: μοντέλο για την προέλευση και τον τρόπο σχηματισμού των γαλαξιών

1.1.1 Πολυεπεξεργαστές κοινής / κατανεμημένης μνήμης

Τα παράλληλα συστήματα είναι γνωστά και ως πολυεπεξεργαστές και χωρίζονται ανάλογα με την αρχιτεκτονική τους σε 2 βασικές κατηγορίες:



(α) (β)

Σχήμα 1-1 Οργανώσεις πολυεπεξεργαστών

Στους πολυεπεξεργαστές κοινής μνήμης (shared memory multiprocessors) *δεν υπάρχει σύνδεση μεταξύ των επεξεργαστών, παρά μόνο ανάμεσα στους επεξεργαστές και τη μνήμη* η οποία είναι άμεσα προσπελάσιμη από όλους (Σχήμα 1-1α). Κάθε επεξεργαστής μπορεί να εκτελεί διαφορετική εντολή επάνω σε διαφορετικά δεδομένα τα οποία όμως μπορούν να τα προσπελάσουν όλοι. Αυτό έχει σαν αποτέλεσμα ότι όλοι οι επεξεργαστές «βλέπουν» την ίδια μνήμη και με τον ίδιο τρόπο. Η επικοινωνία μεταξύ των επεξεργαστών γίνεται μέσω της τροποποίησης κοινών μεταβλητών στην μνήμη.

Αντίθετα με τους πολυεπεξεργαστές κοινής μνήμης, στα συστήματα κατανεμημένης μνήμης (distributed memory multiprocessors - Σχήμα 1-1β) κάθε επεξεργαστής έχει την ιδιωτική του μνήμη την οποία μπορεί να προσπελάσει απευθείας μόνο αυτός. Η επικοινωνία των επεξεργαστών είναι εφικτή *λόγω ύπαρξης διασυνδετικού δικτύου διά μέσω του οποίου γίνεται ανταλλαγή μηνυμάτων*. Έτσι, οτιδήποτε χρειαστεί ο επεξεργαστής i από τον επεξεργαστή j (π.χ. να προσπελάσει κάποιο δεδομένο από την μνήμη του j) θα το ζητήσει μέσω μηνύματος από τον j, ο οποίος πάλι μέσω μηνύματος που θα ταξιδέψει στο δίκτυο θα το στείλει στον επεξεργαστή i.

Πρόσφατα, μία τρίτη κατηγορία αποτελούν οι λεγόμενοι *ομαδοποιημένοι πολυεπεξεργαστές* (clustered multiprocessors) οι οποίοι αποτελούν «διασταύρωση» των πολυεπεξεργαστών κοινής και κατανεμημένης μνήμης. Πιο συγκεκριμένα, τα συστήματα είναι οργανωμένα όπως στο Σχήμα 1-1β με τη διαφορά ότι κάθε επεξεργαστής έχει αντικατασταθεί από ομάδα επεξεργαστών οι οποίοι μοιράζονται μία κοινή μνήμη. Το δίκτυο διασύνδεσης πλέον είναι δίκτυο διασύνδεσης *ομάδων*. Η οργάνωση αυτή φαίνεται να γίνεται η πιο δημοφιλής.

1.2 Προγραμματισμός παράλληλων υπολογιστών

Ενώ στον σειριακό προγραμματισμό συνήθως δεν ενδιαφερόμαστε για την αρχιτεκτονική του συστήματος εκτέλεσης, στον παράλληλο προγραμματισμό τα πράγματα είναι πολύ διαφορετικά. Ο προγραμματιστής πρέπει να γνωρίζει την αρχιτεκτονική (κοινής/κατανεμημένης μνήμης κτλ) για να μπορεί να εκμεταλλευτεί καλύτερα τις δυνατότητές της. Αν και έχουν αναπτυχθεί διάφορα μοντέλα παράλληλου προγραμματισμού, κυρίαρχα είναι τα δύο που περιγράφονται στη συνέχεια.

Στο *μοντέλο κοινού χώρου διευθύνσεων* ή απλά *κοινής μνήμης* ο κάθε επεξεργαστής ακολουθεί την δική του ροή εκτέλεσης, εισάγοντας έτσι την έννοια της *διεργασίας*. Η επικοινωνία μεταξύ των διεργασιών γίνεται μέσω *κοινών* ή *διαμοιραζόμενων μεταβλητών*, ενώ εμφανίζονται προβλήματα και φυσικά αλγόριθμοι για την *δρομολόγηση*, τον *συγχρονισμό* και τον *αμοιβαίο αποκλεισμό* των εργασιών. Γνωστοί τρόποι προγραμματισμού για συστήματα κοινής μνήμης είναι η γλώσσα Sequent-C, μια επέκταση της C που χρησιμοποιήθηκε για τον προγραμματισμό των MIMD υπολογιστών κοινής μνήμης της Sequent, οι βιβλιοθήκες δημιουργίας και χειρισμού νημάτων όπως POSIX (pthreads) και Solaris threads και το System V Shared Memory IPC του Unix.

Το *μοντέλο μεταβίβασης μηνυμάτων* θεωρείται το δυσκολότερο μοντέλο παράλληλου προγραμματισμού, αφού ο προγραμματιστής θα πρέπει να καθορίσει επακριβώς τα σημεία στα οποία σταματούν προσωρινά οι υπολογισμοί και αρχίζουν οι επικοινωνίες. Το μοντέλο αυτό είναι αρκετά δημοφιλές αφού επιτρέπει «φθηνό» παραλληλισμό: ένα τοπικό δίκτυο υπολογιστών μπορεί να συνθέσει έναν «πολυϋπολογιστή» με μεγάλη επεξεργαστική ισχύ αν και με μικρότερες ταχύτητες επικοινωνίας μεταξύ των επεξεργαστών. Ένα τέτοιο σύστημα μπορεί να προγραμματιστεί κάνοντας χρήση της βιβλιοθήκης των sockets του Unix, αν και συνήθως για παράλληλο προγραμματισμό σε τοπικό δίκτυο προτιμώνται βιβλιοθήκες όπως το PVM [Sund90] και το MPI [MPI94]. Σε αυτό το μοντέλο βέβαια δεν είναι δυνατή η ύπαρξη κοινών μεταβλητών ανάμεσα στις διεργασίες. Έτσι όμως αποφεύγεται και η ανάγκη για αμοιβαίο αποκλεισμό.

Λόγω της δημοτικότητας του μοντέλου κοινού χώρου διευθύνσεων, γίνονται προσπάθειες να υποστηρίζεται παντού, ακόμα και πάνω από φυσικά κατανεμημένη μνήμη. Έτσι μιλάμε για συστήματα κατανεμημένης κοινής μνήμης (distributed shared memory) όπου η εικονική κοινή μνήμη παρέχεται είτε απευθείας από το υλικό είτε από ένα κατάλληλο επίπεδο λογισμικού.

1.3 Τι είναι το OpenMP

Το OpenMP είναι οι προδιαγραφές που έθεσε το OpenMP Architecture Review Board (ARB) για ένα σύνολο οδηγιών (directives) προς τους μεταγλωττιστές, συναρτήσεις βιβλιοθηκών και μεταβλητές περιβάλλοντος που μπορούν να χρησιμοποιηθούν για τον καθορισμό παραλληλισμού κοινής μνήμης σε προγράμματα Fortran και C/C++ [OARB98], [OARB02]. Σκοπός του είναι να παρέχει ένα εύχρηστο, απλό και φιλικό περιβάλλον παράλληλου προγραμματισμού κοινής μνήμης.

Το ARB πρότεινε αυτές τις προδιαγραφές επειδή οι οδηγίες παράλληλου προγραμματισμού κοινής μνήμης δεν καθορίστηκαν ποτέ σαν πρότυπο στην βιομηχανία. Μία προηγούμενη προσπάθεια προτυποποίησης, το ANSI X3H5 δεν υιοθετήθηκε ποτέ επίσημα. Οι διάφορες εταιρίες παρείχαν η κάθε μία το δικό της σύνολο οδηγιών, παρόμοιων σε σύνταξη και σημασιολογία, και όλες αυτές ακολουθούσαν έναν ξεχωριστό τρόπο σήμανσης των οδηγιών, είτε με κάποιο ειδικό σχόλιο (στην Fortran) είτε με οδηγίες `#pragma` (στις C/C++). Το OpenMP συγχωνεύει αυτά τα σύνολα οδηγιών σε ενιαία σύνταξη και σημασιολογία, και καταφέρνει τελικά να κάνει μεταφέρσιμο τον παράλληλο κώδικα κοινής μνήμης.

Το OpenMP, αν και είναι ήδη ευρέως διαδεδομένο, δεν έχει γίνει ακόμα αποδεκτό σαν επίσημο πρότυπο. Ουσιαστικά αποτελεί μία «συμφωνία» μεταξύ πολλών βιομηχανικών εταιριών και των χρηστών. Στο OpenMP Architecture Review Board συμμετέχουν μεγάλες εταιρίες υπολογιστών, όπως οι Compaq, Hewlett-Packard, Intel, IBM, Silicon Graphics και η Sun Microsystems. Επίσης, αρκετές είναι οι εταιρίες ανάπτυξης λογισμικού που υποστηρίζουν το OpenMP, όπως οι Absoft Corporation, Edinburgh Portable Compilers, ADINA R&D, ANSYS και άλλες.

1.4 Αντικείμενο της εργασίας

Η εργασία αυτή έχει σαν αντικείμενο την υλοποίηση του OMPi, ενός μεταφραστή της γλώσσας C ο οποίος υποστηρίζει τις προδιαγραφές OpenMP για παράλληλο προγραμματισμό κοινής μνήμης, και αποτελεί συνέχεια προηγούμενης πτυχιακής εργασίας [LeTz02]. Σύμφωνα τις προδιαγραφές αυτές, ένας προγραμματιστής μπορεί να αναπτύξει ένα σειριακό πρόγραμμα και στη συνέχεια να το παραλληλοποιήσει προσθέτοντας στον πηγαίο του κώδικα ειδικές οδηγίες προς τον μεταγλωττιστή. Ο μεταγλωττιστής λαμβάνοντας υπ' όψη του αυτές τις οδηγίες παράγει το αντίστοιχο παράλληλο πρόγραμμα, το οποίο είναι σε μορφή πολυνηματικού πηγαίου κώδικα C.

Ένα από τα μέρη της εργασίας αφορούσε στην απομόνωση της προγραμματιστικής *διεπαφής* (Application Programming Interface, API) *χειρισμού νημάτων* που χρησιμοποιούσε ο μεταφραστής. Έγινε μία μελέτη διαφόρων πακέτων πολύνηματικού προγραμματισμού ώστε να καθοριστούν με σαφήνεια οι απαιτήσεις που θα έπρεπε να έχει το OMPi από τα πακέτα αυτά. Στη συνέχεια το σύνολο των τύπων και των συναρτήσεων που προέκυψαν δομήθηκε σαν ξεχωριστή βιβλιοθήκη (`othread.h`) και έγιναν οι αντίστοιχες υλοποιήσεις για POSIX και Solaris threads, ενώ το API είναι αρκετά γενικό ώστε να μπορεί να υλοποιηθεί στο μέλλον και σε διαφορετικές αρχιτεκτονικές.

Ένα δεύτερο κομμάτι της εργασίας ήταν η αυτόματη ενσωμάτωση *κώδικα παρακολούθησης της επίδοσης* (profiling) στα εκτελέσιμα αρχεία που παρήγαγε το OMPi. Ο παράλληλος προγραμματισμός εμπεριέχει αρκετά προβλήματα συγχρονισμού και βέλτιστης κατανομής των εργασιών ανάμεσα στους επεξεργαστές. Είναι λοιπόν απαραίτητη η παροχή εποπτικών εργαλείων που θα κάνουν δυνατή την ανεύρεση των «προβληματικών» σημείων του κώδικα, ώστε να γνωρίζει ο προγραμματιστής πού θα πρέπει να επικεντρώσει τις προσπάθειές του για την αύξηση της ταχύτητας εκτέλεσης. Μελετήθηκαν λοιπόν τα διάφορα πρωτόκολλα παρακολούθησης που έχουν προταθεί για το OpenMP και έγινε υλοποίηση ενός από αυτά, του POMP [MMSW01].

Μέρος της εργασίας αποτέλεσε και η βελτίωση της επίδοσης των παραγόμενων προγραμμάτων. Έτσι έγιναν *αλγοριθμικές* βελτιώσεις στην βιβλιοθήκη χρόνου εκτέλεσης και στον κώδικα που παρήγαγε ο συντακτικός αναλυτικής. Επίσης, βελτιστοποιήθηκαν αρκετά σημεία του κώδικα ανάλογα με την *αρχιτεκτονική* του συστήματος εκτέλεσης, λαμβάνοντας υπόψη σημεία όπως η λανθάνουσα μνήμη και η ανάλυση των διαθέσιμων ρολογιών του συστήματος για χρονομέτρηση.

1.5 Δομή της εργασίας

Η εργασία είναι οργανωμένη ως εξής:

Το Κεφάλαιο 2 κάνει μια μικρή εισαγωγή στο OpenMP. Στο ίδιο κεφάλαιο γίνεται επίσης επισκόπηση των μεταγλωττιστών που το υποστηρίζουν, συμπεριλαμβανομένης και της αρχικής υλοποίησης του OMPi.

Στο Κεφάλαιο 3 παρουσιάζεται η ενιαία προγραμματιστική διεπαφή νημάτων που δημιουργήθηκε στα πλαίσια της εργασίας. Παρουσιάζονται οι συναρτήσεις και οι

τύποι δεδομένων που απαιτούνται καθώς επίσης και η ενσωμάτωση των POSIX threads και των Solaris threads.

Στο Κεφάλαιο 4 περιγράφονται οι βελτιστοποιήσεις που έγιναν στον παραγόμενο κώδικα. Περιλαμβάνονται αλγοριθμικές και αρχιτεκτονικές βελτιώσεις στον συντακτικό αναλυτή και στην βιβλιοθήκη χρόνου εκτέλεσης. Οι βελτιστοποιήσεις επιβεβαιώνονται και πειραματικά.

Στο Κεφάλαιο 5 παρουσιάζεται το πρωτόκολλο POMP για την παρακολούθηση της επίδοσης (profiling) των προγραμμάτων OpenMP και η υλοποίηση της διεπαφής του στο OMPi.

Τέλος, στο Κεφάλαιο 6 γίνεται ανασκόπηση της εργασίας καθώς και μία συζήτηση για μελλοντικές κατευθύνσεις.

Κεφάλαιο 2

Εισαγωγή στο OpenMP

Όπως ήδη είπαμε, ο παράλληλος προγραμματισμός είναι συνήθως επίπονη υπόθεση και προϋποθέτει γνώση της υποκείμενης αρχιτεκτονικής και κάποιας παράλληλης γλώσσας ή απαιραίτητων βιβλιοθηκών. Το OpenMP προσπαθεί να δώσει έναν πιο απλό τρόπο προγραμματισμού για το μοντέλο κοινής μνήμης, ο οποίος μάλιστα έχει γίνει ευρύτατα αποδεκτός.

Ο προγραμματιστής δεν χρειάζεται να αλλάξει τον τρόπο σκέψης του. Μπορεί να υλοποιήσει με τις συνηθισμένες μεθόδους το σειριακό πρόγραμμα και στη συνέχεια να προσθέσει οδηγίες παραλληλοποίησης. Το τελικό πρόγραμμα αν μεταγλωττιστεί από έναν compiler που υποστηρίζει τις οδηγίες OpenMP θα δώσει ένα εκτελέσιμο που θα μπορεί να εκτελείται παράλληλα. Επιπλέον όμως ο πηγαίος κώδικας μπορεί να μεταγλωττιστεί και από απλό compiler και να προκύψει σειριακό εκτελέσιμο.

Οι υλοποιήσεις όμως του OpenMP δεν είναι απαραίτητο να ελέγχουν για εξαρτήσεις, συγκρούσεις, αδιέξοδα, συνθήκες ανταγωνισμού ή άλλα προβλήματα τα οποία μπορεί να έχουν ως αποτέλεσμα την λανθασμένη εκτέλεση των προγραμμάτων. Είναι ευθύνη του χρήστη να επιλέξει και να τοποθετήσει τις σωστές οδηγίες και κλήσεις συναρτήσεων ώστε το πρόγραμμά του να έχει σωστή συμπεριφορά.

2.1 Ιστορία του OpenMP

Στις αρχές της δεκαετίας του '90 οι διάφορες εταιρίες κατασκευής και πώλησης παράλληλων συστημάτων κοινής μνήμης παρείχαν παραπλήσιες επεκτάσεις της

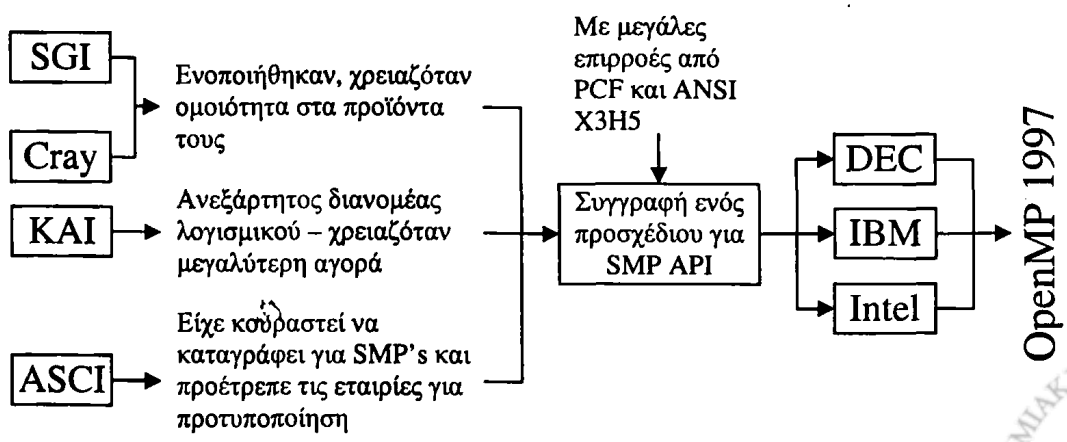
γλώσσας Fortran, με τις οποίες οι χρήστες μπορούσαν να παραλληλοποιήσουν τον κώδικά τους προσθέτοντας οδηγίες προς τον μεταγλωττιστή. Οι οδηγίες αυτές καθόριζαν τους βρόχους που έπρεπε να παραλληλοποιηθούν και ο μεταγλωττιστής αναλάμβανε την αυτόματη κατανομή των επαναλήψεων στους επεξεργαστές του συστήματος.

Οι μεταγλωττιστές αυτοί παρείχαν παρόμοια λειτουργικότητα, με διαφορετική όμως σύνταξη κάνοντας τα προγράμματα μη μεταφέρσιμα. Η πρώτη προσπάθεια προτυποποίησης ήταν το προσχέδιο για το ANSI X3H5 το 1994, το οποίο όμως δεν υιοθετήθηκε ποτέ επίσημα, κυρίως λόγω έλλειψης ενδιαφέροντος αφού τα συστήματα κατανεμημένης μνήμης είχαν γίνει πιο διαδεδομένα.

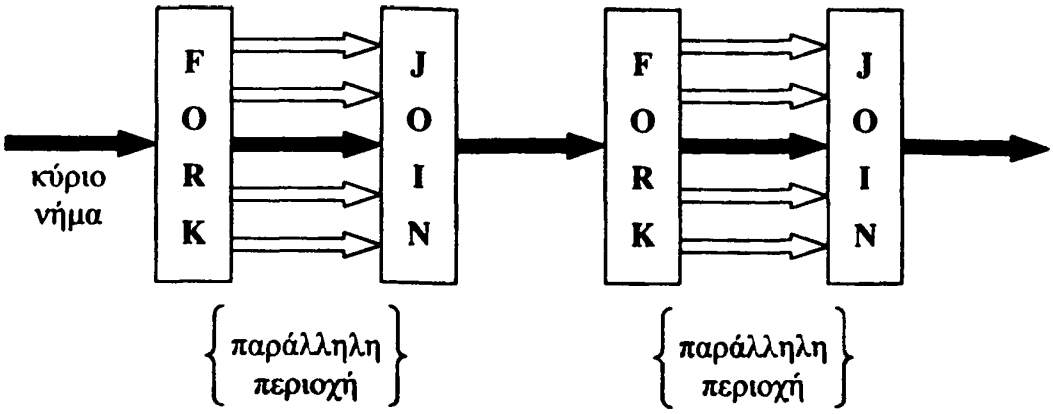
Αργότερα όμως, νεότερες αρχιτεκτονικές κοινής μνήμης άρχισαν να γίνονται δημοφιλείς. Το OpenMP ξεκινά την ιστορία του την άνοιξη του 1997 (Σχήμα 2-1), συνεχίζοντας από εκεί που είχε σταματήσει το X3H5. Οι κύριοι στόχοι της αρχικής του έκδοσης ήταν να ενοποιήσει τις διάφορες παραλλαγές των γλωσσών προγραμματισμού κοινής μνήμης που βασιζόταν σε οδηγίες, ορίζοντας εκ νέου την λειτουργικότητά τους. Η Fortran 90 αγνοήθηκε κατά τον σχεδιασμό, η υποστήριξη για την C ήταν αρκετά καλή ενώ για την C++ προστέθηκε εκ των υστέρων.

Με την έκδοση 1.0 και 2.0 των προδιαγραφών του OpenMP (βλ. και παράγραφο 2.3.6) οι στόχοι επιτευχθήκαν και με το παραπάνω. Η υποστήριξη της Fortran 90 και της C είναι αρκετά καλή, αν και για την C++ είναι σχετικά αδύνατη, χωρίς όμως να υπάρχει ιδιαίτερη ζήτηση για την υποστήριξή της. Οι εταιρίες πλέον στράφηκαν στο OpenMP και οι διάφορες παραλλαγές στις οδηγίες παραλληλισμού κοινής μνήμης έχουν σχεδόν εγκαταλειφθεί.

Το OpenMP Architecture Review Board (OARB) στοχεύει στην ενοποίηση των



Σχήμα 2-1 Δημιουργία της πρώτης έκδοσης του OpenMP



Σχήμα 2-2 Το fork-join μοντέλο παράλληλης εκτέλεσης

προδιαγραφών για Fortran και C/C++ σε ένα κοινό έγγραφο (OpenMP 2.5, [Shah03]), με στόχους

- την εξάλειψη της ανάγκης για συγχρονισμό μεταξύ των προδιαγραφών
- την αποφυγή μεγάλων χρονικών διαστημάτων για την ανανέωση των προδιαγραφών
- την ελάττωση της σύγχυσης για τους προγραμματιστές που μετακινούνται μεταξύ των δύο υποστηριζόμενων γλωσσών προγραμματισμού

Το OARB σχεδιάζει να επεκτείνει το OpenMP στο μέλλον, και προσανατολίζεται να συμπεριλάβει αρκετές ακόμα δυνατότητες στην έκδοση 3.0.

2.2 Περιβάλλον εκτέλεσης

Το OpenMP χρησιμοποιεί το *fork-join* μοντέλο παράλληλης εκτέλεσης (Σχήμα 2-2). Η εκτέλεση ενός προγράμματος OpenMP ξεκινά με ένα μοναδικό νήμα εκτέλεσης, το *νήμα-αφέντη* (master thread). Το νήμα-αφέντης εκτελείται σειριακά ώσπου να συναντηθεί η πρώτη παράλληλη περιοχή (parallel construct). Η παράλληλη περιοχή δηλώνεται στο OpenMP C/C++ API με την οδηγία `#pragma omp parallel`. Το νήμα-αφέντης τότε δημιουργεί μία ομάδα νημάτων (*fork*) και γίνεται αφέντης της ομάδας. Όλα τα νήματα της ομάδας εκτελούν τις εντολές της παράλληλης περιοχής, εκτός από τις περιοχές *καταμερισμού εργασίας* (work-sharing constructs), στις οποίες τα νήματα συνεργάζονται ώστε το καθένα να εκτελέσει ένα μέρος μόνο της συνολικής δουλειάς. Στο τέλος των παράλληλων περιοχών το νήμα-αφέντης περιμένει σε ένα *σημείο φραγής* (barrier) μέχρι να τελειώσουν όλα τα νήματα τις εργασίες τους (*join*) και στη συνέχεια συνεχίζει μόνο του σειριακά την εκτέλεση του προγράμματος.

Αν κάποιο νήμα τροποποιήσει ένα κοινό αντικείμενο, δεν επηρεάζει μόνο το δικό του περιβάλλον εκτέλεσης, αλλά και των υπόλοιπων νημάτων του προγράμματος. Από την οπτική γωνία των υπόλοιπων νημάτων, η αλλαγή αυτή θεωρείται πλήρης πριν αρχίσει να εκτελείται η επόμενη εντολή, αλλά μόνο αν το αντικείμενο αυτό έχει δηλωθεί ως `volatile`. Αλλιώς η αλλαγή γίνεται εμφανής μόνο αφού το νήμα που τροποποίησε το αντικείμενο αλλά και το νήμα που προσπαθεί να το προσπελάσει συναντήσουν μία οδηγία `flush` η οποία να αναφέρει το αντικείμενο (είτε ρητώς είτε έμμεσα).

2.3 Οδηγίες του OpenMP για C/C++

Το OpenMP ορίζει ένα σύνολο οδηγιών για την παραλληλοποίηση των προγραμμάτων, οι οποίες βασίζονται στις οδηγίες `#pragma` των γλωσσών προγραμματισμού C και C++. Η γενική μορφή των οδηγιών του OpenMP είναι:

```
#pragma omp directive-name [clause[ [,] clause]...]
```

Στο τέλος όλων των οδηγιών του OpenMP θα πρέπει να υπάρχει ένα τέλος γραμμής (new-line). Παρουσιάζονται στη συνέχεια εν συντομία μερικές από τις βασικότερες οδηγίες. Περισσότερες πληροφορίες μπορεί να βρει κανείς στις προδιαγραφές των εκδόσεων 1.0 [OARB98] και 2.0 [OARB02] του OpenMP.

2.3.1 Οδηγία parallel

Είδαμε ήδη σε προηγούμενη παράγραφο την έννοια της οδηγίας `parallel`. Η γενική της μορφή είναι

```
#pragma omp parallel [clause[ [,] clause]...]
     structured-block
```

όπου η φράση (*clause*) μπορεί να είναι μία από τις παρακάτω:

- **`if`**`(`*scalar-expression*`)`
- **`private`**`(`*variable-list*`)`
- **`firstprivate`**`(`*variable-list*`)`
- **`default(shared | none)`**
- **`shared`**`(`*variable-list*`)`

- **copyin**(*variable-list*)
- **reduction**(*operator*: *variable-list*)
- **num_threads**(*integer-expression*)

Μόλις ένα νήμα συναντήσει μια παράλληλη περιοχή, δημιουργεί μία ομάδα νημάτων (ο αριθμός τους *μπορεί* να καθορίζεται από την φράση num_threads) και γίνεται αφέντης της ομάδας. Εάν όμως υπάρχει φράση if που η συνθήκη της δεν επαληθεύεται, τότε η αντίστοιχη περιοχή δεν εκτελείται παράλληλα. Οι υπόλοιπες φράσεις αφορούν στον χειρισμό των μεταβλητών που αναφέρονται στην παράλληλη περιοχή και περιγράφονται στην ενότητα 2.3.4.

Σαν παράδειγμα, ο κώδικας

```
#pragma omp parallel num_threads(3)
{
  printf("This is thread %d\n", omp_get_thread_num());
}
```

θα δημιουργήσει τρία νήματα τα οποία θα εκτυπώσουν το παραπάνω μήνυμα και τον αριθμό του νήματος που το τύπωσε. Η συνάρτηση omp_get_thread_num είναι μέρος της βιβλιοθήκης χρόνου εκτέλεσης του OpenMP και επιστρέφει τον αύξοντα αριθμό του νήματος που την καλεί.

2.3.2 Οδηγίες καταμερισμού εργασίας

Σημαντικές είναι επίσης και οι οδηγίες *καταμερισμού εργασίας* του OpenMP. Η οδηγία for προσδιορίζει ότι η *εντολή* for που ακολουθεί θα εκτελεστεί παράλληλα. Οι επαναλήψεις του βρόχου for κατανέμονται στα νήματα που υπάρχουν ήδη στην ομάδα η οποία εκτελεί την παράλληλη περιοχή που περιέχει την οδηγία for. Η σύνταξή της είναι:

```
#pragma omp for [clause[ [,] clause]...]
      for-loop
```

Βέβαια τίθενται κάποιοι περιορισμοί για τον αντίστοιχο βρόχο for της οδηγίας. Θα πρέπει να έχει *κανονική μορφή*, δηλαδή ο μετρητής να είναι ακέραια μεταβλητή, η συνθήκη τερματισμού να είναι <, <=, > ή >=, το βήμα να είναι σταθερό κ.α. Η

κανονική μορφή επιτρέπει στον μεταφραστή να κάνει υπολογισμούς σε σχέση με τον συνολικό αριθμό των επαναλήψεων και το ποσοστό από αυτές που θα πρέπει να αναλάβει κάθε νήμα.

Μία από τις φράσεις που επιτρέπεται στην οδηγία `for` είναι η `schedule(`*kind[, chunk_size*`)`. Αυτή καθορίζει τον τρόπο με τον οποίο θα διανεμηθούν οι επαναλήψεις του βρόχου στα νήματα της ομάδας. Το *kind* μπορεί να είναι:

- `static`: Σε αυτήν την περίπτωση οι επαναλήψεις διαιρούνται σε συνεχόμενα τμήματα (*chunks*) μεγέθους *chunk_size*. Τα τμήματα ανατίθενται στατικά σε νήματα της ομάδας με κυκλικό τρόπο, με βάση την σειρά αρίθμησης των νημάτων. Αν δεν οριστεί το *chunk_size* τότε οι επαναλήψεις διαιρούνται σε τμήματα σχεδόν ίσου μεγέθους και κάθε νήμα αναλαμβάνει ένα τμήμα.
- `dynamic`: Όταν οριστεί η επιλογή `dynamic` οι επαναλήψεις διαιρούνται σε μία σειρά τμημάτων το καθένα από τα οποία περιέχει *chunk_size* επαναλήψεις. Κάθε τμήμα ανατίθεται σε ένα νήμα που βρίσκεται σε αναμονή εργασίας. Το νήμα εκτελεί το τμήμα επαναλήψεων και στην συνέχεια περιμένει για νέα ανάθεση, μέχρι να τελειώσουν τα τμήματα. Φυσικά το τελευταίο τμήμα μπορεί να περιέχει μικρότερο αριθμό επαναλήψεων. Αν δεν δηλωθεί το *chunk_size* θεωρείται 1.
- `guided`: Είναι περίπου σαν το `dynamic` με την διαφορά ότι το πλήθος επαναλήψεων που περιέχει κάθε τμήμα μειώνεται εκθετικά. Η μορφή αυτή θεωρείται ότι κατανέμει καλύτερα τον φόρτο εργασίας ανάμεσα στους επεξεργαστές (*load balancing*).
- `runtime`: όταν ορίζεται `schedule(runtime)` η απόφαση για τον καταμερισμό των επαναλήψεων αναβάλλεται για τον χρόνο εκτέλεσης. Ο τύπος της κατανομής και ο αριθμός των επαναλήψεων μπορούν να οριστούν κατά την εκτέλεση με την μεταβλητή περιβάλλοντος `OMP_SCHEDULE`.

Μία άλλη οδηγία καταμερισμού εργασίας είναι η `sections`, με την εξής σύνταξη:

```
#pragma omp sections [clause[ [,] clause]...]
      {
      [#pragma omp section]
            structured-block
      #pragma omp section
            structured-block
      ...
      }
```

Αυτή είναι χρήσιμη όταν θέλουμε τα νήματα να εκτελέσουν *διαφορετικά* τμήματα κώδικα. Κάθε τμήμα (*structured-block*) εκτελείται μία μόνο φορά από κάποιο νήμα της ομάδας.

Οι οδηγίες `for` και `sections` μπορούν να ομαδοποιηθούν με την `parallel`, ώστε να μην χρειάζεται να γραφούν δύο ξεχωριστές οδηγίες. Για παράδειγμα, στον κώδικα

```
#pragma omp parallel sections
{
      #pragma omp section
            printf("This is section #1\n");
      #pragma omp section
            printf("This is section #2\n");
}
```

οι δύο `printf` θα εκτελεστούν από διαφορετικά νήματα.

Η τελευταία οδηγία καταμερισμού εργασίας είναι η `single`, η οποία καθορίζει ότι ένα μόνο νήμα (όχι απαραίτητα ο αφέντης) θα εκτελέσει το τμήμα κώδικα που ακολουθεί. Η σύνταξή της είναι:

```
#pragma omp single [clause[ [,] clause]...]
      structured-block
```

2.3.3 Οδηγίες συγχρονισμού

Υπάρχουν επίσης και αρκετές οδηγίες συγχρονισμού ή αμοιβαίου αποκλεισμού. Η οδηγία `master` καθορίζει ότι το τμήμα κώδικα που ακολουθεί πρέπει να εκτελεστεί μόνο από το νήμα-αφέντη της ομάδας:

```
#pragma omp master
      structured-block
```

Η οδηγία `critical` περιορίζει την εκτέλεση του συνδεδεμένου τμήματος κώδικα σε ένα νήμα την φορά, δηλαδή ορίζει μια κρίσιμη περιοχή στην οποία μπορεί να βρίσκεται κάθε φορά μόνο ένα νήμα (αμοιβαίος αποκλεισμός).

```
#pragma omp critical [(name)]
      structured-block
```

Προαιρετικά μπορεί να οριστεί ένα *όνομα* για την κρίσιμη περιοχή. Ένα νήμα που προσπαθεί να εισέλθει στην κρίσιμη περιοχή θα πρέπει να περιμένει ώσπου κανένα τμήμα να μην βρίσκεται εντός κρίσιμης περιοχής με το ίδιο όνομα *οπουδήποτε* στο πρόγραμμα. Οι ανώνυμες οδηγίες `critical` θεωρείται ότι έχουν όλες το ίδιο όνομα.

Η οδηγία `barrier` συγχρονίζει όλα τα νήματα μιας ομάδας. Όταν τα νήματα την συναντήσουν, περιμένουν μέχρι να φτάσουν στο σημείο φραγής και όλα τα υπόλοιπα. Η σύνταξή της είναι:

```
#pragma omp barrier
```

Αφού όλα τα νήματα της ομάδας φτάσουν την εντολή `barrier`, η αναμονή σταματά και η εκτέλεση όλων των νημάτων συνεχίζεται από την επόμενη εντολή. Η τοποθέτηση της οδηγίας `barrier` χρειάζεται λίγη προσοχή, επειδή δεν περιλαμβάνει τμήμα κώδικα σαν μέρος της σύνταξής της. Έτσι δεν μπορεί να μπει ακριβώς μετά από μία εντολή `if`, αν δεν εισαχθεί προηγουμένως το άγκιστρο.

Εκτός από τις `barrier` που εισάγονται ρητά στον κώδικα, στο τέλος πολλών οδηγιών υπονοούνται `barriers` για τον συγχρονισμό των νημάτων, όπως για παράδειγμα στις `parallel`, `for`, `sections` και `single`. Για να αποφευχθούν αυτά τα υπονοούμενα `barriers` μπορεί να οριστεί η φράση `nowait`.

Η οδηγία `atomic` χρησιμοποιείται για την ατομική εκτέλεση μίας εντολής:

```
#pragma omp atomic
      expression-stmt
```

Η *expression-stmt* έχει περιορισμένη μορφή, και μπορεί να είναι μόνο μία από τις παρακάτω:

- `x binop= expr`
- `x++`
- `++x`

- `x--`
- `--x`

όπου το x είναι μία μεταβλητή βαθμωτού τύπου και ο *binop* ένας από τα +, *, -, /, &, ^, |, << ή >>. Αν και η `atomic` μοιάζει με την `critical`, οι μεταγλωττιστές που υποστηρίζουν το OpenMP μπορούν να εκμεταλλευτούν τις ατομικές εντολές που υπάρχουν σε κάποιες αρχιτεκτονικές για την παραγωγή αποδοτικότερου κώδικα.

Η οδηγία `flush` καθορίζει ότι στο συγκεκριμένο σημείο του προγράμματος τα νήματα θα πρέπει να έχουν κοινή εικόνα για κάποια αντικείμενα στην μνήμη. Αυτό σημαίνει ότι όλοι οι προηγούμενοι υπολογισμοί που αναφέρονται σε αυτά τα αντικείμενα θα πρέπει να έχουν ολοκληρωθεί και να μην έχουν ακόμα ξεκινήσει οι επόμενοι. Για παράδειγμα, οι μεταγλωττιστές θα πρέπει στο σημείο αυτό να μην χρησιμοποιούν καταχωρητές για την προσωρινή αποθήκευση των αντικειμένων, ενώ το υλικό ίσως χρειαστεί να εκκενώσει κάποιους write buffers ώστε να ενημερωθεί η μνήμη με τις νεότερες τιμές των αντικειμένων αυτών. Η σύνταξή της είναι:

```
#pragma omp flush [(variable-list)]
```

Αν τα αντικείμενα που χρειάζεται να συγχρονιστούν είναι συγκεκριμένα και είναι όλα μεταβλητές (σε αντιδιαστολή με τα objects της C++), τότε μπορούν αυτές να οριστούν σαν παράμετρος της `flush`. Αν δεν υπάρχει λίστα μεταβλητών τότε συγχρονίζονται όλα τα κοινόχρηστα αντικείμενα.

Η οδηγία `flush` υπονοείται (χωρίς να χρειάζεται να δηλωθεί) σε πολλές περιπτώσεις, όπως για παράδειγμα στην είσοδο και στην έξοδο των `critical`, `ordered`, `parallel`, `parallel for` και `parallel sections`.

2.3.4 Περιβάλλον δεδομένων

Το OpenMP παρέχει μία οδηγία και μερικές φράσεις για να δώσει την δυνατότητα στους προγραμματιστές να ελέγξουν τις ιδιότητες διαμοίρασης των μεταβλητών κατά τις παράλληλες περιοχές ή τις περιοχές καταμερισμού εργασίας.

```
#pragma omp threadprivate(variable-list)
```

Οι καθολικές και οι στατικές μεταβλητές ενός προγράμματος είναι εξ ορισμού κοινόχρηστες για όλα τα νήματα. Η οδηγία αυτή παρέχει έναν μηχανισμό για την

δημιουργία ξεχωριστών αντιγράφων των μεταβλητών που δηλώνονται στην *variable-list* για κάθε νήμα. Στην *variable-list* επιτρέπεται να αναφέρονται μεταβλητές διαχωρισμένες με κόμματα, των οποίων ο τύπος έχει προηγουμένως οριστεί πλήρως. Κάθε αντίγραφο μιας `threadprivate` μεταβλητής αρχικοποιείται μία φορά, σε κάποιο ακαθόριστο σημείο του προγράμματος πριν γίνει αναφορά σε αυτό, και με τον συνήθη τρόπο (δηλαδή όπως θα αρχικοποιούταν το κύριο αντίγραφο της μεταβλητής σε σειριακή εκτέλεση του προγράμματος).

Όπως με όλες τις ιδιωτικές μεταβλητές, τα νήματα δεν πρέπει να μεταβάλλουν (μέσω αναφοράς) τα `threadprivate` αντίγραφα των μεταβλητών άλλων νημάτων. Στις σειριακές περιοχές του προγράμματος οι αναφορές γίνονται στο αντίγραφο του νήματος-αφέντη. Μετά το τέλος της πρώτης παράλληλης περιοχής, τα δεδομένα στις `threadprivate` μεταβλητές παραμένουν σταθερά μόνο αν ο μηχανισμός δυναμικών νημάτων έχει απενεργοποιηθεί (βλ. την συνάρτηση `omp_set_dynamic()` στην ενότητα 2.3.5) και αν ο αριθμός των νημάτων μείνει σταθερός για όλες τις παράλληλες περιοχές.

Αρκετές οδηγίες δέχονται *clauses* οι οποίες επιτρέπουν στον χρήστη να καθορίσει τις ιδιότητες διαμοιρασμού των μεταβλητών για την διάρκεια αυτής της περιοχής:

```
private(variable-list)
```

Δηλώνει τις μεταβλητές που αναφέρονται στην λίστα ως τοπικές για κάθε νήμα της ομάδας. Έτσι, μία νέα μεταβλητή με αυτόματη κλάση αποθήκευσης (automatic storage class) δεσμεύεται για κάθε νήμα της ομάδας. Οι νέες μεταβλητές έχουν ακαθόριστες αρχικές τιμές εκτός κι αν πρόκειται για αντικείμενα, οπότε χρησιμοποιείται ο προεπιλεγμένος κατασκευαστής (default constructor). Η αρχική μεταβλητή έχει ακαθόριστη τιμή κατά την είσοδο και έξοδο από την περιοχή και δεν πρέπει να μεταβάλλεται μέσα στην δυναμική έκταση της περιοχής.

```
firstprivate(variable-list)
```

Η `firstprivate` *clause* παρέχει ένα υπερσύνολο των λειτουργιών της `private`. Εκτός από την προαναφερθείσα δημιουργία νέων τοπικών μεταβλητών, κάθε μεταβλητή αρχικοποιείται με την τιμή που είχε η αρχική μεταβλητή πριν την είσοδο στην περιοχή.

```
lastprivate(variable-list)
```

Η `lastprivate` *clause* παρέχει ένα υπερσύνολο των λειτουργιών της `private`. Όταν αυτή χρησιμοποιείται σε μια οδηγία καταμερισμού εργασίας, η τιμή κάθε `lastprivate` μεταβλητής από την ακολουθιακά τελευταία επανάληψη του συνδεδεμένου βρόχου, ή του λεκτικά τελευταίου `section`, ανατίθεται στην αρχική μεταβλητή. Οι μεταβλητές που δεν αναφέρονται στην τελευταία επανάληψη ή το τελευταίο λεκτικά section έχουν ακαθόριστη τιμή μετά το τέλος της περιοχής.

`shared``(`*`variable-list`*`)`

Καθορίζει ότι όλα τα νήματα προσπελαύνουν την ίδια περιοχή μνήμης για κάθε μεταβλητή της *variable-list*, δηλαδή ότι δεν υπάρχουν πολλαπλά αντίγραφα.

`reduction``(`*`op`*`:`*`variable-list`*`)`

Η `reduction` εκτελεί αναγωγή με βάση τον τελεστή *op* για όλες τις μεταβλητές που αναφέρονται στην *variable-list*. Συνήθως ορίζεται για εντολές που έχουν μια από τις παρακάτω μορφές:

- `x = x op expr`
- `x binop= expr`
- `x = expr op x` *(εκτός από την αφαίρεση)*
- `x++`
- `++x`
- `x--`
- `--x`

όπου:

x	Μία από τις `reduction` μεταβλητές που αναφέρονται στην *variable-list*
variable-list	Μία λίστα βαθμωτών μεταβλητών διαχωρισμένων με κόμματα
expr	Μία έκφραση βαθμωτού τύπου που δεν αναφέρεται στο *x*
op	Ένας από τους `+`, `*`, `-`, `&`, `^`, `|`, `&&` ή `||`.
binop	Ένας από τους `+`, `*`, `-`, `&`, `^` ή `|`.

Ανάλογα με τον τελεστή που χρησιμοποιείται γίνεται και η αρχικοποίηση του αθροίσματος (σε 0), του γινομένου (σε 1) κτλ.

```
copyin(variable-list)
```

Παρέχει έναν μηχανισμό για την ανάθεση της ίδιας τιμής σε threadprivate μεταβλητές για κάθε νήμα της ομάδας που εκτελεί μια παράλληλη περιοχή. Για κάθε μεταβλητή που αναφέρεται στην λίστα, αντιγράφεται η τιμή της αντίστοιχης μεταβλητής του νήματος-αφέντη στην αρχή της παράλληλης περιοχής, σαν να χρησιμοποιήθηκε ο τελεστής ανάθεσης τιμής.

```
copyprivate(variable-list)
```

Παρέχει έναν μηχανισμό για την εκπομπή μιας ιδιωτικής μεταβλητής από ένα μέλος της ομάδας σε άλλα μέλη. Είναι μια εναλλακτική λύση αντί για την χρήση κοινών μεταβλητών, όταν η παροχή τέτοιων κοινών μεταβλητών είναι δύσκολη (για παράδειγμα, σε μια αναδρομή που χρειάζεται διαφορετική μεταβλητή για κάθε επίπεδο). Η φράση copyprivate μπορεί να εμφανίζεται μόνο σε οδηγίες single.

2.3.5 Συναρτήσεις βιβλιοθήκης χρόνου εκτέλεσης

Το OpenMP, εκτός από τις οδηγίες #pragma που ορίζει, απαιτεί και την ύπαρξη βιβλιοθήκης χρόνου εκτέλεσης (run-time library) η οποία επιτρέπει στα προγράμματα να προσαρμόσουν με βάση τις ανάγκες τους το περιβάλλον παράλληλης εκτέλεσης. Η βιβλιοθήκη αυτή ονομάζεται <omp.h> και εκτός από την προσαρμογή του περιβάλλοντος παρέχει και συναρτήσεις χρονομέτρησης και αμοιβαίου αποκλεισμού μεταξύ των νημάτων.

```
void omp_set_num_threads(int num_threads);
```

Θέτει τον προεπιλεγμένο αριθμό των νημάτων που θα δημιουργηθούν στις επόμενες παράλληλες περιοχές. Ο αριθμός των νημάτων μπορεί επίσης να οριστεί με την παράμετρο num_threads της οδηγίας parallel και με την μεταβλητή περιβάλλοντος OMP_NUM_THREADS. Η προτεραιότητα των μεθόδων αυτών είναι: πρώτα η num_threads, μετά η omp_set_num_threads και τελευταία η OMP_NUM_THREADS.

```
int omp_get_num_threads(void);
```

Επιστρέφει τον αριθμό των νημάτων της τρέχουσας ομάδας.

```
int omp_get_max_threads(void);
```

Επιστρέφει τον μέγιστο αριθμό των νημάτων που θα μπορούσαν να δημιουργηθούν αν υπήρχε μία οδηγία `parallel` στο σημείο αυτό του κώδικα. Χρησιμεύει για την δέσμευση μνήμης για όλα τα νήματα της ομάδας πριν την είσοδο σε παράλληλες περιοχές.

```
int omp_get_thread_num(void);
```

Επιστρέφει τον αύξων αριθμό του νήματος μέσα στην ομάδα του, από 0 (δηλαδή το νήμα-αφέντης) μέχρι `omp_get_num_threads() - 1`.

```
int omp_get_num_procs(void);
```

Επιστρέφει τον αριθμό των επεξεργαστών που είναι διαθέσιμοι στο πρόγραμμα την στιγμή της κλήσης.

```
int omp_in_parallel(void);
```

Επιστρέφει μη μηδενική τιμή αν κληθεί από μία περιοχή που εκτελείται παράλληλα.

```
void omp_set_dynamic(int dynamic_threads);
```

Η συνάρτηση αυτή αν κληθεί με μη μηδενική παράμετρο ενεργοποιεί την δυναμική επιλογή του αριθμού νημάτων για την εκτέλεση των παράλληλων περιοχών, αφήνοντας το σύστημα να επιλέξει την καταλληλότερη τιμή ανάλογα με τους διαθέσιμους πόρους. Το *dynamic_threads* καθορίζει τον *μέγιστο* αριθμό νημάτων, ενώ αν είναι ίσο με μηδέν η συμπεριφορά αυτή απενεργοποιείται. Ένας δεύτερος τρόπος καθορισμού της δυναμικής επιλογής αριθμού νημάτων είναι μέσω της μεταβλητής περιβάλλοντος `OMP_DYNAMIC`.

```
int omp_get_dynamic(void);
```

Επιστρέφει μη μηδενική τιμή αν η δυναμική επιλογή του αριθμού νημάτων είναι ενεργοποιημένη.

```
void omp_set_nested(int nested);
```

Η συνάρτηση αυτή ενεργοποιεί (για μη μηδενικές τιμές) ή απενεργοποιεί (nested = 0) τον εμφωλευμένο παραλληλισμό. Αν μέσα σε μία οδηγία parallel υπάρχει δεύτερη parallel, τότε συνήθως η τελευταία εκτελείται από ένα μόνο νήμα. Αν ενεργοποιηθεί ο εμφωλευμένος παραλληλισμός *και* αν το υποστηρίζει η υλοποίηση τότε θα δημιουργηθεί νέα ομάδα νημάτων για την εκτέλεση της δεύτερης παράλληλης περιοχής. Ενεργοποίηση του εμφωλευμένου παραλληλισμού μπορεί να γίνει και με την μεταβλητή περιβάλλοντος OMP_NESTED.

```
int omp_get_nested(void);
```

Επιστρέφει μη μηδενική τιμή αν έχει ενεργοποιηθεί ο εμφωλευμένος παραλληλισμός.

Επίσης, στην βιβλιοθήκη χρόνου εκτέλεσης ορίζονται και δέκα συναρτήσεις χειρισμού κλειδαριών:

```
#include <omp.h>

void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int  omp_test_lock(omp_lock_t *lock);

void omp_init_nest_lock(omp_nest_lock_t *nest_lock);
void omp_destroy_nest_lock(omp_nest_lock_t * nest_lock);
void omp_set_nest_lock(omp_nest_lock_t * nest_lock);
void omp_unset_nest_lock(omp_nest_lock_t * nest_lock);
int  omp_test_nest_lock(omp_nest_lock_t * nest_lock);
```

Οι οποίες με τη σειρά αρχικοποιούν, καταστρέφουν, κλειδώνουν, ξεκλειδώνουν ή δοκιμάζουν να κλειδώσουν μία απλή ή εμφωλευμένη (*nested*) κλειδαριά. Στην τελευταία περίπτωση, αν η κλειδαριά είναι ξεκλείδωτη, τότε την κλειδώνουν και επιστρέφουν μη μηδενική τιμή, αλλιώς επιστρέφουν μηδέν χωρίς να μπλοκάρουν την εκτέλεση του προγράμματος.

Οι εμφωλευμένες κλειδαριές επιτρέπεται να ξανακλειδωθούν από το νήμα που τις έχει ήδη κλειδώσει. Έτσι διατηρούν έναν *μετρητή* ο οποίος αυξάνεται κάθε φορά που

το νήμα θέτει την κλειδαριά και μειώνεται σε κάθε ξεκλείδωμα. Ένα άλλο νήμα μπορεί να κλειδώσει την κλειδαριά μόνο αν ο εσωτερικός μετρητής της είναι μηδέν.

Τέλος, στην βιβλιοθήκη `<omp.h>` ορίζονται και δύο συναρτήσεις χρονομέτρησης:

```
double omp_get_wtime(void);
```

Επιστρέφει έναν πραγματικό αριθμό ο οποίος εκφράζει τα δευτερόλεπτα που έχουν περάσει από κάποια αυθαίρετη αλλά σταθερή αφετηρία στο παρελθόν.

```
double omp_get_wtick(void);
```

Επιστρέφει έναν πραγματικό αριθμό ίσο με τον αριθμό των δευτερολέπτων μεταξύ δύο συνεχόμενων χτύπων του ρολογιού.

2.3.6 Διαφορές της έκδοσης 2.0 από την 1.0

Η έκδοση 1.0 των προδιαγραφών του OpenMP είχε σαν κύριο στόχο να ενοποιήσει τις διάφορες παραλλαγές των γλωσσών προγραμματισμού κοινής μνήμης που βασιζόταν σε οδηγίες, ορίζοντας εκ νέου την λειτουργικότητά τους. Η πρώτη έκδοση του προτύπου αφορούσε στην γλώσσα Fortran και δημοσιεύτηκε τον Οκτώβρη του 1997, ενώ μετά από έναν χρόνο ακολούθησε η αντίστοιχη έκδοση για C/C++.

Στην συνέχεια τον Ιούνη του 2000 βγήκε η έκδοση 2.0 του OpenMP για την Fortran και τον Απρίλιο του 2002 η αντίστοιχη έκδοση για C/C++. Οι βασικότερες αλλαγές που έγιναν συνοψίζονται παρακάτω:

- Προστέθηκε η φράση `num_threads` που επιτρέπει στους χρήστες να ζητήσουν συγκεκριμένο αριθμό νημάτων για την εκτέλεση μιας παράλληλης περιοχής
- Η οδηγία `threadprivate` επεκτάθηκε ώστε να δέχεται και στατικές μεταβλητές τοπικής εμβέλειας
- Επιτρέπεται η χρήση πινάκων μεταβλητού μεγέθους της C99 σε λίστες μεταβλητών των φράσεων `private`, `firstprivate`, `lastprivate` κτλ
- Οι `private` μεταβλητές των παράλληλων περιοχών μπορούν να ξαναδηλωθούν ως `private` σε εμφωλευμένες οδηγίες
- Προστέθηκε η φράση `copyprivate` η οποία παρέχει έναν μηχανισμό εκπομπής μιας ιδιωτικής μεταβλητής από ένα μέλος μίας ομάδας στα υπόλοιπα μέλη, και μπορεί να χρησιμοποιηθεί μόνο σε οδηγία `single`

- Προστέθηκαν οι συναρτήσεις χρονομέτρησης `omp_get_wtick` και `omp_get_wtime`, παρόμοια με αυτές που χρησιμοποιούνται στο MPI [MPI94]
- Τέλος, σχολιάστηκε καλύτερα η συμπεριφορά διαφόρων συναρτήσεων

2.4 Υπάρχουσες υλοποιήσεις

Υπάρχουν πολλοί εμπορικοί μεταγλωττιστές που υποστηρίζουν το OpenMP. Οι περισσότεροι από αυτούς υπόκεινται στα κλασσικά μειονεκτήματα του εμπορικού λογισμικού: ο κώδικάς τους είναι κλειστός και δεν επιτρέπουν την ενσωμάτωση διαφορετικών αλγορίθμων για ερευνητικούς σκοπούς, το κόστος προμήθειας είναι συνήθως υψηλό, πολλοί από αυτούς είναι προσανατολισμένοι μόνο σε συγκεκριμένα παράλληλα συστήματα ενώ αρκετοί υποστηρίζουν μόνο την προηγούμενη έκδοση (1.0) του OpenMP. Ενδεικτικά αναφέρονται:

- Fujitsu/Lahey Fortran, C και C++ για λειτουργικά Linux και Solaris
- HP Tru64 Unix για Fortran, C και C++
- Μεταγλωττιστές C++ και Fortran της Intel για συστήματα IA32 και Itanium και για λειτουργικά Linux και Windows
- Guide Fortran και C/C++ από το Intel's KAI Softare Lab για λειτουργικά Linux και Windows
- Μεταγλωττιστές PGF77 και PGF90 από το Portland Group, Inc. (PGI) για λειτουργικά Linux, Solaris και Windows
- Μεταγλωττιστές SGI MIPSpro 7.4 για συστήματα SGI IRIX

Υπάρχουν όμως και κάποιοι ερευνητικοί μεταγλωττιστές ανοιχτού κώδικα. Οι περισσότεροι από αυτούς είναι source-to-source μεταφραστές, δηλαδή δεν μεταγλωττίζουν τον πηγαίο κώδικα σε γλώσσα μηχανής, αλλά σε γλώσσα υψηλού επιπέδου. Για παράδειγμα το OMPi μεταφράζει ένα σειριακό πρόγραμμα σε C με οδηγίες OpenMP σε ισοδύναμο παράλληλο πρόγραμμα πάλι σε γλώσσα C. Το ενδιάμεσο πρόγραμμα στη συνέχεια μετατρέπεται σε γλώσσα μηχανής από τον μεταγλωττιστή του συστήματος.

Από τους ερευνητικούς μεταγλωττιστές που υποστηρίζουν το OpenMP ο μόνος που συνεχίζεται ενεργά (εκτός του OMPi) είναι ο Omni (Omni OpenMP Compiler Project for C and F77, [SSKY99]), ο οποίος είναι ένας αρκετά ολοκληρωμένος source-to-source μεταφραστής που υποστηρίζει Solaris threads, Linux threads, POSIX threads, IRIX sproc, Stackthreads και παρακολούθηση της επίδοσης μέσω του προγράμματος

tlogview ενώ λόγω της χρήσης Java είναι λίγο αργός κατά την μεταγλώττιση. Μέχρι στιγμής υποστηρίζει μόνο το OpenMP 1.0.

Ο Odin [BrBr00] είναι ένας μεταγλωττιστής απλής C (όχι C++) που υποστηρίζει το OpenMP. Ενσωματώθηκε στο INTONE project το οποίο έχει πλέον λήξει. Υποστηρίζει τις αρχιτεκτονικές Origin 2000/3000 με λειτουργικό IRIX και x86 με Linux. Έχει υλοποιημένη την διεπαφή ενορχήστρωσης OMPI [MMHS02] αλλά χωρίς να χρησιμοποιείται κάποια βιβλιοθήκη αποθήκευσης και εμφάνισης των αποτελεσμάτων. Δεν υποστηρίζει όλο το OpenMP 2.0 και επιπλέον ούτε τις `threadprivate` μεταβλητές του 1.0.

Ο μεταγλωττιστής Nanos [MLNA96] υποστηρίζει μόνο Fortran 77 και προσανατολίζεται κυρίως σε αρχιτεκτονικές SGI. Το κύριο βάρος δόθηκε στην ανάπτυξη και χρήση της βιβλιοθήκης NthLib νημάτων επιπέδου χρήστη. Διατίθεται μαζί με γραφικό περιβάλλον προγραμματισμού (HTG-DT) και εργαλεία παρακολούθησης της επίδοσης (Paraver), όμως η ανάπτυξή του έχει πλέον σταματήσει.

2.5 OMPi

Το OMPi είναι ένας ερευνητικός μεταφραστής πηγαίου κώδικα που μετατρέπει ένα σειριακό πρόγραμμα C με οδηγίες OpenMP σε ισοδύναμο παράλληλο πρόγραμμα. Προέκυψε ως αποτέλεσμα προηγουμένων πτυχιακών εργασιών και ήταν ο πρώτος μεταφραστής που υποστήριξε την έκδοση 2.0 του OpenMP [LeTz02]. Η αρχική υλοποίηση του OMPi ήταν λειτουργική, είχε όμως κάποιες ελλείψεις και σε πολλά σημεία δεν ήταν βελτιστοποιημένη, ενώ έκανε αποκλειστικά χρήση της βιβλιοθήκης νημάτων POSIX.

Συγκεκριμένα η έκδοση 0.8.1 του OMPi υποστήριζε όλες τις προδιαγραφές της έκδοσης 2.0 του OpenMP εκτός από κάποια προαιρετικά στοιχεία, όπως τον εμφωλευμένο παραλληλισμό και τη δυναμική προσαρμογή του αριθμού των νημάτων που εκτελούν τις παράλληλες περιοχές, και κάποια ελλιπή σημεία όπως την υποστήριξη εξωτερικών `threadprivate` μεταβλητών. Είχε επίσης μερικά μικρά σφάλματα και έκανε χρήση ορισμένων μη αποδοτικών αλγορίθμων, γενικά όμως μπορούσε να χρησιμοποιηθεί για την κατασκευή παράλληλων προγραμμάτων και σε ορισμένες περιπτώσεις να ξεπεράσει την επίδοση ακόμα και εμπορικών μεταγλωττιστών [DLTz03].

Σε σημαντικό ποσοστό η αρχική υλοποίηση του OMPi δεν διέθετε λεπτομερή σχολιασμό/τεκμηρίωση. Για παράδειγμα, ο συντακτικός αναλυτής αποτελούνταν από 3600 γραμμές κώδικα και έκανε ταυτόχρονα συντακτική ανάλυση και παραγωγή του παράλληλου κώδικα, χωρίς σχολιασμό ούτε των επιπρόσθετων βοηθητικών στοιχείων της γραμματικής ούτε της μεθοδολογίας παραγωγής του κώδικα. Αρχικό μας μέλημα, το οποίο θα αποτελέσει και παρακαταθήκη για μελλοντικές εργασίες επάνω στο OMPi, ήταν η αναδόμηση και τεκμηρίωση ολόκληρου του μεταφραστή.

Το επόμενο βήμα αφορούσε στην απομόνωση της προγραμματιστικής *διεπαφής* (Application Programming Interface, API) *χειρισμού νημάτων* που χρησιμοποιούσε ο μεταφραστής, ώστε να μην περιορίζεται σε νήματα POSIX. Μελετήθηκαν λοιπόν διάφορα πακέτα πολυνηματικού προγραμματισμού αλλά και οι ανάγκες του μεταφραστή ώστε να καθοριστούν με σαφήνεια οι απαιτήσεις που θα έπρεπε να έχει το OMPi από τα πακέτα αυτά. Στη συνέχεια το σύνολο των τύπων και των συναρτήσεων που προέκυψαν δομήθηκε σαν ξεχωριστή βιβλιοθήκη (`othread.h`) και έγιναν οι αντίστοιχες υλοποιήσεις για POSIX και Solaris threads, ενώ το API είναι αρκετά γενικό ώστε να μπορεί να υποστηρίξει μελλοντικά και άλλα πακέτα νημάτων.

Μία ακόμα βελτίωση του OMPi ήταν η αυτόματη ενσωμάτωση *κώδικα παρακολούθησης της επίδοσης* (profiling) στα παραγόμενα εκτελέσιμα αρχεία. Ο παράλληλος προγραμματισμός εμπεριέχει αρκετά προβλήματα συγχρονισμού και βέλτιστης κατανομής των εργασιών ανάμεσα στους επεξεργαστές. Είναι λοιπόν απαραίτητη η παροχή εποπτικών εργαλείων που θα κάνουν δυνατή την ανεύρεση των «προβληματικών» σημείων του κώδικα, ώστε να γνωρίζει ο προγραμματιστής πού θα πρέπει να επικεντρώσει τις προσπάθειές του για την αύξηση της ταχύτητας εκτέλεσης. Μελετήθηκαν λοιπόν τα διάφορα πρωτόκολλα παρακολούθησης που έχουν προταθεί για το OpenMP και έγινε υλοποίηση ενός από αυτά, του POMP [MMSW01].

Στα πλαίσια της εργασίας έγιναν και αρκετές αλγοριθμικές βελτιώσεις, όπως η προσθήκη λογικής για ανίχνευση και αφαίρεση των περιττών σημείων φραγής, η κατασκευή αλγορίθμου για την οδηγία `ordered` χωρίς την χρήση condition variables, η χρησιμοποίηση των εμφωλευμένων κλειδαριών που προσφέρουν τα pthreads και η δήλωση των συναρτήσεων χειρισμού των barriers ως weak symbols για να μπορεί εύκολα ο χρήστης να δοκιμάζει δικούς του αλγορίθμους. Τέλος, έγιναν διάφορες βελτιστοποιήσεις ανάλογα με την αρχιτεκτονική του συστήματος, όπως για παράδειγμα η υλοποίηση της οδηγίας `flush` και της συνάρτησης `omp_get_wtime`,

ενώ τροποποιήθηκε ο κώδικας ώστε να λαμβάνει υπόψη του την λανθάνουσα μνήμη του συστήματος.

Κεφάλαιο 3

Παραγωγή πολυνηματικού κώδικα

Ο σκοπός του OMPi είναι η μετατροπή ενός σειριακού προγράμματος με οδηγίες OpenMP σε ισοδύναμο παράλληλο πρόγραμμα, το οποίο να μπορεί να εκτελεστεί σε υπολογιστές με περισσότερους από έναν επεξεργαστή, αυξάνοντας έτσι την ταχύτητα εκτέλεσής του. Όπως αναφέρθηκε στην παράγραφο 1.2, υπάρχουν πολλές μέθοδοι προγραμματισμού παράλληλων συστημάτων κοινής μνήμης.

Το OpenMP φυσικά δεν καθορίζει με ποιον τρόπο θα υλοποιηθεί τελικά η παράλληλη εκτέλεση του προγράμματος. Η συνάρτηση `fork` του Unix που δημιουργεί αντίγραφα μιας διεργασίας, οι ελαφρές διεργασίες (lightweight processes) και τα νήματα είναι μερικοί μόνο από τους τρόπους με τους οποίους θα μπορούσε να επιτευχθεί η παραλληλοποίηση. Φυσικά θα έπρεπε να επιλεγεί ο πιο αποδοτικός από τους παραπάνω τρόπους, με κριτήρια την ελαχιστοποίηση της κατανάλωσης των πόρων του συστήματος και την μεγιστοποίηση της ταχύτητας.

Η κατασκευή και διαχείριση διεργασιών είναι γενικά χρονοβόρα διαδικασία, αφού απαιτεί την κατασκευή αντιγράφων για την μνήμη στην οποία φυλάσσεται ο κώδικας, την στοίβα, την μνήμη των δεδομένων, τον σωρό καθώς και την δημιουργία των σχετικών δομών από το λειτουργικό σύστημα (Process ID, Group ID, User ID, File Descriptors, Locks, Sockets). Αντίθετα, η δημιουργία νημάτων είναι δεκάδες φορές γρηγορότερη: για ένα νήμα χρειάζεται μόνο να δημιουργηθεί η στοίβα και οι αντίστοιχες δομές χρονοδρομολόγησης. Εξάλλου η επικοινωνία μεταξύ διεργασιών είναι λιγότερο αποδοτική από την επικοινωνία μεταξύ νημάτων, αφού τα τελευταία μοιράζονται τον ίδιο χώρο διευθύνσεων. Τέλος, η εναλλαγή μεταξύ νημάτων είναι

γρηγορότερη από την εναλλαγή μεταξύ διεργασιών επειδή δεν χρειάζεται να γίνει αλλαγή του χώρου διευθύνσεων αλλά και λόγω λιγότερων πληροφοριών του περιβάλλοντος (context).

Έτσι η υλοποίηση του OMPi προσανατολίστηκε στη χρήση νημάτων. Στην προηγούμενη έκδοσή του το OMPi προϋπέθετε την ύπαρξη της βιβλιοθήκης νημάτων POSIX. Αν και τα περισσότερα συστήματα υποστηρίζουν πλέον τα νήματα POSIX, λόγω της γενικότητάς τους δεν εκμεταλλεύονται πάντα πλήρως την αρχιτεκτονική του συστήματος. Θα μπορούσε λοιπόν κανείς να χρησιμοποιήσει άλλες βιβλιοθήκες νημάτων ή μηχανισμούς που να είναι αποδοτικότεροι για κάθε συγκεκριμένο σύστημα. Γι' αυτόν τον λόγο αποφασίστηκε η απομόνωση του συνόλου των τύπων δεδομένων και των συναρτήσεων που απαιτούσε το OMPi από την βιβλιοθήκη νημάτων σε μια ξεχωριστή βιβλιοθήκη (`othread.h`). Με αυτή την δομή η υποστήριξη επιπλέον βιβλιοθηκών νημάτων μπορεί να γίνεται χωρίς να χρειάζεται ο προγραμματιστής να επέμβει ή να χρειάζεται να ξέρει τον υπόλοιπο πηγαίο κώδικα..

3.1 Προγραμματιστική διεπαφή νημάτων

Ο σχεδιασμός της ενοποιημένης βιβλιοθήκης νημάτων προέκυψε από τις ανάγκες του μεταφραστή. Χρειαζόμαστε τύπους και συναρτήσεις για την δημιουργία και τον χειρισμό νημάτων, για τον ορισμό του βαθμού παραλληλίας, για την διαχείριση διαφόρων ειδών κλειδαριών αλλά και κάποιον μηχανισμό για πρόσβαση σε τοπικό αποθηκευτικό χώρο. Όπου ήταν δυνατό, διατηρήθηκε η σημασιολογία των νημάτων POSIX, αν και αφαιρέθηκαν τα πεδία των τύπων και οι παράμετροι των συναρτήσεων που δεν χρησιμοποιούνται από τον μεταφραστή. Ο στόχος ήταν η διεπαφή να είναι αρκετά μινιμαλιστική και ευέλικτη ώστε να μπορεί να υλοποιηθεί εύκολα με διαφορετικά πακέτα νημάτων, χωρίς όμως να χάνονται πολύτιμες δυνατότητες βελτιστοποίησης κατά την παραγωγή του πολυνηματικού κώδικα.

Οι τύποι δεδομένων που ορίζονται στην βιβλιοθήκη `othread.h` είναι:

Τύπος	Έννοια
`othread_t`	Αναπαριστά ένα νήμα
`othread_attr_t`	Οι ιδιότητες δημιουργίας ενός νήματος
`othread_key_t`	Κλειδί πρόσβασης σε τοπικό αποθηκευτικό χώρο
`othread_lock_t`	Απλή κλειδαριά
`othread_nest_lock_t`	Εμφωλευμένη κλειδαριά
`othread_spin_lock_t`	Κλειδαριά με busy-waiting

Ο τύπος `othread_spin_lock_t` θα πρέπει να χρησιμοποιείται σε περιπτώσεις όπου η αναμονή πριν την απόκτηση της κλειδαριάς είναι ιδιαίτερα μικρή και δεν δικαιολογεί τον φόρτο της αναστολής του νήματος και της προσθήκης του σε ουρά χρονοδρομολόγησης. Η υλοποίησή του (και των αντίστοιχων συναρτήσεων) εξαρτάται από την αρχιτεκτονική του κάθε συστήματος, και θα πρέπει είτε να εκμεταλλεύεται αντίστοιχες κλειδαριές που προσφέρει το λειτουργικό σύστημα, είτε να γίνεται σε γλώσσα assembly, ή, στην χειρότερη περίπτωση να εξομοιώνεται με απλή κλειδαριά.

Οι συναρτήσεις που ορίζονται στην βιβλιοθήκη `othread.h` φαίνονται παρακάτω:

```
int othread_create(othread_t *thread, const othread_attr_t
          *attr, void *(*start_routine)(void *), void *arg);
```

Η συνάρτηση `othread_create` δημιουργεί ένα νέο νήμα εκτέλεσης με τις ιδιότητες `attr`, το οποίο θα εκτελέσει την ρουτίνα `start_routine` με παράμετρο `arg`. Αν το νήμα δημιουργήθηκε επιτυχώς επιστρέφει μηδέν.

```
int othread_equal(othread_t t1, othread_t t2);
```

Επιστρέφει 1 αν οι παράμετροι αναφέρονται στο ίδιο νήμα, διαφορετικά 0.

```
void *othread_getspecific(othread_key_t key);
```

Η συνάρτηση αυτή ανακτά την τοπική αποθηκευτική μνήμη ενός νήματος που έχει αντιστοιχηθεί με το κλειδί `key`. Αν το κλειδί δεν είναι έγκυρο επιστρέφει NULL.

```
int othread_key_create(othread_key_t *key,
                       void (*destructor)(void *));
```

Η `othread_key_create` δημιουργεί ένα κλειδί τοπικής αποθηκευτικής μνήμης για την διεργασία και συνδέει με αυτό την συνάρτηση καταστροφής `destructor`. Μετά την δημιουργία του, το κλειδί μπορεί να χρησιμοποιηθεί για την ανάθεση και ανάκτηση ενός δείκτη σε τοπικά δεδομένα για κάθε νήμα. Επιστρέφει 0 σε επιτυχία.

```
othread_t othread_self(void);
```

Επιστρέφει τον αριθμό χειρισμού (handle) του καλώντος νήματος.

```
int othread_setconcurrency(int concurrency);
```

Δηλώνει στο σύστημα ότι η εφαρμογή προτιμά να εκτελεστεί σε concurrency αριθμό επεξεργαστών. Με μηδενική παράμετρο το σύστημα επιλέγει μόνο του τον βαθμό παραλληλισμού.

```
int othread_setspecific(othread_key_t key,
                        const void *value);
```

Θέτει την τιμή τοπικής αποθήκευσης που έχει αντιστοιχηθεί με το κλειδί key. Χρησιμοποιείται όποτε τα νήματα χρειάζονται ιδιωτικό αποθηκευτικό χώρο.

Επίσης ορίζονται αρκετές συναρτήσεις χειρισμού κλειδαριών, οι οποίες έχουν παρόμοιο νόημα με τις αντίστοιχες συναρτήσεις του OpenMP που παρουσιάστηκαν στην παράγραφο 2.3.5:

```
int othread_init_lock(othread_lock_t *lock);
int othread_destroy_lock(othread_lock_t *lock);
int othread_set_lock(othread_lock_t *lock);
int othread_unset_lock(othread_lock_t *lock);
int othread_test_lock(othread_lock_t *lock);

int othread_init_nest_lock(othread_nest_lock_t *lock);
int othread_destroy_nest_lock(othread_nest_lock_t *lock);
int othread_set_nest_lock(othread_nest_lock_t *lock);
int othread_unset_nest_lock(othread_lock_t *lock);
int othread_test_nest_lock(othread_nest_lock_t *lock);

int othread_init_spin_lock(othread_spin_lock_t *lock);
int othread_destroy_spin_lock(othread_spin_lock_t *lock);
int othread_set_spin_lock(othread_spin_lock_t *lock);
int othread_unset_spin_lock(othread_spin_lock_t *lock);
int othread_test_spin_lock(othread_spin_lock_t *lock);
```

3.2 Posix threads

Η βιβλιοθήκη των νημάτων POSIX [IEEE96] προδιαγράφει περισσότερες από 60 συναρτήσεις για την δημιουργία, χρονοδρομολόγηση, δέσμευση τοπικής μνήμης, αποσύνδεση από την κύρια διεργασία, ακύρωση και συγχρονισμό μεταξύ των

νημάτων. Είναι πλέον ευρύτατα διαδεδομένη, γι' αυτό και η διεπαφή της χρησιμοποιήθηκε σαν γνώμονας κατά τον σχεδιασμό της `othread.h`.

Εκτός όμως από τον χειρισμό νημάτων, η IEEE έχει ορίσει κι άλλα πρότυπα, όπως το Real Time Extension [IEEE93] και το Advanced Real Time Extension [IEEE00], τα οποία προσφέρουν υποστήριξη για barriers, recursive locks, spin locks κτλ, δηλαδή τύπους και συναρτήσεις που χρειαζόμαστε για την υλοποίηση της ενιαίας βιβλιοθήκης νημάτων. Το πρόβλημα είναι ότι αυτά τα πρότυπα δεν υποστηρίζονται από όλα τα συστήματα, ενώ κάποια λειτουργικά παρέχουν μερικές μη φορητές (Non Portable, NP) επεκτάσεις. Για παράδειγμα το IRIX υποστηρίζει spin locks (`PTHREAD_SPINBLOCK_MUTEX`) και το Linux υποστηρίζει αναδρομικές κλειδαριές, αλλά με μη φορητό τρόπο (`PTHREAD_MUTEX_RECURSIVE_NP`). Επομένως η υλοποίηση της βιβλιοθήκης `othread.h` λαμβάνει υπόψη της το λειτουργικό σύστημα, χρησιμοποιώντας όσες δομές και συναρτήσεις παρέχονται και εξομοιώνοντας τις υπόλοιπες.

Επειδή σε πολλά συστήματα δεν είναι διαθέσιμο το πλήρες pthread API, επιλέχθηκε προσεκτικά ένα υποσύνολο το οποίο συνήθως υποστηρίζεται. Επίσης, επειδή οι πολυνηματικές εφαρμογές θα πρέπει να χρησιμοποιούν μόνο thread-safe συναρτήσεις, δηλώνεται (με `#define`) το σύμβολο `_MULTI_THREADED` πριν την συμπερίληψη του `<pthread.h>`.ώστε να επεκτείνεται κατά το δυνατόν η υποστήριξη για πολυνηματικές εφαρμογές από τις συναρτήσεις του συστήματος. Τέλος, το πρόγραμμα μεταγλώττισης ompicc φροντίζει να συνδέσει το αντικείμενο (object) πρόγραμμα με την βιβλιοθήκη pthread.

3.3 Solaris threads

Τα νήματα Solaris [SUN02] προϋπήρχαν και μάλιστα επηρέασαν αρκετά τον καθορισμό του προτύπου POSIX threads. Η διεπαφή και λειτουργικότητά τους είναι αρκετά παρόμοια με αυτές των pthreads, αν και τα Solaris threads υποστηρίζουν συναρτήσεις που δεν υπάρχουν στα pthreads και αντίθετα. Παρόλα αυτά είναι δυνατή η μίξη κλήσεων σε Solaris και POSIX threads.

Οι βασικότερες διαφορές μεταξύ τους συνοψίζονται στον παρακάτω πίνακα:

Solaris Threads (libthread)	**POSIX Threads (libpthread)**
Πρόθεμα `thr_` για τα ονόματα των	Πρόθεμα `pthread_` για τα ονόματα των

συναρτήσεων νημάτων και sema_ για τις συναρτήσεις σηματοφόρων	συναρτήσεων νημάτων και sem_ για τις συναρτήσεις σηματοφόρων
Δυνατότητα δημιουργίας νημάτων-δαιμόνων (daemon threads)	Δυνατότητες ακύρωσης νημάτων
Παύση και συνέχιση νημάτων	Πολιτικές χρονοδρομολόγησης

Τα Solaris threads υποστηρίζουν αναδρομικά mutexes, τα οποία μπορούν να χρησιμοποιηθούν για την αποδοτική υλοποίηση των εμφωλευμένων κλειδαριών που ορίζονται στο OpenMP. Όμως, μετά από πειραματισμό διαπιστώσαμε ότι αν και στην έκδοση 5.9 του λειτουργικού SunOs οι εμφωλευμένες κλειδαριές που υλοποιήθηκαν με αναδρομικά mutexes δούλευαν κανονικά, στην έκδοση 5.8 είχαν προβλήματα: αν ένα νήμα προσπαθούσε να κλειδώσει εκ νέου μια εμφωλευμένη κλειδαριά που είχε ήδη κλειδώσει προηγουμένως, διέκοπτε την εκτέλεσή του επ' άπειρο. Έτσι στην περίπτωση των Solaris threads αποφύγαμε τα αναδρομικά mutexes και υλοποιήσαμε τις εμφωλευμένες κλειδαριές με μεταβλητές συνθήκης.

Η επιλογή των νημάτων Solaris γίνεται κατά την εγκατάσταση του OMPi. Αναλυτικές οδηγίες δίνονται στο παράρτημα Α.

Κεφάλαιο 4

Βελτιστοποιήσεις

Αυτό το κεφάλαιο αναφέρεται σε ορισμένους αλγορίθμους που βελτιστοποιήθηκαν στα πλαίσια της συγκεκριμένης εργασίας, καθώς και σε βελτιώσεις που έγιναν στον παραγόμενο κώδικα ανάλογα με την αρχιτεκτονική του συστήματος εκτέλεσης. Συγκεκριμένα, τροποποιήθηκε ο μεταφραστής ώστε να αφαιρεί τα περιττά σημεία φραγής, επιταχύνοντας έτσι την εκτέλεση, ενώ δόθηκε και η δυνατότητα στους χρήστες να δοκιμάζουν τους δικούς τους αλγορίθμους χωρίς να επεμβαίνουν στον πηγαίο κώδικα του OMPi. Επίσης, εφαρμόστηκε διαφορετικός αλγόριθμος για την υλοποίηση της οδηγίας `ordered`, ώστε να μην γίνεται χρήση των αργών μεταβλητών συνθήκης (condition variables). Ακόμη, έγιναν αρκετές βελτιστοποιήσεις στον παραγόμενο κώδικα ώστε να λαμβάνει υπόψη του την λανθάνουσα μνήμη, ενώ υλοποιήθηκε ορθά η οδηγία `flush`.

4.1 Σημεία φραγής

Ο χειρισμός των barriers, είτε ρητώς δηλωμένων είτε υπονοούμενων, υλοποιείται με κλήσεις στις τρεις παρακάτω συναρτήσεις:

```
void omp_barrier_init(ompi_barrier_t *bar, int n)
```

Αρχικοποιεί το σημείο φραγής `bar`. Ουσιαστικά δεσμεύει `n` κλειδαριές στις οποίες θα «κοιμηθούν» τα νήματα που συμμετέχουν στο barrier μέχρι όλα τους να φτάσουν στο σημείο αυτό. Οι κλειδαριές κλειδώνονται εξαρχής ώστε μόλις κάποιο από τα νήματα ξαναπροσπαθήσει να τις κλειδώσει, να κοιμηθεί.

```
void omp_barrier_destroy(ompi_barrier_t *bar)
```

Αποδεσμεύει το barrier, ελευθερώνοντας την μνήμη που δεσμεύτηκε.

```
void omp_barrier_wait(ompi_barrier_t *bar)
```

Καλείται από τα νήματα για συγχρονισμό. Ο αλγόριθμος του σημείου φραγής είναι αρκετά απλός. Καταρχάς κάθε πρόσβαση στις εσωτερικές μεταβλητές του barrier σειριοποιείται μέσω μίας κλειδαριάς. Κάθε νήμα που φτάνει στο σημείο φραγής μειώνει έναν εσωτερικό μετρητή, ο οποίος έχει αρχικοποιηθεί σε n, και στη συνέχεια προσπαθεί να κλειδώσει την *ήδη κλειδωμένη* προσωπική του κλειδαριά ώστε να κοιμηθεί. Τελικά το τελευταίο νήμα που φτάνει στο σημείο φραγής μηδενίζει τον εσωτερικό μετρητή, οπότε και αναλαμβάνει να τον επαναφέρει στην αρχική του τιμή (για να μπορεί να ξαναχρησιμοποιηθεί το barrier) και να ξυπνήσει όλα τα υπόλοιπα νήματα, ξεκλειδώνοντας τις κλειδαριές στις οποίες κοιμούνται.

Οι παραπάνω συναρτήσεις έχουν δηλωθεί ως *weak symbols*, ώστε να μπορεί ο χρήστης αν θέλει να χρησιμοποιήσει τον δικό του αλγόριθμο barrier χωρίς να επέμβει στα ενδότερα του OMPi. Αρκεί να δηλώσει και να υλοποιήσει στον δικό του κώδικα τις παρακάτω συναρτήσεις:

```
void omp_barrier_init(void *bar, int n);
void omp_barrier_wait(void *bar);
void omp_barrier_destroy(void *bar);
```

Το `bar` εδώ δηλώνεται ως `void *` αφού ο χρήστης δεν έχει πρόσβαση στον εσωτερικό τύπο `ompi_barrier_t`. Θα πρέπει λοιπόν να δηλώσει την δική του δομή `barrier_t` και να κάνει typecasting σε `void *` όπου χρειάζεται. Φυσικά ο χρήστης είναι υπεύθυνος για την ορθότητα του αλγορίθμου που χρησιμοποιεί.

4.1.1 Αφαίρεση περιττών σημείων φραγής

Μερικές βελτιστοποιήσεις σε προγράμματα OpenMP που μπορούν να γίνουν από την πλευρά του μεταγλωττιστή έχουν προταθεί στο [Mull01]. Μία πολύ βασική από αυτές είναι η αφαίρεση των περιττών σημείων φραγής. Για παράδειγμα, οι χρήστες αρκετά συχνά μπορεί να γράψουν κώδικα με την παρακάτω μορφή:

```
#pragma omp parallel
{
      [κώδικας]
      #pragma omp for
      {
            [κώδικας]
      }
}
```

όπου δηλαδή δύο οδηγίες *με υπονοούμενα barriers* τελειώνουν στο ίδιο ακριβώς σημείο. Στο τέλος δηλαδή αυτού του κώδικα οι μεταγλωττιστές που υποστηρίζουν το OpenMP θα παράγουν δύο barriers, και φυσικά το ένα από αυτά είναι περιττό.

Μία λύση είναι να μετατεθεί το βάρος στον προγραμματιστή, ο οποίος στο συγκεκριμένο παράδειγμα θα μπορούσε να προσθέσει την φράση `nowait` στην `for` για να αποφύγει το επιπρόσθετο barrier. Όμως ο παράλληλος προγραμματισμός είναι αρκετά δύσκολη υπόθεση και οι μεταφραστές θα πρέπει να αναλαμβάνουν οι ίδιοι τον περισσότερο φόρτο, όπου αυτό είναι δυνατόν. Έτσι, ο συντακτικός αναλυτής τροποποιήθηκε ώστε να θυμάται πότε παρήγαγε τον τελευταίο κώδικα για barrier, και κατά το κλείσιμο των οδηγιών αν διαπιστώσει ότι έχει ήδη εισαχθεί σημείο φραγής ακριβώς πριν το κλείσιμο της οδηγίας τότε αποφεύγει να παράγει και άλλο barrier.

Στην περίπτωση της οδηγίας `parallel` βέβαια η υλοποίηση ήταν ελαφρώς δυσκολότερη. Τυπικά, στο τέλος των παράλληλων περιοχών θα πρέπει να υπάρχει ένα σημείο φραγής και στη συνέχεια να καταστραφούν όλα τα νήματα εκτός του αφέντη. Για την βελτίωση όμως της επίδοσης, στο OMPi ακολουθήσαμε διαφορετική προσέγγιση: αντί να καταστρέφονται, τα νήματα μπαίνουν σε έναν χώρο ανακύκλωσης (pool) όπου και κοιμούνται μέχρι τη στιγμή που θα ξαναχρειαστεί να δημιουργηθεί ένα νέο νήμα, οπότε και ξυπνάμε κάποιο από αυτά. Με αυτήν την μέθοδο αφενός αποφεύγουμε την χρονοβόρα καταστροφή και δημιουργία νημάτων και αφετέρου δεν χρειαζόμαστε σημείο φραγής στο τέλος των παράλληλων περιοχών, αφού η αναμονή μέχρι να μπουν όλα τα νήματα (εκτός του αφέντη) στον χώρο ανακύκλωσης ισοδυναμεί με σημείο φραγής. Έτσι προστέθηκε κώδικας για την αφαίρεση του *προηγουμένου* σημείου φραγής το οποίο είχε ήδη εισαχθεί, δηλαδή του `for` στο παράδειγμά μας.

Η βελτίωση των επιδόσεων είναι σημαντική όπως θα δούμε παρακάτω στα πειραματικά αποτελέσματα (παράγραφος 4.5.2).

4.2 Ordered

Η οδηγία `ordered` μπορεί να χρησιμοποιηθεί μόνο στο εσωτερικό οδηγιών (και βρόχων) `for` και η λειτουργία της μπορεί να περιγραφεί καλύτερα μέσα από ένα παράδειγμα:

```
#pragma omp parallel for ordered
{
  for (i = 0; i < N; i++) {
    statements...
    #pragma omp ordered
    printf("This is loop %d\n", i);
    statements...
  }
}
```

Οι επαναλήψεις της εντολής `for` του παραδείγματος θα κατανεμηθούν μεταξύ των διαθέσιμων νημάτων. Χωρίς την οδηγία `ordered`, οι επαναλήψεις εκτελούνται παράλληλα και επομένως τα μηνύματα της εντολής `printf` μπορεί να εμφανιστούν με οποιαδήποτε σειρά. Η `ordered` καθορίζει ότι αν και οι επαναλήψεις της `for` μπορεί να εκτελεστούν με οποιαδήποτε σειρά, οι εντολές που βρίσκονται στο εσωτερικό της `ordered` θα εκτελεστούν σειριακά (και επομένως τα μηνύματα της `printf` θα εμφανιστούν με αύξουσα σειρά). Αυτό φυσικά ωφελεί περισσότερο όταν οι εντολές γύρω από την `ordered` είναι πιο χρονοβόρες από το εσωτερικό της.

Η προηγούμενη υλοποίηση της οδηγίας `ordered` βασιζόταν σε *μεταβλητές συνθήκης* (condition variables), οι οποίες γενικά είναι αργές και επιβράδυναν την εκτέλεση. Επιπλέον ανάγκαζαν την βιβλιοθήκη `othread.h` να απαιτεί την ύπαρξη μεταβλητών συνθήκης από το υποκείμενο πακέτο νημάτων, χωρίς αυτό να είναι απαραίτητο. Επίσης υπήρχαν προβλήματα σε κάποιες εκδόσεις λειτουργικών συστημάτων όπου οι μεταβλητές συνθήκης δεν ήταν σωστά υλοποιημένες. Έτσι έγινε αντικατάστασή τους με απλές κλειδαριές, σύμφωνα με τον αλγόριθμο που περιγράφεται στη συνέχεια.

Για κάθε `ordered` δεσμεύουμε δυναμικά αριθμό κλειδαριών ίσο με τον αριθμό των νημάτων της ομάδας. Η δυναμική δέσμευση γίνεται για να καλύπτεται η περίπτωση που διαφορετικές ομάδες νημάτων εκτελούν ταυτόχρονα την ίδια ordered (αναφέρονται ως SSF rounds στον πηγαίο κώδικα). Η χρήση των κλειδαριών αυτών είναι απλά για να κοιμούνται τα νήματα που περιμένουν την σειρά τους. Η σειρά των νημάτων καθορίζεται με βάση την «καθολική» μεταβλητή `ordered_lb` (lower

bound) της αντίστοιχης `for`, η οποία εκφράζει την τρέχουσα τιμή του μετρητή της, και τις ιδιωτικές μεταβλητές `lb` που εκφράζουν την τρέχουσα επανάληψη που εκτελεί κάθε νήμα.

Αρχικοποιούμε όλες τις κλειδαριές. Για να κοιμηθεί λοιπόν ένα νήμα θα πρέπει να κλειδώσει *δύο* φορές την κλειδαριά του. Τα νήματα που *μπαίνουν* στην ordered κάνουν με τη σειρά τα εξής βήματα:

1. Κλειδώνουν την κλειδαριά ατομικής πρόσβασης στις πληροφορίες της `ordered`.
2. Ελέγχουν αν έχει έρθει η σειρά τους (`lb` < ή > του `ordered_lb`). Αν ναι, πηγαίνουν στο βήμα 8.
3. Δοκιμάζουν να κλειδώσουν την κλειδαριά τους. Έτσι βεβαιώνονται ότι έχει κλειδωθεί χωρίς ακόμα να κοιμηθούν.
4. Ξεκλειδώνουν την κλειδαριά ατομικής πρόσβασης
5. Κλειδώνουν την κλειδαριά τους για να *προσπαθήσουν* να κοιμηθούν.
6. Ξεκλειδώνουν την κλειδαριά τους για να είναι έτοιμη για επόμενες επαναλήψεις.
7. Κλειδώνουν την κλειδαριά ατομικής πρόσβασης και πηγαίνουν ξανά στο βήμα 2.
8. Ξεκλειδώνουν την κλειδαριά ατομικής πρόσβασης και προχωρούν με την εκτέλεσης της `ordered`.

Ο λόγος που ο παραπάνω αλγόριθμος έχει πολλούς ελέγχους βρίσκεται ανάμεσα από τα βήματα 4 και 5. Επειδή δεν είναι δυνατόν να ξεκλειδωθεί η κλειδαριά ατομικής πρόσβασης και *ταυτόχρονα* να κοιμηθεί κάποιο νήμα, ανάμεσα από τα βήματα αυτά είναι πιθανό να έχει έρθει η σειρά τους και να *μην πρέπει να κοιμηθούν*. Σε αυτήν την περίπτωση το νήμα που μόλις τελείωσε την τελευταία επανάληψη της `ordered` θα αυξήσει μεν το `ordered_lb` αλλά επίσης θα ξεκλειδώσει τις κλειδαριές όλων των νημάτων. Έτσι το επόμενο νήμα δεν θα κοιμηθεί φτάνοντας στο βήμα 5 αλλά θα προχωρήσει στο βήμα 6.

Δηλαδή τα βήματα που κάνουν τα νήματα κατά την έξοδο από την `ordered` είναι:

1. Κλειδώνουν την κλειδαριά ατομικής πρόσβασης στις πληροφορίες της `ordered`.
2. Αυξάνουν ή μειώνουν το `ordered_lb` ανάλογα με το βήμα της `for`.
3. Ξεκλειδώνουν *μία φορά* όλες τις κλειδαριές που πιθανώς κοιμούνται τα υπόλοιπα νήματα

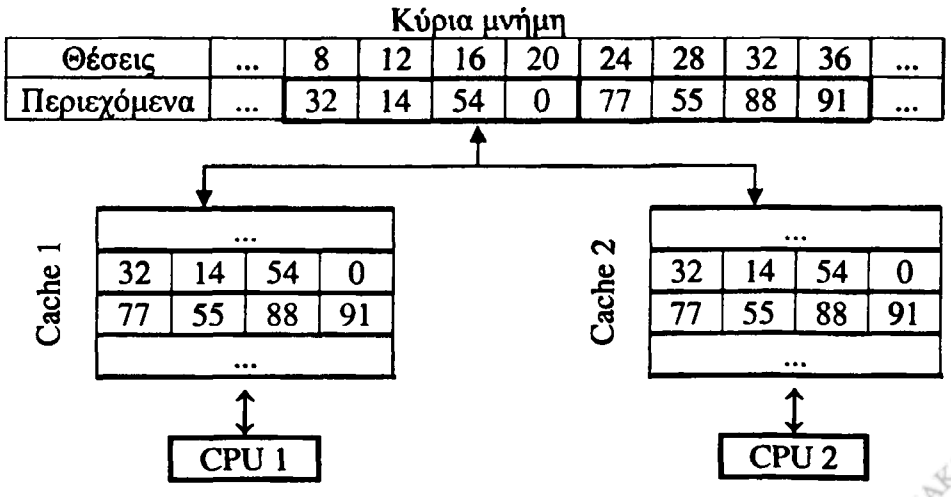
4. Ξεκλειδώνουν την κλειδαριά ατομικής πρόσβασης και συνεχίζουν με την εκτέλεση του υπόλοιπου κώδικα.

4.3 Βελτιστοποιήσεις σε σχέση με την λανθάνουσα μνήμη

Στα συστήματα κοινής μνήμης ο ρόλος της λανθάνουσας μνήμης (cache) των επεξεργαστών είναι ιδιαίτερα σημαντικός: εκτός από την συνήθη βελτίωση που επιφέρει λόγω της μεγαλύτερης ταχύτητάς της από την κύρια μνήμη, μειώνει τον ανταγωνισμό μεταξύ των επεξεργαστών για την προσπέλαση της ίδιας μνήμης αλλά και τις καθυστερήσεις λόγω του δικτύου διασύνδεσης των επεξεργαστών με την μνήμη.

Η λανθάνουσα μνήμη είναι συνήθως οργανωμένη σε γραμμές μεγέθους μερικών δεκάδων ή λίγων εκατοντάδων bytes (Σχήμα 4-2). Σε κάθε γραμμή περιέχονται ισάριθμα και συνεχόμενα bytes της μνήμης. Έτσι όταν ένας επεξεργαστής μεταβάλλει έστω και ένα από τα αντίστοιχα byte της κυρίας μνήμης, θα πρέπει να σημειωθούν ως άκυρες *ολόκληρες* οι cache lines των άλλων επεξεργαστών που αναφέρονται στην περιοχή αυτού του byte.

Έστω για παράδειγμα ότι ένας χρήστης δηλώνει έναν πίνακα ακεραίων, όπως φαίνεται στο Σχήμα 4-1. Έστω ότι το μέγεθος των ακεραίων είναι 4 byte και η κάθε γραμμή της λανθάνουσας μνήμης 16 bytes. Αν η CPU 1 θελήσει να μεταβάλλει τον ακέραιο 14 που βρίσκεται στην θέση 12, τότε η αντίστοιχη γραμμή (εφόσον υπάρχει) της Cache 2 θα θεωρηθεί άκυρη. Αν λοιπόν η CPU 2 θελήσει να διαβάσει την τιμή της θέσης 20, τότε παρόλο που αυτή δεν έχει τροποποιηθεί θα πρέπει να διαβαστεί



Σχήμα 4-1: Λαμβάνοντας υπ' όψη την λανθάνουσα μνήμη

CPU	L2			L1 (data)		
	LineSize	Way	CacheSize	LineSize	Way	CacheSize
486 (unified L1)				16B	4	8/16KB
P5 (L2 external, typical)	32B	4	256/512KB	32B	4/2	16/8KB
P6 (most/old models)	32B	4	128KB – 2MB	32B	4/2	16/8KB
UltraSparc IIe	64B		256KB			
UltraSparc IIi	64B		256KB – 2 MB	32B	2	16KB
UltraSparc IIIi	64B	4	1MB	32B	4	64KB
UltraSparc IIIcu	64 – 512B	2	1/4/8 MB		4	64KB

Σχήμα 4-2 Χαρακτηριστικά της λανθάνουσας μνήμης μερικών επεξεργαστών

από την μνήμη, μαζί με τις υπόλοιπες θέσεις που αντιστοιχούν στην ίδια cache line.

Το φαινόμενο αυτό είναι γνωστό ως *false sharing* [Schi94] και μπορεί να επιφέρει μεγάλη επιβράδυνση σε παράλληλα προγράμματα. Η λύση είναι απλά να μην τοποθετούνται κοντά στην μνήμη δεδομένα που μπορεί να μεταβάλλονται ταυτόχρονα από διαφορετικά νήματα. Όπως καταλαβαίνουμε αυτό είναι ευθύνη και του μεταφραστή αλλά και του προγραμματιστή. Έγιναν αρκετές βελτιστοποιήσεις του αρχικού κώδικα του OMPi προς αυτή την κατεύθυνση με καλά αποτελέσματα όπως θα δούμε στην παράγραφο 4.5.3. Συγκεκριμένα το false sharing επηρέαζε ιδιαίτερα την απόδοση των σημείων φραγής αλλά και των κλειδαριών γενικότερα: επειδή τα σημεία φραγής υλοποιούνται με κλειδαριές οι οποίες βρίσκονται σε διπλανές θέσεις στην μνήμη, οι αλλεπάλληλες προσπάθειες των νημάτων να κλειδώσουν τις αντίστοιχες κλειδαριές μπορεί να προκαλούν διαρκείς ανανεώσεις της λανθάνουσας μνήμης. Για να αποφύγουμε αυτό το φαινόμενο αυξήσαμε (padding) το μέγεθος των κλειδαριών ώστε να μην είναι δυνατόν να συμπέσουν δύο κλειδαριές στο ίδιο cache line. Τα νήματα POSIX χρησιμοποιούν την ίδια τεχνική για την υλοποίηση των κλειδαριών τους.

4.4 Υλοποίηση της οδηγίας Flush

Η οδηγία `flush` του OpenMP είναι, συνήθως, παραμελημένη στις υλοποιήσεις των ερευνητικών τουλάχιστον μεταφραστών. Η σωστή υλοποίησή της περιλαμβάνει δύο θέματα (βλ. και παράγραφο 2.3.3): καταρχάς θα πρέπει να ειδοποιηθεί το υλικό για την εκκένωση των buffers στην μνήμη και κατά δεύτερο θα πρέπει να ειδοποιηθεί ο μεταγλωττιστής (π.χ. gcc) που χρησιμοποιείται να μην μεταφέρει τα περιεχόμενα θέσεων μνήμης στους καταχωρητές σε αυτό το σημείο.

Η αντιμετώπιση του πρώτου θέματος εξαρτάται από την αρχιτεκτονική του συστήματος εκτέλεσης. Για παράδειγμα στα Origin 2000/3000 δεν χρειάζεται να γίνει τίποτα, αφού διαθέτουν σειριακή συνέπεια (αν και ο επεξεργαστής MIPS που χρησιμοποιούν δεν έχει). Η έκδοση V8 των Sparc υποστηρίζει δύο μοντέλα συνέπειας, το TSO (Total Store Order) και το PSO (Partial Store Order). Ανάλογα με το μοντέλο θα πρέπει να επιλέξουμε την κατάλληλη εντολή (Πίνακας 4-1), αν και μπορούμε απλά να εκτελέσουμε και τις δύο εντολές, αφού στο μοντέλο TSO η εντολή `STBAR` αντιστοιχεί σε `NOOP`. Στην έκδοση V9 των UltraSparc υποστηρίζεται ένα ακόμα μοντέλο συνέπειας, το RMO (Relaxed Memory Order). Στα UltraSparc η πιο εύκολη μέθοδος είναι η εκτέλεση της εντολής `MEMBAR #Sync`.

Σε αρχιτεκτονικές IA-32 (x86) μπορούμε να χρησιμοποιήσουμε κάποια από τις εντολές που αναφέρονται στον παρακάτω πίνακα, ανάλογα με τον τύπο του

Sparc

Εντολή	CPU	Ordering model(s)
`SWAP / LDSTUB`	V8	TSO
`STBAR`	V8	PSO
`MEMBAR #LoadStore`	V9	TSO/PSO/RMO
`MEMBAR #LoadLoad`	V9	TSO/PSO/RMO
`MEMBAR #StoreLoad`	V9	TSO/PSO/RMO
`MEMBAR #StoreStore`	V9	TSO/PSO/RMO
`MEMBAR #Sync`	V9	TSO/PSO/RMO

Origin 2000/3000

Δεν είναι αναγκαία κάποια εντολή, διαθέτει σειριακή συνέπεια μνήμης

486/Pentium

Εντολή	486?	Περιγραφή
`IRET(D)`	ναι	μπορεί να χρειάζεται προνόμια σε μερικές περιστάσεις
`RSM`	ναι	μπορεί να εκτελεστεί μόνο εντός SMM (System Management Mode)
`CPUID`	όχι	μη προνομιακή
`LGDT Ms`	όχι	προνομιακή
`LIDT Ms`	όχι	προνομιακή
`LLDT Ms`	όχι	προνομιακή
`LTR Ew`	όχι	προνομιακή
`INVLPG M`	όχι	προνομιακή, κακώς υλοποιημένη στους Intel P5 και P54
`INVD`	όχι	προνομιακή, δεν γράφει τα περιεχόμενα της write-back cache
`WBINVD`	ναι	προνομιακή
`WRMSR`	Δ/Υ	προνομιακή
`SFENCE`	Δ/Υ	μη προνομιακή, αλλά απαιτεί SSE (Streaming SIMD Extensions)
`MFENCE`	Δ/Υ	μη προνομιακή, αλλά απαιτεί SSE2
`LFENCE`	Δ/Υ	μη προνομιακή, αλλά απαιτεί SSE2

Πίνακας 4-1: Εντολές σειριοποίησης σε διάφορες αρχιτεκτονικές

επεξεργαστή. Εναλλακτικά μπορεί να χρησιμοποιηθεί η εντολή `xchg` με τον ένα τελεστή της να αναφέρεται σε κάποια θέση μνήμης. Αυτή παράγει αυτόματα ένα σήμα #LOCK κλειδώνοντας τον δίαυλο του συστήματος και περιμένει όλες τις προηγούμενες εντολές να τελειώσουν και όλες τις buffered εγγραφές να καταγραφούν στην μνήμη.

Η αντιμετώπιση του δεύτερου θέματος εξαρτάται από τον χρησιμοποιούμενο μεταγλωττιστή και μπορεί να γίνει μόνο έμμεσα σε source to source μεταφραστές όπως ο OMPi. Οι μεταγλωττιστές της Microsoft παρέχουν την οδηγία `#pragma optimize {on | off}` για τον έλεγχο του πότε θα γίνεται βελτιστοποίηση (και άρα πότε θα χρησιμοποιούνται καταχωρητές για τις μεταβλητές) αλλά αυτό δεν μπορεί να εφαρμοστεί γενικά. Μία απλοϊκή λύση θα ήταν να απαγορεύσουμε εντελώς την βελτιστοποίηση μέσω command line επιλογών, αλλά αυτό θα έκανε αργή την εκτέλεση του παραγόμενου προγράμματος. Η μόνη γενική λύση που μένει είναι να απαγορεύουμε με κάποιον έμμεσο τρόπο στον μεταγλωττιστή να διατηρεί τα περιεχόμενα μεταβλητών σε καταχωρητές κατά την διάρκεια της `flush`.

Για τον gcc τουλάχιστον αυτό γίνεται αυτόματα κατά την κλήση οποιασδήποτε συνάρτησης, *εφόσον αυτή βρίσκεται σε εξωτερική βιβλιοθήκη*, αφού ο μεταγλωττιστής δεν έχει πρόσβαση στον κώδικα της συνάρτησης για να δει ποιους καταχωρητές χρησιμοποιεί και να προσπαθήσει να «δεσμεύσει» κάποιους καταχωρητές κατά την κλήση της. Επιπλέον, επειδή η υλοποίηση της `flush` βρίσκεται στην βιβλιοθήκη libompi.a δεν είναι δυνατόν να κάνει inlining και βελτιστοποιήσεις.

4.5 Πειράματα

Στην ενότητα αυτή παρουσιάζονται τα αποτελέσματα διαφόρων πειραματικών μετρήσεων που εκτελέσαμε για την μέτρηση της επίδοσης του OMPi μετά τις προαναφερθείσες αλλαγές στον κώδικα.

4.5.1 Περιβάλλον πειραμάτων

Για την διεξαγωγή των πειραμάτων είχαμε στην διάθεσή μας τρία παράλληλα συστήματα. Ο Πίνακας 4-2 συνοψίζει τα χαρακτηριστικά τους. Η atlantis και ο zeus είναι *κοινής μνήμης* και οι επεξεργαστές τους συνδέονται μέσω διαύλου, ενώ ο helios είναι *κατανεμημένης μνήμης* αλλά μπορεί να προγραμματιστεί και με βάση το μοντέλο κοινής μνήμης, καθώς υποστηρίζει εγγενώς κατανεμημένη κοινή μνήμη.

Σύστημα	Μοντέλο	Επεξεργαστές	Μνήμη	Λειτουργικό σύστημα
helios (cc.uoi.gr)	Silicon Graphics Origin 2000	64 MIPS R12000 @250 MHz	32 Gb	IRIX64 6.5.17
atlantis (cs.uoi.gr)	Compaq ML 570	4 Xeon @700 MHz, 1Mb cache	1,5 Gb	RedHat Linux 9
zeus (cs.uoi.gr)	Sun Ultra Enterprise 3000	2 Dual Ultra Sparc 2i @336 MHz, 4 Mb cache	1 Gb	SunOS 5.9

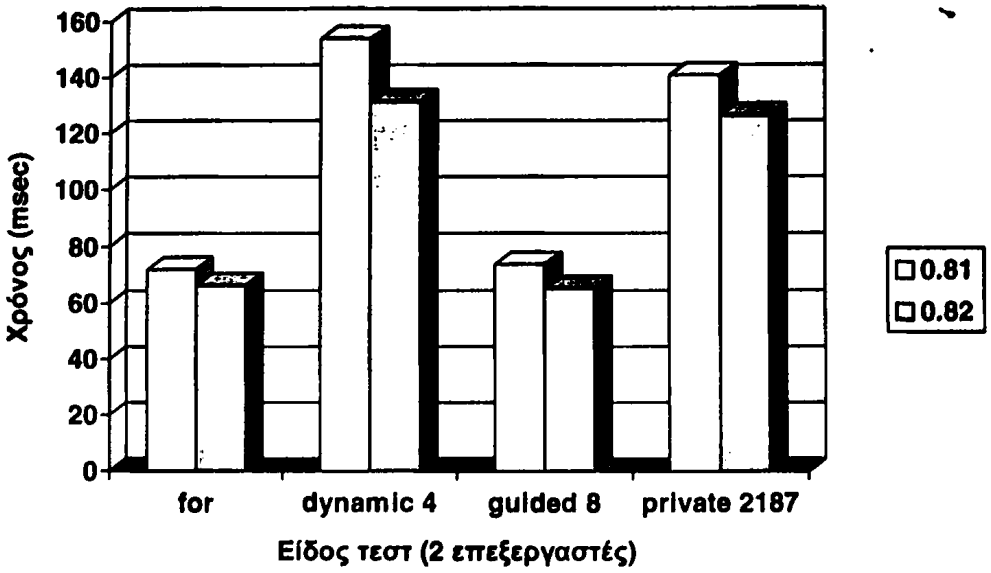
Πίνακας 4-2: Περιβάλλον πειραμάτων

4.5.2 Αφαίρεση περιττών σημείων φραγής

Για να δοκιμάσουμε την επίδοση της τεχνικής αφαίρεσης των επιπλέων barriers, εκτελέσαμε τα προγράμματα μέτρησης επιδόσεων EPCC microbenchmarks [Bull99]. Σε κάποια από τα τεστ που εκτελούνε τα παραπάνω προγράμματα υπάρχει κώδικας με την παρακάτω μορφή:

```
Έναρξη χρονομέτρησης ...
#pragma omp parallel private(j)
{
  κώδικας ...
  #pragma omp for
   for (i = 0; i < nthreads; i++) {
     κώδικας ...
   }
}
Τέλος χρονομέτρησης ...
```

Βλέπουμε ότι μέσα στην οδηγία `parallel` υπάρχει εμφωλευμένη μία `for`, η οποία τελειώνει στο ίδιο σημείο με την `parallel`. Στην οδηγία `for` δεν έχει οριστεί η φράση `nowait`. Η έκδοση 0.81 εισάγει δύο υπονοούμενα σημεία φραγής ενώ η 0.82 κρατάει μόνο ένα, αυτό της `parallel`. Τα αποτελέσματα μερικών από τα τεστ που περιλαμβάνουν κώδικα με την παραπάνω μορφή φαίνονται στο Γράφημα 4-1. Βλέπουμε ότι η αφαίρεση του περιττού σημείου φραγής επιφέρει βελτίωση στον χρόνο εκτέλεσης που φτάνει και το 15% σε κάποιες περιπτώσεις.



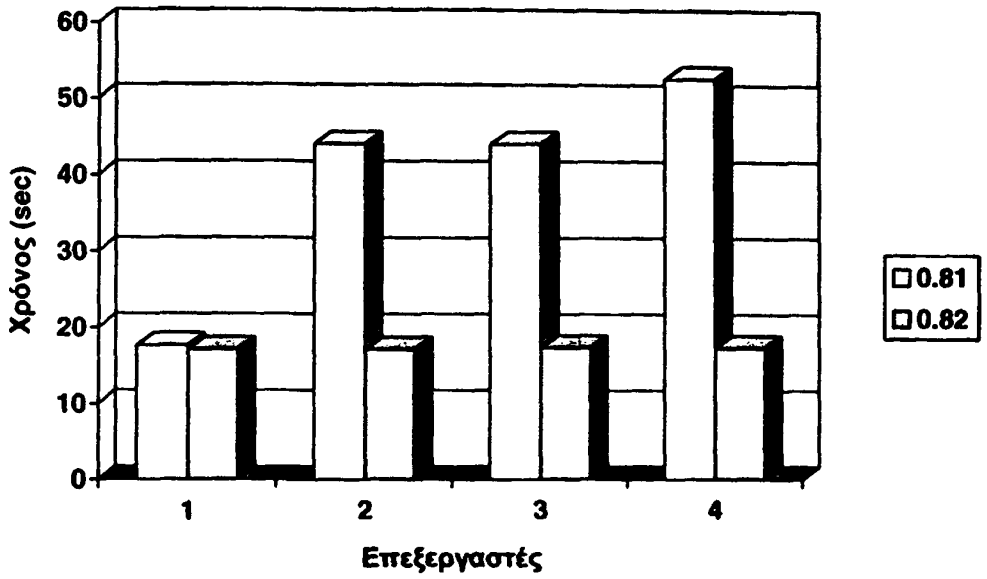
Γράφημα 4-1 Αφαίρεση περιττών σημείων φραγής στο EPCC (zeus)

4.5.3 Αποφυγή του φαινόμενου false sharing

Όπως είδαμε στην παράγραφο 4.3 το φαινόμενο του false sharing κάνει έντονη την παρουσία του όταν πολλά νήματα προσπελαύνουν ταυτόχρονα δεδομένα που βρίσκονται στην ίδια cache line. Για να επιδείξουμε καλύτερα την βελτίωση σε σχέση με το φαινόμενο αυτό επιλέξαμε τον παρακάτω κώδικα:

```
omp_lock_t *locks;
#pragma omp parallel
{
  long l, t, id;

  { Δέσμευση μνήμης και αρχικοποίηση κλειδαριών ... }
  #pragma omp for schedule(static,1)
  for (t = 0; t < omp_get_num_threads(); t++) {
    id = omp_get_thread_num();
    for (l = 0; l < 100000000; l++) {
      omp_set_lock(&locks[id]);
      global++;
      omp_unset_lock(&locks[id]);
    }
  }
  { Καταστροφή κλειδαριών ... }
}
```



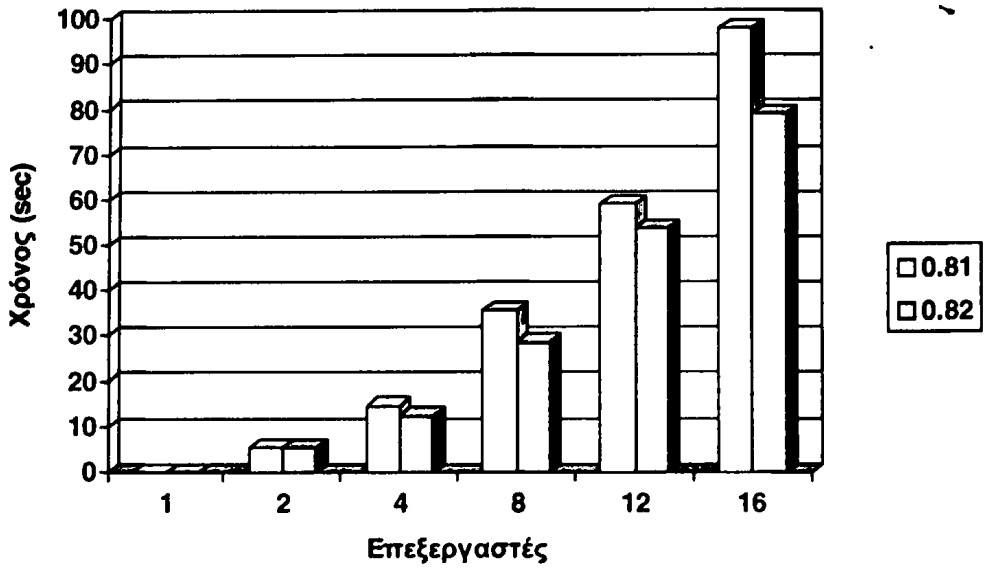
Γράφημα 4-2 Αποφυγή του φαινόμενου false sharing (atlantis)

Ο κώδικας αυτός παράγει *σταθερό φόρτο* για κάθε επεξεργαστή: η εξωτερική οδηγία `for` κάνει λίγες επαναλήψεις και ίσες κάθε φορά με τον αριθμό των νημάτων. Στην συνέχεια ο κάθε επεξεργαστής εκτελεί πολλά κλειδώματα και ξεκλειδώματα *διαφορετικών* κλειδαριών. Ιδανικά λοιπόν ο παραπάνω κώδικας χρειάζεται *σταθερό χρόνο εκτέλεσης* ανεξάρτητα από τον αριθμό των επεξεργαστών.

Το προσωρινό κλείδωμα μιας κλειδαριάς για την μεταβολή κάποιας μεταβλητής απαντάται συχνά στον κώδικα και του μεταφραστή αλλά και των χρηστών, αν και στην συγκεκριμένη περίπτωση οι κλειδαριές είναι ξεχωριστές για κάθε νήμα και δεν υπάρχει αμοιβαίος αποκλεισμός.

Το Γράφημα 4-2 δείχνει καταρχάς ότι η επίδοση της καινούργιας έκδοσης μένει σχεδόν σταθερή ανεξάρτητα από τον αριθμό των νημάτων, αφού κάθε επεξεργαστής αναλαμβάνει συγκεκριμένο φόρτο. Ο χρόνος εκτέλεσης της παλιότερης έκδοσης όμως αυξάνεται λόγω του false sharing. Φυσικά για ένα νήμα δεν υπάρχει διαφορά.

Για δύο νήματα και επειδή οι δύο κλειδαριές πέφτουν στο ίδιο cache line, το false sharing προκαλεί μεγάλη επιβράδυνση, αφού όταν το ένα νήμα μεταβάλλει την κατάσταση της κλειδαριάς του τα περιεχόμενα της cache του *άλλου* νήματος θα πρέπει να ανανεωθούν. Στα τρία νήματα δεν υπάρχει ιδιαίτερη αλλαγή, προφανώς επειδή η τρίτη κλειδαριά «έπεσε» σε διαφορετικό cache line. Στο συγκεκριμένο



Γράφημα 4-3 Βελτίωση των barriers με αποφυγή false sharing (helios)

σύστημα η cache line είναι 64 bytes, ενώ οι κλειδαριές που χρησιμοποιεί η παλιά έκδοση έχουν μέγεθος 32 bytes[1] και έτσι χωράνε μόνο δύο κλειδαριές στην ίδια γραμμή της cache. Η προσθήκη ενός ακόμη νήματος σε διαφορετικό επεξεργαστή έριξε λίγο ακόμη την επίδοση, αφού πλέον τέσσερις κλειδαριές βρίσκονται σε δύο cache lines ανά ζεύγος.

Το φαινόμενο του false sharing επηρεάζει και την επίδοση των σημείων φραγής: για την υλοποίησή τους χρειαζόμαστε μία κλειδαριά για κάθε νήμα. Αυτές οι κλειδαριές δεσμεύονται συνήθως σε κοντινές θέσεις μνήμης, είτε με ξεχωριστή `malloc` για κάθε νήμα (στην προηγούμενη έκδοση) είτε με μία συνολική `malloc` για όλα τα νήματα μαζί (στην έκδοση 0.82). Έτσι καθώς τα νήματα προσπαθούν να κλειδώσουν τις κλειδαριές τους ακυρώνουν συχνά τις cache lines των υπόλοιπων επεξεργαστών, οι οποίες μπορεί να περιέχουν κάποια κλειδαριά άλλου νήματος.

Το Γράφημα 4-3 δείχνει τα πειραματικά αποτελέσματα. Το πρόγραμμα που δοκιμάσαμε εκτελούσε απλά πολλές φορές μία οδηγία `barrier`. Η εκτέλεση έγινε 3 φορές για κάθε αριθμό νημάτων και εξαχθήκαν οι μέσοι όροι των χρόνων. Η βελτίωση που παρατηρήσαμε έφτανε σε ορισμένες περιπτώσεις το 20%.

[1] Αυτό αποτελεί απλούστευση: στο σύστημα αυτό τα pthread_mutex_t που χρησιμοποιεί η παλιά έκδοση έχουν μέγεθος 24 bytes, αλλά δημιουργούνται με malloc και έτσι κατακρατούνται ακόμα 8 bytes για την οργάνωση μνήμης του συστήματος.

Ο κώδικας που εκτελέσαμε ήταν:

```
StartTiming();
#pragma omp parallel
{
  int l;

  for (l = 0; l < 100000; l++) {
    #pragma omp barrier
  }
}
EndTiming();
ReportTiming();
```

Κεφάλαιο 5

Βιβλιοθήκη Παρακολούθησης Επίδοσης

Η βελτίωση της επίδοσης ενός παράλληλου προγράμματος είναι, συνήθως, επίπονη εργασία. Δεν αρκεί η επιλογή του βέλτιστου σειριακού αλγόριθμου. Μερικά από τα σημεία τα οποία θα πρέπει να προσέξει ο προγραμματιστής είναι να ισοκατανείμει τον φόρτο σε όλα τα νήματα, να ελαχιστοποιήσει την χρήση κλειδαριών αλλά και τα σειριακά μέρη του προγράμματος, τα οποία αποτρέπουν την επίτευξη γραμμικής επιτάχυνσης.

Από την άλλη, η παρακολούθηση της επίδοσης κατά την εκτέλεση των παράλληλων προγραμμάτων είναι δύσκολο να γίνει από τον χρήστη. Για παράδειγμα, σε έναν παράλληλο βρόχο `for` ενδιαφερόμαστε για τον φόρτο προετοιμασίας της οδηγίας, τους χρόνους κατανομής των επαναλήψεων και της εκτέλεσής τους από τα αντίστοιχα νήματα, και τον χρόνο αναμονής του κάθε νήματος στο σημείο φραγής κατά το τέλος της `for`. Η μελέτη αυτών των χρόνων μπορεί να βοηθήσει τους χρήστες να παραλληλοποιήσουν αποδοτικότερα το πρόγραμμά τους αλλά και τους ίδιους τους προγραμματιστές των μεταγλωττιστών OpenMP να εντοπίσουν υποβέλτιστα σημεία στον παραγόμενο κώδικα. Αυτό το κενό έρχονται να καλύψουν τα πρότυπα παρακολούθησης της επίδοσης, όπως είναι το PMPI [MiGe97] για το MPI και τα POMP [MMSW01] και OMPI [MMHS02] για το OpenMP.

Μέχρι σήμερα το Architecture Review Board (ARB) του OpenMP δεν έχει υιοθετήσει επίσημα κάποιο πρωτόκολλο παρακολούθησης επίδοσης (profiling). Έχει όμως έμπρακτα εκδηλώσει ενδιαφέρον με τον σχηματισμό της υπο-επιτροπής "Tools", η οποία θα είναι υπεύθυνη για την ενσωμάτωση ενός τέτοιου προτύπου στο

OpenMP. Το πρώτο υποψήφιο είναι το OMPI "performance instrumentation interface" το οποίο αναπτύχθηκε από την Pallas GmbH στα πλαίσια του έργου IST INTONE και υποστηρίζεται από το σύστημα μεταγλώττισης INTONE για την γλώσσα Fortran και από τον μεταγλωττιστή INTONE C ο οποίος αναπτύχθηκε από την KTH στην Σουηδία. Το δεύτερο υποψήφιο είναι το POMP "performance tool interface" που σχεδιάστηκε και υλοποιήθηκε από το Forschungszentrum Julich, το Πανεπιστήμιο του Oregon και το εργαστήριο λογισμικού KAI (KSL) της Intel. Επειδή το ARB μπορεί να υιοθετήσει μόνο ένα πρωτόκολλο παρακολούθησης, οι παραπάνω οργανισμοί έχουν κάνει κάποιες προσπάθειες για την ενοποίηση των προτεινόμενων πρωτοκόλλων, χωρίς όμως να καταλήξουν ακόμα σε μια οριστική πρόταση. Επιλέξαμε να υλοποιήσουμε το πρωτόκολλο POMP αφ' ενός επειδή η διεπαφή της βιβλιοθήκης που προτείνει είναι ταχύτερη, πιο δομημένη και πιο κοντά στο προτεινόμενο ενιαίο πρότυπο, και αφετέρου επειδή υποστηρίζεται από ανοιχτού κώδικα εργαλεία γραφικής αναπαράστασης των αποτελεσμάτων, όπως το πρόγραμμα ανάλυσης της επίδοσης EXPERT [MoWo02] και το εργαλείο παρουσίασης CUBE [SoWo04], σε αντιδιαστολή με το Vampir που χρησιμοποιείται από το OMPI και είναι πλέον εμπορικό.

5.1 Μετασχηματισμοί του POMP

Η βασική ιδέα του POMP είναι ο μετασχηματισμός του πηγαίου κώδικα ενός προγράμματος OpenMP ώστε να συμπεριληφθούν κλήσεις στην βιβλιοθήκη παρακολούθησης επίδοσης `pomplib`. Ο στόχος είναι να χρονομετρηθούν ο επιπλέον φόρτος που εισάγεται από κάθε οδηγία λόγω της παραλληλοποίησης, ο καθαρός χρόνος εργασίας για κάθε νήμα καθώς και οι χρόνοι αναμονής σε κλειδαριές και στα σημεία φραγής. Έτσι εισάγονται κλήσεις πριν την είσοδο και μετά την έξοδο από κάθε οδηγία OpenMP, ενώ σε οδηγίες με μεγάλο φόρτο όπως η `parallel` εισάγονται κλήσεις και ακριβώς *μετά την είσοδο*, ώστε να μετρηθεί η επιβάρυνση εξαιτίας της συγκεκριμένης οδηγίας. Επιπλέον χρονομετρούνται όλα τα σημεία φραγής, είτε ρητώς δηλωμένα είτε υπονοούμενα (κάτι που αντιστοιχεί σε πολλές χρονομετρήσεις, μία για κάθε νήμα), όπως επίσης και οι κλήσεις στις πιθανώς χρονοβόρες συναρτήσεις χειρισμού κλειδαριών που ορίζονται στην βιβλιοθήκη omp.h.

Ο προτεινόμενος (και υλοποιημένος) πίνακας μετασχηματισμών είναι:

Οδηγία OpenMP	Μετασχηματισμός POMP

`#pragma omp parallel`
`  structured-block`
	`POMP_Parallel_fork(d);`
	`#pragma omp parallel`
	`{`
	`  POMP_Parallel_begin(d);`
	`  structured-block`
	`  POMP_Barrier_enter(d);`
	`  #pragma omp barrier`
	`  POMP_Barrier_exit(d);`
	`  POMP_Parallel_end(d);`
	`}`
	`POMP_Parallel_join(d);`

`#pragma omp for`
`  for-loop`
	`POMP_For_enter(d);`
	`#pragma omp for nowait`
	`  for-loop`
	`POMP_Barrier_enter(d);`
	`#pragma omp barrier`
	`POMP_Barrier_exit(d);`
	`POMP_For_exit(d);`

`#pragma omp sections`
`{`
`  #pragma omp section`
`  structured-block`
`  #pragma omp section`
`  structured-block`
`}`
	`POMP_Sections_enter(d);`
	`#pragma omp sections nowait`
	`{`
	`  #pragma omp section`
	`  {`
	`  POMP_Section_begin(d);`
	`  structured-block`
	`  POMP_Section_end(d);`
	`  }`
	`  #pragma omp section`
	`  {`
	`  POMP_Section_begin(d);`
	`  structured-block`
	`  POMP_Section_end(d);`
	`  }`
	`}`
	`POMP_Barrier_enter(d);`
	`#pragma omp barrier`
	`POMP_Barrier_exit(d);`
	`POMP_Sections_exit(d);`

#pragma omp barrier	POMP_Barrier_enter(d); **#pragma omp barrier** POMP_Barrier_exit(d);

#pragma omp critical
structured-block
	POMP_Critical_enter(d);
	#pragma omp critical
	{
	POMP_Critical_begin(d);
	structured-block
	POMP_Critical_end(d);
	}
	POMP_Critical_exit(d);

#pragma omp single
structured-block
	POMP_Single_enter(d);
	#pragma omp single nowait
	{
	POMP_Single_begin(d);
	structured-block
	POMP_Single_end(d);
	}
	POMP_Barrier_enter(d);
	#pragma omp barrier
	POMP_Barrier_exit(d);
	POMP_Single_exit(d);

#pragma omp atomic *atomic-expression*	POMP_Atomic_enter(d); **#pragma omp atomic** *atomic-expression* POMP_Atomic_exit(d);

#pragma omp master
structured-block
	#pragma omp master
	{
	POMP_Master_begin(d);
	structured-block
	POMP_Master_end(d);
	}

Και για τις "μικτές" παράλληλες οδηγίες οι μετασχηματισμοί είναι:	
#pragma omp parallel *for clauses ...* *for-loop*	POMP_Parallel_fork(d); **#pragma omp parallel** *other-clauses ...* { POMP_Parallel_begin(d); POMP_For_enter(d); **#pragma omp for nowait** *schedule-clauses, ordered-clauses, lastprivate-*

	clauses
	  for-loop
	  POMP_Barrier_enter(d);
	  #pragma omp barrier
	  POMP_Barrier_exit(d);
	  POMP_For_exit(d);
	  POMP_Parallel_end(d);
	}
	POMP_Parallel_join(d);

#omp parallel *section clauses ...*
#omp section
  structured-block
	POMP_Parallel_fork(d);
	#pragma omp parallel *other-clauses ...*
	{
	  POMP_Parallel_begin(d);
	  POMP_Sections_enter(d);
	  #pragma omp section *lastprivate-clauses*
	  #pragma omp section nowait
	  {
	  POMP_Section_begin(d);
	  structured-block
	  POMP_Section_end(d);
	  }
	  POMP_Barrier_enter(d);
	  #pragma omp barrier
	  POMP_Barrier_exit(d);
	  POMP_Parallel_end(d);
	}
	POMP_Parallel_join(d);

Πίνακας 5-1: Μετασχηματισμοί των οδηγιών OpenMP κατά το POMP

Η κύρια διαφορά της υλοποίησης από την προτεινόμενη μέθοδο αφορά στα barriers. Οι δημιουργοί του POMP ανέπτυξαν το εργαλείο OPARI για την εκτέλεση των παραπάνω source to source μετασχηματισμών. Στη συνέχεια η έξοδος του OPARI πρέπει να περάσει από έναν OpenMP compiler. Έτσι δεν είχαν την δυνατότητα να επέμβουν στον "κώδικα OpenMP", δηλαδή να απομονώσουν και να χρονομετρήσουν για παράδειγμα το υπονοούμενο φράγμα της `#pragma omp for`. Αναγκαστικά λοιπόν προτείνουν να αφαιρούνται όλα τα `nowait` και να μπαίνει ένα επιπλέον ρητό barrier ώστε να μπορούν να το χρονομετρήσουν. Εμείς όμως μέσω του parser.y έχουμε πρόσβαση στα "ενδότερα" του OpenMP, και επομένως αποφύγαμε αυτούς τους περιορισμούς.

Μία ακόμα διαφορά στην υλοποίηση είναι στις συνδυασμένες παράλληλες οδηγίες (parallel for/sections), όπου είχαμε την δυνατότητα απευθείας υλοποίησης των μετασχηματισμών χωρίς να χρειάζεται διάσπαση σε δύο ξεχωριστές οδηγίες.

5.2 Παρακολούθηση run-time Βιβλιοθήκης

Το πρότυπο POMP ορίζει ότι οι κλήσεις στις παρακάτω συναρτήσεις θα πρέπει επίσης να παρακολουθούνται, ώστε να αποκαλύπτονται τυχόν χρονοβόρα κλειδώματα. Αυτό υλοποιείται απλά με την προσθήκη σχετικών οδηγιών #define στην αρχή του παραγόμενου κώδικα.

Συνάρτηση της <omp.h>	Αντίστοιχη συνάρτηση της <pomplib.h>
`void omp_init_lock( omp_lock_t *s);`	`void POMP_Init_lock( omp_lock_t *s);`
`void omp_destroy_lock( omp_lock_t *s);`	`void POMP_Destroy_lock( omp_lock_t *s);`
`void omp_set_lock( omp_lock_t *s);`	`void POMP_Set_lock( omp_lock_t *s);`
`void omp_unset_lock( omp_lock_t *s);`	`void POMP_Unset_lock( omp_lock_t *s);`
`int omp_test_lock( omp_lock_t *s);`	`int POMP_Test_lock( omp_lock_t *s);`
`void omp_init_nest_lock( omp_nest_lock_t *s);`	`void POMP_Init_nest_lock( omp_nest_lock_t *s);`
`void omp_destroy_nest_lock( omp_nest_lock_t *s);`	`void POMP_Destroy_nest_lock( omp_nest_lock_t *s);`
`void omp_set_nest_lock( omp_nest_lock_t *s);`	`void POMP_Set_nest_lock( omp_nest_lock_t *s);`
`void omp_unset_nest_lock( omp_nest_lock_t *s);`	`void POMP_Unset_nest_lock( omp_nest_lock_t *s);`
`int omp_test_nest_lock( omp_nest_lock_t *s);`	`int POMP_Test_nest_lock( omp_nest_lock_t *s);`

Πίνακας 5-2: Αντικατάσταση συναρτήσεων κλειδαριών κατά το POMP

5.3 Παρακολούθηση κώδικα χρήστη

Το POMP αναφέρει ότι ο χρήστης μπορεί να καθορίσει δικές του περιοχές κώδικα προς παρακολούθηση, μέσω των παρακάτω οδηγιών:

Οδηγία χρήστη	Αντίστοιχες συναρτήσεις της <pomplib.h>

`#pragma omp inst begin(region_name)`
`  arbitrary user code`
`#pragma omp inst end(region_name)`
	`POMP_Begin(d);`
	`  arbitrary user code`
	`POMP_End(d);`

Αυτό χρησιμεύει στον εντοπισμό σημείων συμφόρησης του πηγαίου κώδικα. Επιπλέον, το πρότυπο προτείνει ότι οι μεταφραστές που υλοποιούν το POMP θα πρέπει να ενορχηστρώνουν αυτόματα όλες τις οριζόμενες από τον χρήστη συναρτήσεις με τις αντίστοιχες συναρτήσεις `POMP_Begin(d)` και `POMP_End(d)`. Δείτε την παράγραφο 5.5 για περισσότερες λεπτομέρειες σχετικά με την αυτόματη ενορχήστρωση των συναρτήσεων.

5.4 Ενεργοποίηση και απενεργοποίηση του POMP

Η παρακολούθηση της επίδοσης δεν είναι χρήσιμη σε όλα τα μέρη ενός προγράμματος. Για μεγάλα προγράμματα, εκτός από την πτώση στην επίδοση παράγεται και καταιγισμός πληροφοριών, κάτι που δυσκολεύει τον προγραμματιστή στον εντοπισμό των σημείων που χρειάζονται βελτιστοποίηση. Γι' αυτόν τον λόγο το πρότυπο POMP προτείνει τρεις τρόπους για την επιμέρους ή την καθολική απενεργοποίησή του.

5.4.1 Οδηγίες απενεργοποίησης της παρακολούθησης

Κάθε συνάρτηση της βιβλιοθήκης pomplib, όταν καλείται, καταγράφει τα δεδομένα παρακολούθησης της επίδοσης σε ένα αρχείο με επέκταση .elg. Αυτό φυσικά επιφέρει πτώση της επίδοσης και αύξηση της καταγραφόμενης πληροφορίας. Ο προγραμματιστής μπορεί να ειδοποιήσει την βιβλιοθήκη να σταματήσει προσωρινά την καταγραφή δεδομένων στο αρχείο χρησιμοποιώντας τις παρακάτω οδηγίες:

Οδηγία	Αντίστοιχη συνάρτηση της <pomplib.h>
`#pragma omp inst on`	`void POMP_On();`
`#pragma omp inst off`	`void POMP_Off();`

Για την έναρξη και τον τερματισμό της παρακολούθησης καθορίζονται οι εξής οδηγίες:

Οδηγία	Αντίστοιχη συνάρτηση της <pomplib.h>
`#pragma omp inst init`	`void POMP_Init();`
`#pragma omp inst finalize`	`void POMP_Finalize();`

Όμως δεν απαιτείται από τον προγραμματιστή η χρησιμοποίησή τους, αφού παράγονται αυτόματα από τον μεταφραστή.

5.4.2 Οδηγίες απενεργοποίησης του POMP

Σε μερικά σημεία ο προγραμματιστής μπορεί να χρειαστεί όχι απλά να απενεργοποιήσει την καταγραφή δεδομένων που κάνει η pomplib, αλλά να σταματήσει τελείως την παραγωγή POMP κώδικα. Αυτό μπορεί να γίνει με τις παρακάτω οδηγίες:

Οδηγία	Περιγραφή
`#pragma omp instrument`	Ο μεταφραστής παράγει κώδικα POMP
`#pragma omp noinstrument`	Ο μεταφραστής δεν παράγει κώδικα POMP

Υπάρχουν όμως ορισμένα σημεία που πρέπει να προσεχθούν από την πλευρά του μεταφραστή κατά την υλοποίηση αυτών των οδηγιών, δείτε την παράγραφο 5.5 για περισσότερες λεπτομέρειες.

5.4.3 Παράμετροι απενεργοποίησης του POMP

Η παραγωγή κώδικα POMP μπορεί να προσαρμοστεί και από την γραμμή εντολών, μέσω της παράμετρου `--pomp-disable=...` του μεταφραστή. Οι τιμές της παραμέτρου μπορεί να είναι `atomic`, `barrier`, `critical`, `for`, `master`, `parallel`, `section`, `sections` ή `single` για την απενεργοποίηση της παρακολούθησης των αντίστοιχων οδηγιών, `region` για τις περιοχές που καθορίζονται από τον χρήστη και `sync` για την καθολική απενεργοποίηση παραγωγής κώδικα POMP.

Στον OMPi διαφοροποιηθήκαμε λίγο από το πρότυπο για χρηστικούς λόγους. Επειδή σε συνήθη χρήση η παραγωγή κώδικα παρακολούθησης *δεν χρειάζεται*, αποφασίσαμε από προεπιλογής το POMP να είναι *απενεργοποιημένο* και για την ενεργοποίησή του να δίνεται η παράμετρος `--pomp-enable`. Για παράδειγμα για να ενεργοποιήσει κάποιος το POMP και συγχρόνως να απενεργοποιήσει την παρακολούθηση των περιοχών `atomic` θα πρέπει να δώσει

```
ompicc filename.c --pomp-enable --pomp-disable=atomic
```

5.5 Υλοποίηση του POMP

Για την υποστήριξη των παραπάνω μετασχηματισμών χρειάστηκε να προστεθούν τα κατάλληλα tokens στον λεκτικό αναλυτή και ο αντίστοιχος χειρισμός και οι προσθήκες κώδικα στον συντακτικό αναλυτή. Ένα σημείο που χρίζει σχολιασμού είναι η κατασκευή του παραγόμενου κώδικα σε σημεία ενεργοποίησης / απενεργοποίησης του POMP. Ένα παράδειγμα:

```
#pragma omp parallel
{
  statements...
  #pragma omp inst off
  #pragma atomic
    atomic expression
}
```

Φυσικά αφού η οδηγία `#pragma omp inst off` απενεργοποιεί την ενορχήστρωση, δεν θα παραχθεί κώδικας παρακολούθησης της οδηγίας `#pragma atomic`. Το πρόβλημα παρουσιάζεται κατά το κλείσιμο της `#pragma omp parallel`, όπου η ενορχήστρωση είναι απενεργοποιημένη και δεν θα έπρεπε να παραχθεί κώδικας παρακολούθησης. Έτσι όμως θα έμενε «ανοιχτή» η `#pragma omp parallel`, αφού παρακολουθείται η αρχή της αλλά όχι το τέλος της.

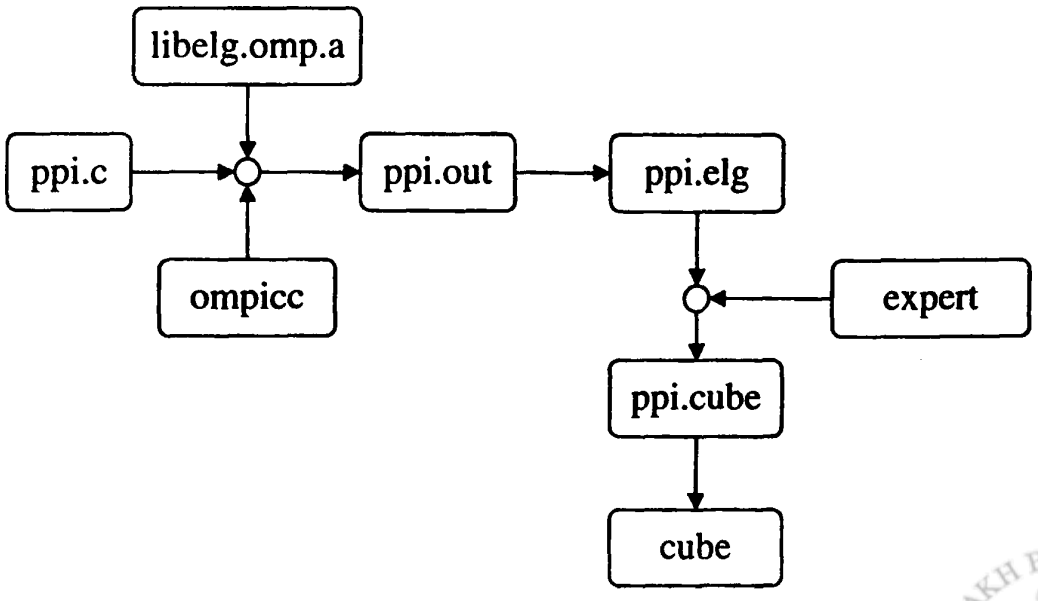
Μία λύση στο πρόβλημα είναι να κρατείται στοίβα με όλες τις οδηγίες, και αν η ενορχήστρωση ήταν ενεργή κατά το άνοιγμα κάποιας τότε να ενορχηστρώνεται και το τέλος της ακόμα και αν τότε είναι ανενεργή. Μαζί με την στοίβα βέβαια πρέπει να φυλάσσεται και το «βάθος» κάθε οδηγίας για να αποφευχθεί το εξής πρόβλημα: αν έχουμε δύο παράλληλες οδηγίες, και η ενορχήστρωση είναι ενεργή κατά το άνοιγμα της πρώτης και κλειστή κατά την δεύτερη, τότε πρέπει να ξέρουμε σε ποια οδηγία

αναφέρεται ποιο «τέλος» της κάθε παράλληλης περιοχής. Η υλοποίηση πλέον λαμβάνει υπόψη της όλες αυτές τις περιπτώσεις.

5.6 Παράδειγμα παρακολούθησης με το POMP

Καταρχάς να επισημάνουμε ότι η *υλοποίηση* της βιβλιοθήκης παρακολούθησης δεν είναι μέρος του OMPi, αφού αυτό παράγει απλά τις *κλήσεις* προς την βιβλιοθήκη. Η `libpomp.a` που διατίθεται με το OMPi εμφανίζει απλά κάποια μηνύματα στην οθόνη, δεν καταγράφει τις πληροφορίες σε αρχείο. Για να μπορέσει λοιπόν κάποιος να παρακολουθήσει με γραφικό τρόπο την επίδοση των προγραμμάτων του θα πρέπει (βλ. και Σχήμα 5-1):

1. Να προμηθευτεί και να μεταγλωττίσει την βιβλιοθήκη χρόνου εκτέλεσης EPILOG (Event Processing, Investigating and LOGging). Η βιβλιοθήκη αυτή παρέχεται μαζί με τον πηγαίο κώδικα του εργαλείου KOJAK από τους δημιουργούς του POMP. Το αντικειμενικό αρχείο `libelg.omp.a` θα πρέπει να μετονομαστεί και να αντικαταστήσει την βιβλιοθήκη `libpomp.a` του OMPi. Έτσι κατά την εκτέλεση των προγραμμάτων OpenMP θα δημιουργείται αυτόματα ένα αρχείο *filename.elg*.-με τις πληροφορίες εκτέλεσης του προγράμματος.
2. Στη συνέχεια θα πρέπει να γίνει μετατροπή του τύπου του αρχείου με τις πληροφορίες εκτέλεσης από *.elg* σε *.cube*, ώστε να μπορεί να εμφανιστεί με το γραφικό περιβάλλον απεικόνισης. Η μετατροπή γίνεται με το εργαλείο EXPERT το οποίο επίσης διατίθεται μαζί με το KOJAK.



Σχήμα 5-1: Βήματα από την μεταγλώττιση ενός προγράμματος (ppi.c) μέχρι την γραφική αναπαράσταση της εκτέλεσής του

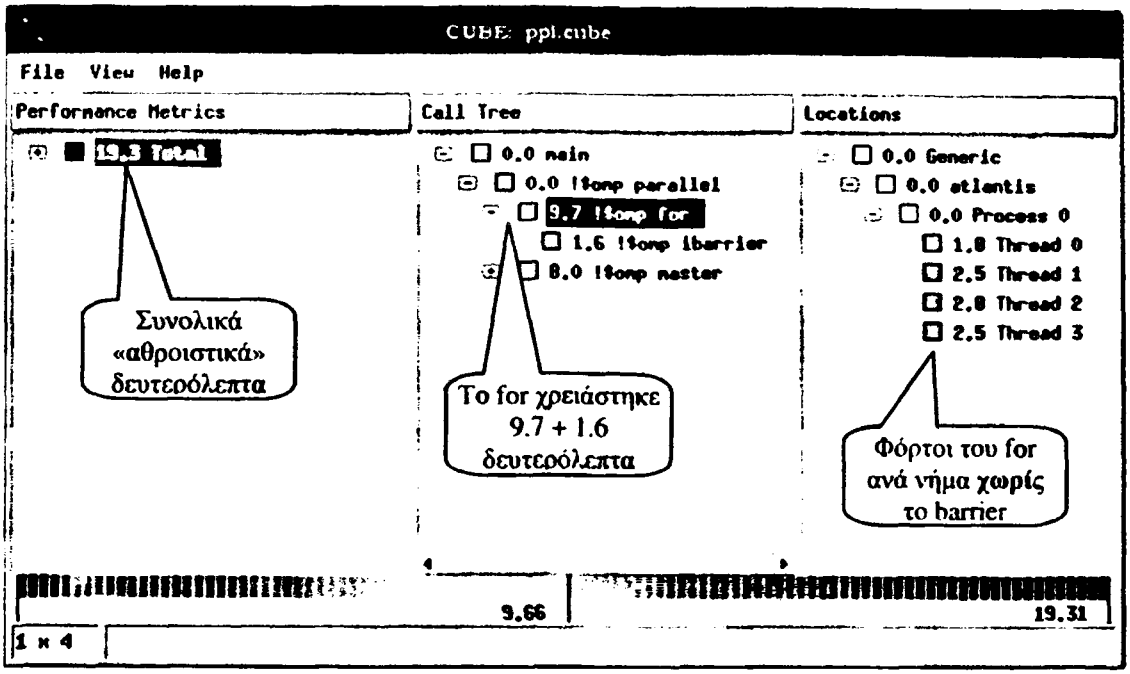
3. Τέλος θα πρέπει να εγκατασταθεί, να μεταγλωττιστεί και να εκτελεστεί το γραφικό περιβάλλον απεικόνισης CUBE (CUBE Uniform Behavioral Encoding, [SoWo04]), το οποίο είναι ένα γενικό εργαλείο παρουσίασης μετρικών επίδοσης για παράλληλα προγράμματα που υποστηρίζει MPI [MPI94] και OpenMP εφαρμογές. Η εγκατάστασή του προϋποθέτει την ύπαρξη των πακέτων wxWidgets και libxml2.

Σαν παράδειγμα profiling επιλέξαμε μία εφαρμογή υπολογισμού του αριθμού π (Πίνακας 5-3). Για να επιδείξουμε την δυνατότητα παρακολούθησης περιοχών ορισμένων από τον χρήστη αλλά και τα ανενεργά (idle) νήματα προσθέσαμε στον κώδικα την περιοχή `sleeping`, κατά την οποία μόνο το νήμα αφέντης θεωρείται ενεργό (για 2 δευτερόλεπτα).

Το πρόγραμμα εκτελέστηκε με τέσσερα νήματα σε ισάριθμους επεξεργαστές. Όπως παρατηρούμε στην Εικόνα 5-1 τα νήματα εργαζόταν για 19.3 αθροιστικά δευτερόλεπτα, στα οποία προσμετράται και ο ανενεργός χρόνος. Φυσικά ο πραγματικός χρόνος εκτέλεσης είναι ο μέγιστος από τα νήματα (4.8 δευτερόλεπτα), τον οποίο θα μπορούσαμε να παρακολουθήσουμε από την περιοχή Locations της ίδιας εικόνας αν επιλέγαμε την συνάρτηση `main` στην περιοχή Call Tree. Γενικά δηλαδή επιλέγοντας κάποιον κόμβο βλέπουμε τον συνολικό του φόρτο όταν είναι «κλειστός» και τον αποκλειστικό του φόρτο (χωρίς τους υποκόμβους) όταν είναι «ανοιχτός». Διαλέγοντας λοιπόν το `!$omp for` από το Call Tree βλέπουμε στο

```
#pragma omp parallel
 {
   #pragma omp for reduction(+: pi) private(local)
   for (i = 0; i < N; i++) {
     local = (i + 0.5)*w;
     pi += 4.0/(1.0 + local*local);
   }
   #pragma omp master
   {
     printf("pi is %1.20lf\n", pi*w);
     #pragma omp inst begin(sleeping)
     sleep(2);
     #pragma omp inst end(sleeping)
   }
 }
```

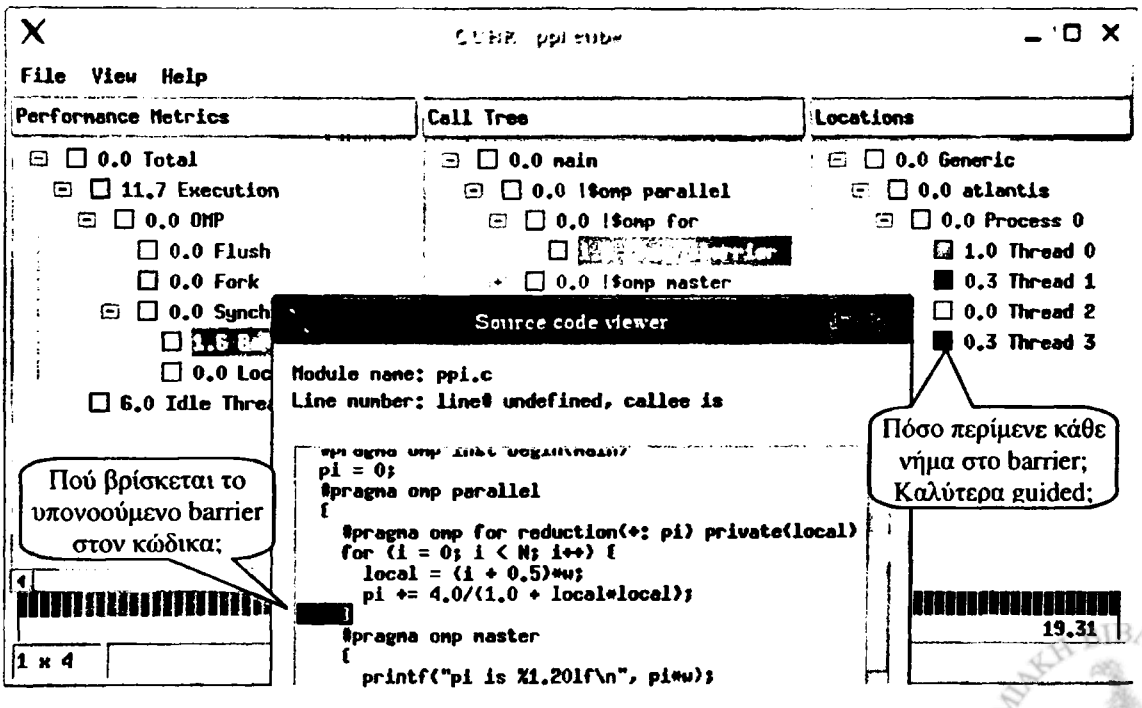
Πίνακας 5-3: Πρόγραμμα υπολογισμού του π



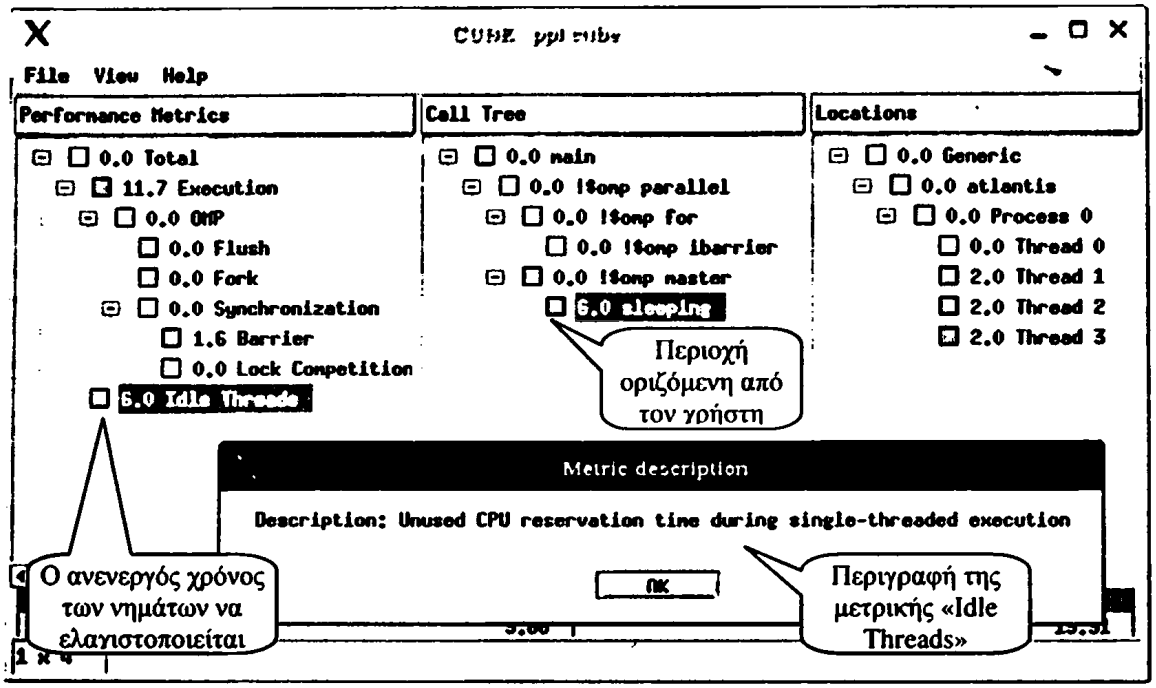
Εικόνα 5-1: Παρακολούθηση μικτών και καθαρών χρόνων

Locations την πραγματική εργασία υπολογισμού του π που έκανε το καθένα από τα νήματα.

Βλέπουμε επίσης ότι το νήμα 0 (αφέντης) ανέλαβε μικρότερο μερίδιο εργασίας (1.8 δευτερόλεπτα), επομένως θα πρέπει να περίμενε περισσότερο στο υπονοούμενο barrier. Πράγματι, αν επιλέξουμε το barrier στο Performance Metrics και στο Call Tree, φαίνεται στην Εικόνα 5-2 ότι το νήμα αφέντης περίμενε 1 ολόκληρο δευτερόλεπτο στο barrier. Ο χρήστης θα πρέπει να σκεφτεί ότι ίσως να πετύχαινε



Εικόνα 5-2: Φόρτος της υπονοουμένης barrier και θέση στον πηγαίο κώδικα



Εικόνα 5-3: Ανενεργά νήματα και περιοχές οριζόμενες από τον χρήστη

καλύτερη επίδοση αν πρόσθετε την φράση `guided` στην οδηγία `for`, ώστε να γίνει καλύτερος καταμερισμός φόρτου.

Τέλος, μπορούμε να παρατηρήσουμε τον χρόνο που τα νήματα ήταν ανενεργά, επιλέγοντας Idle Threads από το Performance Metrics (Εικόνα 5-3). Στο Call Tree βλέπουμε την οριζόμενη από την χρήστη περιοχή `sleeping`, και στο Locations ότι μόνο το νήμα αφέντης εργαζόταν κατά την περιοχή sleeping, ενώ τα υπόλοιπα απλά περίμεναν για 2 δευτερόλεπτα.

Κεφάλαιο 6

Συμπεράσματα

Στην εργασία αυτή ασχοληθήκαμε με θέματα υλοποίησης ενός μεταφραστή της γλώσσας C, ο οποίος υποστηρίζει τις προδιαγραφές OpenMP για παράλληλό προγραμματισμό κοινής μνήμης.

Σε σημαντικό ποσοστό η αρχική υλοποίηση του OMPi δεν διέθετε λεπτομερή σχολιασμό/τεκμηρίωση. Το πρώτο μέλημά μας λοιπόν ήταν ο σχολιασμός του κώδικα με βάση το στυλ που χρησιμοποιεί το πρόγραμμα doxygen για την παραγωγή αυτόματης τεκμηρίωσης. Στη συνέχεια μελετήθηκαν διάφορα πακέτα νημάτων όπως τα POSIX και Solaris threads και απομονώθηκε η προγραμματιστική διεπαφή χειρισμού νημάτων που χρησιμοποιούσε ο μεταφραστής στην βιβλιοθήκη `othread.h`, ενώ έγινε και υλοποίησή της για τα προαναφερθέντα πακέτα. Ακόμη, διερευνήθηκαν τα δύο γνωστότερα πρωτόκολλα παρακολούθησης της επίδοσης προγραμμάτων που έχουν προταθεί για το OpenMP και έγινε παραγωγή κώδικα για ένα από αυτά, το POMP.

Στα πλαίσια της εργασίας έγιναν και αρκετές αλγοριθμικές βελτιώσεις, όπως η προσθήκη λογικής για ανίχνευση και αφαίρεση των περιττών σημείων φραγής, η κατασκευή αλγορίθμου για την οδηγία `ordered` χωρίς την χρήση condition variables, η χρησιμοποίηση των εμφωλευμένων κλειδαριών που προσφέρουν τα pthreads και η δήλωση των συναρτήσεων χειρισμού των barriers ως weak symbols για να μπορεί εύκολα ο χρήστης να δοκιμάζει δικούς του αλγορίθμους. Τέλος, έγιναν διάφορες βελτιστοποιήσεις ανάλογα με την αρχιτεκτονική του συστήματος, όπως για

παράδειγμα η υλοποίηση της οδηγίας `flush` και της συνάρτησης `omp_get_wtime`, ενώ τροποποιήθηκε ο κώδικας ώστε να λαμβάνει υπόψη του την λανθάνουσα μνήμη του συστήματος.

Όλες οι παραπάνω αλλαγές ήταν ευεργετικές για τις επιδόσεις των παραγόμενων προγραμμάτων, αυξάνοντας την ταχύτητα εκτέλεσης μέχρι και 20% σε ορισμένες περιπτώσεις.

6.1 Μελλοντικές κατευθύνσεις

Μία σημαντική έλλειψη του OMPi είναι η ύπαρξη ενδιάμεσης αναπαράστασης του κώδικα *μετά* την συντακτική ανάλυση. Αυτή τη στιγμή το αρχείο parser.y περιέχει περίπου 5000 γραμμές κώδικα οι οποίες ταυτόχρονα κάνουν και την συντακτική ανάλυση αλλά και την μετατροπή του κώδικα σε ισοδύναμο που χρησιμοποιεί νήματα. Έτσι οι διορθώσεις, τροποποιήσεις και οι προσθήκες νέων λειτουργιών είναι εξαιρετικά επίπονη, ενώ απαιτεί γνώσεις Bison/Flex. Θα ήταν πιο εύκολο σε κάποιον να έρθει σε επαφή και να βελτιώσει το OMPi αν το parser.y είχε σαν αποκλειστική ευθύνη την μετατροπή του κώδικα σε συντακτικά δέντρα και ένα ξεχωριστό αρχείο κώδικα αναλάμβανε την παραλληλοποίηση. Εξάλλου η τρέχουσα λογική του συντακτικού αναλυτή δεν επιτρέπει μεγάλες αναλύσεις και βελτιστοποιήσεις, αφού ο κώδικας παράγεται απευθείας με το πρώτο πέρασμα χωρίς να δίνεται η δυνατότητα πρόβλεψης ή μεταβολής σημείων πού έχουν ήδη περάσει.

Ακόμη, θα πρέπει να υλοποιηθούν και ορισμένα προαιρετικά χαρακτηριστικά του OpenMP, όπως εμφωλευμένος παραλληλισμός, δυναμική προσαρμογή του αριθμού των νημάτων που εκτελούν τις παράλληλες περιοχές και κάποια ελλιπή σημεία όπως η υποστήριξη εξωτερικών `threadprivate` μεταβλητών. Επίσης καλό θα ήταν να διερευνηθεί ένας αποδοτικότερος τρόπος για την προσπέλαση σε *κοινόχρηστες τοπικές μεταβλητές προηγούμενων επιπέδων*, που αυτή τη στιγμή γίνεται μέσω δεικτών.

Μία εναλλακτική προοπτική είναι να ξεφύγουμε τελείως από τους περιορισμούς των source to source μεταφραστών και να στραφούμε στην μετατροπή μεταγλωττιστών ανοιχτού κώδικα, και συγκεκριμένα του gcc. Ο gcc αναπαριστά τον κώδικα σε ενδιάμεσο επίπεδο (μετά την συντακτική ανάλυση) με τα gcc-trees, και σε τελευταίες εκδόσεις με τα ssa-trees. Οι δυνατότητες αναδόμησης του κώδικα και οδηγιών προς τον assembler που προσφέρονται σε αυτό το επίπεδο είναι κατά πολύ μεγαλύτερες οποιασδήποτε μετάφρασης πηγαίου κώδικα, όσο αποδοτική κι αν είναι.

Τέλος, ένα θέμα που θα πρέπει να διερευνηθεί αφορά στην δυνατότητα προγραμματισμού clusters υπολογιστών με βάση το μοντέλο κοινής μνήμης, μέσω ενός επιπέδου λογισμικού το οποίο θα εξομοιώνει την κοινή μνήμη (software DSM). Έχουν ήδη γίνει κάποιες προσπάθειες εκτέλεσης προγραμμάτων OpenMP σε clusters [HLCZ00] χρησιμοποιώντας ενδιάμεσο λογισμικό το οποίο υποστηρίζει διάφορα μοντέλα συνοχής. Τα συστήματα αυτά είναι κυρίως page-based και βασίζονται στα σφάλματα σελίδας του υλικού για την υλοποίηση της κατανεμημένης κοινής μνήμης, αν και έχουν προταθεί και αντικειμενοστραφής προσεγγίσεις οι οποίες έχουν σαν βάση τις μεταβλητές και όχι τις σελίδες της μνήμης, και επομένως δεν απαιτούν την ανάλογη υποστήριξη του υλικού.

Βιβλιογραφία

[BrBr00] C. Brunschen and M. Brorsson, "OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency": *Practice and Experience*, vol. 12, pp 1193-1203, 2000.

[Bull99] J. M. Bull, "Measuring Synchronization and Scheduling Overheads in OpenMP", *Proc. EWOMP'99*: First European Workshop on OpenMP, Lund, Sweden, Sept. 1999

[CuSG98] D. E. Culler, J. P. Singh and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, Aug. 1998

[Dima01] Β. Δημακόπουλος, *Παράλληλη Επεξεργασία*, Πανεπιστήμιο Ιωαννίνων, Φεβ. 2001

[DLTz03] V. Dimakopoulos, E. Leontiadis and G. Tzoumas, "A portable C compiler for OpenMP V2.0", *Proc. EWOMP'03: European workshop for OpenMP*, pp 4-11, Aachen, Germany, Sept. 2003

[Fost95] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995

[HLCZ00] Y.C. Hu, H. Lu, A.L. Cox and W. Zwaenepoel, "OpenMP for Networks of SMPs", *Journal of Parallel and Distributed Computing*, vol. 60 (12), pp 1512-1530, Dec. 2000

[HPCC94] *High Performance Computing and Communications*: "Toward a National Information Infrastructure", the Federal Coordinating Council for Science, Engineering, and Technology, 1994

[IEEE00] IEEE Std. 1003.1j-2000, *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Programming Interface (API) – Amendment j: Advanced Realtime Extension [C Language]*, IEEE Standards Press, 2000

[IEEE93] IEEE Std. 1003.1b-1993, *Information Technology - Portable Operating System Interface (POSIX) – Part 1: System Application Programming Interface (API) – Amendment 1: Realtime Extension [C Language]*, IEEE Standards Press, 1993

[IEEE96] IEEE/ANSI Std 1003.1 1996 Edition, *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language] (ANSI)*, IEEE Standards Press, 1996

[LeTz02] Η. Λεοντιάδης και Γ. Τζούμας, Πτυχιακή εργασία *Υλοποίηση μεταφραστή C με επεκτάσεις OpenMP για παραλληλοποίηση*, Πανεπιστήμιο Ιωαννίνων, Ιούλιος 2002

[MiGe97] S. Mintchev and V. Getov, "PMPI: High-Level Message Passing in Fortran77 and C", *High--Performance Computing and Networking (HPCN'97)*, pp 603-614, Vienna, 1997

[MLNA96] X. Martorell, J. Labarta, N. Navarro and E. Ayguade, "A Library Implementation of the Nano-Threads Programming Model", *Proc. of the Second International Euro-Par Conference*, vol. 2, Lyon, France, Aug. 1996

[MMHS02] B. Mohr, A. Mallony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger and S. Shah, "A Performance Monitoring Interface for OpenMP", *4th European Workshop on OpenMP (EWOMP'02)*, Rome (Italy), Sep. 2002.

[MMSW01] B. Mohr, A. D. Malony, S. Shende and F. Wolf, "Design and Prototype of a Performance Tool Interface for OpenMP", *Proceedings of the LACSI Symposium*, 2001

[MoWo02] B. Mohr and F. Wolf, "Automatic Performance Analysis of Hybrid OpenMP/MPI Programs with EXPERT", *PADC'02*, Dagstuhl, Aug. 2002

[MPI94] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, University of Tennessee, Knoxville, Report No. CS-94-230, May 5, 1994

[Mull01] M. Müller, "Some Simple OpenMP Optimization Techniques", *Proc. of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, pp 31-39, 2001

[OARB02] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface, Version 2.0*, http://www.openmp.org, Mar. 2002

[OARB98] OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interface, Version 1.0*, http://www.openmp.org, Oct. 1998

[Schi94] C. Schimmel, *UNIX systems for modern architectures: symmetric multiprocessing and caching for kernel programmers*, Addison-Wesley, 1994

[Shah03] S. Shah, Invited talk: "OpenMP: What next?", *Proc. EWOMP'03: European workshop for OpenMP*, Aachen, Germany, Sep. 2003

[SoWo04] F. Song and F. Wolf, *CUBE User Manual*, ICL Technical Report, ICL-UT-04-01, Feb. 2004.

[SSKY99] M. Sato, S. Satoh, K. Kusano and Y.o Tanaka "Design of OpenMP compiler for a SMP cluster", *In The 1st European Workshop on OpenMP (EWOMP'99)*, pp 32-39, Sep. 1999

[SUN02] Sun Microsystems, Inc., *Multithreaded Programming Guide*, Prentice#Hall, Englewood Cliffs, NJ, 2002

[Sund90] V. S. Sunderam, *PVM:* "A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience*, 2, 4, pp 315-339, Dec. 1990

Παράρτημα Α

Οδηγίες εγκατάστασης και χρήσης

Εγκατάσταση και προσαρμογή του OMPi

Για την εγκατάσταση του OMPi το πρώτο βήμα είναι η λήψη της τελευταίας έκδοσης (ompi-0.8.2.tar.gz) από την ιστοσελίδα http://www.cs.uoi.gr/~ompi. Η εγκατάστασή του είναι αρκετά απλή και μπορεί να γίνει με τις παρακάτω κλασσικές εντολές:

1. gunzip ompi-0.8.2.tar.gz (αποσυμπίεση του αρχείου ompi-0.8.2.tar)
2. tar xvf ompi-0.8.2.tar (εξαγωγή των περιεχομένων του ompi-0.8.2.tar)
3. cd ompi-0.8.2
4. ./configure (ρύθμιση ανάλογα με το σύστημα)
5. make (μεταγλώττιση)
6. make install (εγκατάσταση)

Αν θέλουμε να εγκαταστήσουμε το OMPi σε κάποιον κατάλογο διαφορετικό του προεπιλεγμένου τότε θα πρέπει να αλλάξουμε το βήμα 4:

4. ./configure --prefix=<κατάλογος εγκατάστασης>

Αν προτιμάμε να χρησιμοποιήσουμε τα Solaris threads αντί των προεπιλεγμένων POSIX threads θα πρέπει να δώσουμε την παράμετρο

4. ./configure --enable-solaristhreads

Μία άλλη επιλογή του configure είναι η

4. ./configure --enable-debug

η οποία εμφανίζει διάφορα εσωτερικά μηνύματα του OMPi κατά την μεταγλώττιση προγραμμάτων.

Θα πρέπει βέβαια να προσέξουμε ώστε ο κατάλογος εγκατάστασης του OMPi να βρίσκεται στο PATH του συστήματος.

Εγκατάσταση των εργαλείων profiling

Η βιβλιοθήκη παρακολούθησης της απόδοσης δεν είναι μέρος του OMPi, αφού αυτό απλά παράγει τις *κλήσεις* προς την βιβλιοθήκη. Για την εγκατάστασή της θα πρέπει καταρχάς να γίνει λήψη του εργαλείου KOJAK 2.x από την διεύθυνση http://www.fz-juelich.de/zam/kojak. Η μεταγλώττιση του EXPERT, του βοηθήματος μετατροπής των δεδομένων παρακολούθησης σε κατάλληλο format, απαιτεί το ακόλουθο λογισμικό:

- wxWidgets (http://www.wxwidgets.org)
- libxml2 (http://www.xmlsoft.org)

Αφού γίνουν οι απαραίτητες ρυθμίσεις, μεταγλωττίσεις και εγκαταστάσεις θα δημιουργηθεί η βιβλιοθήκη libelg.omp.a. Το αρχείο αυτό θα πρέπει να αντιγραφεί πάνω από το libpomp.a του OMPi. Η βιβλιοθήκη libpomp.a που παρέχεται μαζί με το OMPi απλά εμφανίζει διαγνωστικά μηνύματα, ενώ η libelg.omp.a καταγράφει τις πληροφορίες στο αρχείο <όνομα εκτελέσιμου.elg>.

Για την γραφική αναπαράσταση της επίδοσης της εκτέλεσης των προγραμμάτων (profiling) είναι απαραίτητο το λογισμικό CUBE, το οποίο μπορεί να ληφθεί από την ιστοσελίδα http://icl.cs.utk.edu/kojak/cube.

Εκτέλεση του OMPi

Το OMPi εκτελείται με την εντολή

```
ompicc [ompi parameters] [compiler parameters] [filenames]
```

Χρησιμοποιείται ο προεπιλεγμένος μεταγλωττιστής του συστήματος, επομένως οι `[compiler parameters]` εξαρτώνται από αυτόν. Οι παράμετροι που υποστηρίζει το OMPi είναι:

- `-k`: διατήρηση των ενδιάμεσων αρχείων που δημιουργούνται κατά την κατασκευή του αντίστοιχου πολυνηματικού προγράμματος
- `--enable-pomp`: ενεργοποίηση του profiling. Από προεπιλογής είναι ανενεργό επειδή επιβραδύνει την εκτέλεση των προγραμμάτων
- `--disable-pomp=atomic | barrier | critical | for | master | parallel | region | section | sections | single | sync`

Απενεργοποιεί την παρακολούθηση κάποιων από τις οδηγίες του OpenMP, σύμφωνα με τις προδιαγραφές του POMP (παράγραφος 5.4).

Εκτέλεση των εργαλείων profiling

Αν κάποιο πρόγραμμα μεταγλωττιστεί με την παράμετρο `--pomp-enable`, κατά την εκτέλεσή του θα δημιουργηθεί ένα αρχείο <όνομα εκτελέσιμου.elg> με τα δεδομένα του profiling. Στην συνέχεια θα πρέπει να εκτελεστούν οι παρακάτω εντολές:

- `expert filename.elg`: μετατροπή του αρχείου σε .cube format
- `cube filename.cube`: άνοιγμα του γραφικού εργαλείου profiling cube

Ανάπτυξη του OMPi

Ο προγραμματισμός του OMPi μπορεί να διευκολυνθεί με την χρήση του ολοκληρωμένου περιβάλλοντος ανάπτυξης kdevelop (http://www.kdevelop.org). Μαζί με τον πηγαίο κώδικα του OMPi παρέχονται τα ανάλογα project αρχεία OMPi.kdevelop και OMPi.kdevelop.filelist. Η συγγραφή κώδικα, η μεταγλώττιση, η εγκατάσταση αλλά και η δημιουργία του διανεμόμενου συμπιεσμένου αρχείου μπορούν να γίνουν μέσα από το kdevelop.

Ο σχολιασμός του κώδικα γίνεται έτσι ώστε να μπορούν αυτόματα να δημιουργηθούν αρχεία τεκμηρίωσης του OMPi με το πρόγραμμα doxygen (http://www.dogygen.org).

Παράρτημα Β

Πηγαίος κώδικας

ompi-0.8.2/configure.in

```
00001 dnl
00002 dnl     OMPi OpenMP Compiler
00003 dnl     Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas
00004 dnl     Alkis Georgopoulos
00005 dnl
00006 dnl     This file is part of OMPi.
00007 dnl
00008 dnl     OMPi is free software; you can redistribute it and/or modify
00009 dnl     it under the terms of the GNU General Public License as published by
00010 dnl     the Free Software Foundation; either version 2 of the License, or
00011 dnl     (at your option) any later version.
00012 dnl
00013 dnl     OMPi is distributed in the hope that it will be useful,
00014 dnl     but WITHOUT ANY WARRANTY; without even the implied warranty of
00015 dnl     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016 dnl     GNU General Public License for more details.
00017 dnl
00018 dnl     You should have received a copy of the GNU General Public License
00019 dnl     along with OMPi; if not, write to the Free Software
00020 dnl     Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 dnl
00022 dnl     This file must be processed by autoconf
00023
00024 dnl     Configure.in file for OMPi
00025
00026 dnl     AC_PREREQ(2.5)
00027 AC_INIT(ompi/ompicc.c)
00028 AM_CONFIG_HEADER(ompi/config.h)
00029
00030 PACKAGE="ompi"
00031 VERSION="0.8.2"
00032 AM_INIT_AUTOMAKE($PACKAGE,$VERSION)
00033
00034 dnl Standard stuff
00035 AC_PROG_CC
00036 AC_LANG_C
00037 AC_ISC_POSIX
00038 AC_PROG_MAKE_SET
00039 AC_PROG_RANLIB
00040 AC_PROG_YACC
00041 AM_PROG_LEX
00042
00043 dnl Check for functions (will generate approrpiate #defines)
00044 AC_HEADER_STDC
00045
00046 dnl Give a debugging option
00047 AC_ARG_ENABLE(debug, [  --enable-debug          turn on debugging (default is no)],
00048               if eval "test x$enable_debug = xyes"; then
00049                 DEBUGFLAG="-g"
00050               fi, DEBUGFLAG="")
00051
00052 dnl Give an option to select Solaris threads
00053 AC_ARG_ENABLE(
00054   [solaristhreads],
00055   [  --enable-solaristhreads use Solaris threads instead of pthreads (default is no)],
00056   [ case "${enableval}" in
00057       yes) solaristhreads=true
00058            DEFINETHREADLIB="-DSOLARISTHREADS" ;;
00059       no)  solaristhreads=false
00060            DEFINETHREADLIB="" ;;
00061       *)   AC_MSG_ERROR([bad value ${enableval} for --enable-solaristhreads]) ;;
00062   esac],
00063   [solaristhreads=false
00064    DEFINETHREADLIB=""])
00065 AM_CONDITIONAL(SOLARISTHREADS, test x$solaristhreads = xtrue)
00066
00067 dnl The following variables should be substituted in Makefile.in
00068 AC_SUBST(VERSION)
00069 AC_SUBST(PACKAGE)
00070 AC_SUBST(DEBUGFLAG)
00071 AC_SUBST(DEFINETHREADLIB)
00072
00073 #AM_CONDITIONAL(OS_IRIX, test x`uname` = xIRIX64)
```

```
00074
00075 dnl Finally, output the makefiles
00076 AC_OUTPUT(Makefile lib/Makefile ompi/Makefile pomp/Makefile)
```

ompi-0.8.2/Makefile.am

```
00001 #
00002 #    OMPi OpenMP Compiler
00003 #    Copyright 2001-2003 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004 #    Alkis Georgopoulos
00005 #
00006 #    This file is part of OMPi.
00007 #
00008 #    OMPi is free software; you can redistribute it and/or modify
00009 #    it under the terms of the GNU General Public License as published by
00010 #    the Free Software Foundation; either version 2 of the License, or
00011 #    (at your option) any later version.
00012 #
00013 #    OMPi is distributed in the hope that it will be useful,
00014 #    but WITHOUT ANY WARRANTY; without even the implied warranty of
00015 #    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016 #    GNU General Public License for more details.
00017 #
00018 #    You should have received a copy of the GNU General Public License
00019 #    along with OMPi; if not, write to the Free Software
00020 #    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 #
00022 ##   This file must be processed by automake
00023
00024 AUTOMAKE_OPTIONS = gnu
00025 SUBDIRS = ompi lib pomp
00026 EXTRA_DIST = BUGS OMPi.kdevelop OMPi.kdevelop.filelist Doxyfile
```

ompi-0.8.2/lib/generic_othread.c

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*! \file generic_othread.c
00024     \brief The wrapper function implementations for POSIX threads.
00025  */
00026
00027 #include <othread.h>
00028 #include <pthread.h>
00029
00030 /** Create Thread.
00031  The othread_create() function creates a thread with the specified attributes and runs the C
00032  function start_routine in the thread with the single pointer argument specified. The new thread
00033  may, but does not always, begin running before othread_create() returns. If othread_create()
00034  completes successfully, the Othread handle is stored in the contents of the location referred to
00035  by thread. Returns 0 if successful.
00036 */
00037 int othread_create(
00038  othread_t *thread,                              /**< (Out) Othread handle to the created thread */
00039  const othread_attr_t *attr,     /**< (In) The thread attributes object containing the attributes to
```

```
00040     be associated with the newly created thread. If NULL, the default thread attributes are used. */
00041   void *(*start_routine)(void *),/**< (In) The function to be run as the new threads start routine */
00042   void *arg)                     /**< (In) An address for the argument for the threads start routine */
00043 {
00044   return pthread_create(thread, attr, start_routine, arg);
00045 }
00046
00047 /** Compare Two Threads.
00048  The othread_equal() function compares two Othread handles for equality.
00049  Returns 1 if the Othread handles refer to the same thread, 0 otherwise.
00050 */
00051 int othread_equal(
00052  othread_t t1,                                          /**< (In) Othread handle for thread 1 */
00053  othread_t t2)                                          /**< (In) Othread handle for thread 2 */
00054 {
00055   return pthread_equal(t1, t2);
00056 }
00057
00058 /** Get Thread Local Storage Value by Key.
00059  The othread_getspecific() function retrieves the thread local storage value associated with the
00060  key. othread_getspecific() may be called from a data destructor. The thread local storage value is
00061  a variable of type void * that is local to a thread, but global to all of the functions called
00062  within that thread. It is accessed via the key. Returns NULL if key out of range.
00063 */
00064 void *othread_getspecific(
00065  othread_key_t key)     /**< (In) The thread local storage key returned from othread_key_create() */
00066 {
00067   return pthread_getspecific(key);
00068 }
00069
00070 /** Create Thread Local Storage Key.
00071  The othread_key_create() function creates a thread local storage key for the process and associates
00072  the destructor function with that key. After a key is created, that key can be used to set and get
00073  per-thread data pointer. When othread_key_create() completes, the value associated with the newly
00074  created key is NULL. Returns 0 if successful.
00075 */
00076 int othread_key_create(
00077  othread_t *key,/**<(Out) The address of the variable to contain the thread local storage key */
00078  void (*destructor)(void *))     /**< (In) The address of the function to act as a destructor for */
00079 {                                                       /* this thread local storage key */
00080   return pthread_key_create(key, destructor);
00081 }
00082
00083 /** Get othread Handle.
00084  The othread_self() function returns the Othread handle of the calling thread. The othread_self()
00085  function does NOT return the integral thread of the calling thread. You must use
00086  othread_getthreadid_np() to return an integral identifier for the thread.
00087 */
00088 othread_t othread_self(void)
00089 {
00090   return pthread_self();
00091 }
00092
00093 /** Set Process Concurrency Level.
00094  The othread_setconcurrency() function sets the current concurrency level for the process.
00095  A concurrency value of zero indicates that the threads implementation chooses the concurrency
00096  level that best suits the application. A concurrency level greater than zero indicates that the
00097  application wants to inform the system of its desired concurrency level.
00098  Returns 0 if successful.
00099 */
00100 int othread_setconcurrency(
00101  int concurrency)                          /**< (In) The new concurrency level for the process */
00102 {
00103   return pthread_setconcurrency(concurrency);
00104 }
00105
00106 /** Set Thread Local Storage by Key.
00107  The othread_setspecific() function sets the thread local storage value associated with a key. The
00108  othread_setspecific() function may be called from within a data destructor.
00109 */
00110 int othread_setspecific(
00111  othread_key_t key,     /**< (In) The thread local storage key returned from othread_key_create(). */
00112  const void *value)     /**< (In) The pointer to store at the key location for the calling thread. */
00113 {
00114   return pthread_setspecific(key, value);
00115 }
00116
```

```
00117 /* S I M P L E   L O C K S */
00118
00119 /**
00120  The othread_init_lock function initializes the simple lock associated with the parameter lock for
00121  use in subsequent calls.
00122  */
00123 int othread_init_lock(
00124  othread_lock_t *lock)            /**< (In) The address of the variable to contain a lock object. */
00125 {
00126   return pthread_mutex_init(&lock->mutex, NULL);
00127 }
00128
00129 /**
00130  The othread_destroy_lock function ensures that the pointed to lock variable lock is uninitialized.
00131  */
00132 int othread_destroy_lock(
00133  othread_lock_t *lock)                              /**< (In) Address of the lock to be destroyed */
00134 {
00135   return pthread_mutex_destroy(&lock->mutex);
00136 }
00137
00138 /**
00139  The othread_set_lock function blocks the thread executing the function until the specified lock is
00140  available and then sets the lock. A simple lock is available if it is unlocked.
00141  */
00142 int othread_set_lock(
00143  othread_lock_t *lock)                          /**< (In) The address of the lock to set */
00144 {
00145   return pthread_mutex_lock(&lock->mutex);
00146 }
00147
00148 /**
00149  The othread_unset_lock function provide the means of releasing ownership of a simple lock.
00150  */
00151 int othread_unset_lock(
00152  othread_lock_t *lock)                            /**< (In) Address of the lock to unset */
00153 {
00154   return pthread_mutex_unlock(&lock->mutex);
00155 }
00156
00157 /**
00158  The othread_test_lock function attempts to set a lock but does not block execution of the thread.
00159  Returns a nonzero value if the lock is Successfully set; otherwise, it returns zero.
00160  */
00161 int othread_test_lock(
00162  othread_lock_t *lock)                             /**< (In) Address of the lock to set */
00163 {
00164   return pthread_mutex_trylock(&lock->mutex);
00165 }
00166
00167 /* N E S T E D   L O C K S */
00168 #ifndef PTHREAD_MUTEX_RECURSIVE
00169 #define PTHREAD_MUTEX_RECURSIVE PTHREAD_MUTEX_RECURSIVE_NP                    /* Linux */
00170 #endif
00171
00172 /**
00173  The othread_init_nest_lock function initializes the nested lock associated with the parameter lock
00174  for use in subsequent calls.
00175  */
00176 int othread_init_nest_lock(
00177  othread_nest_lock_t *lock)        /**< (In) The address of the variable to contain a nested lock */
00178 {
00179   static pthread_mutexattr_t recursive_attr;
00180   static int firstCall = 1;
00181
00182   if (firstCall) {
00183     firstCall = 0;
00184     pthread_mutexattr_init(&recursive_attr);            /* TODO: check results and halt on error */
00185     pthread_mutexattr_settype(&recursive_attr, PTHREAD_MUTEX_RECURSIVE);
00186   }
00187   return pthread_mutex_init(&lock->mutex, &recursive_attr);
00188 }
00189
00190 /**
00191  The othread_destroy_nest_lock function ensures that the pointed to lock variable lock is
00192  uninitialized.
00193  */
```

```
00194 int othread_destroy_nest_lock(
00195  othread_nest_lock_t *lock)                          /**< (In) Address of the lock to be destroyed */
00196 {
00197   return pthread_mutex_destroy(&lock->mutex);
00198 }
00199
00200 /**
00201  The othread_set_nest_lock function blocks the thread executing the function until the specified
00202  lock is available and then sets the lock. A nestable lock is available if it is unlocked or if it
00203  is already owned by the thread executing the function.
00204  */
00205 int othread_set_nest_lock(
00206  othread_nest_lock_t *lock)                          /**< (In) The address of the lock to set */
00207 {
00208   return pthread_mutex_lock(&lock->mutex);
00209 }
00210
00211 /**
00212  The othread_unset_nest_lock function provide the means of releasing ownership of a nestable lock.
00213  */
00214 int othread_unset_nest_lock(
00215  othread_nest_lock_t *lock)                          /**< (In) Address of the lock to unset */
00216 {
00217   return pthread_mutex_unlock(&lock->mutex);
00218 }
00219
00220 /**
00221  The othread_test_nest_lock function attempts to set a lock but does not block execution of the
00222  thread. Returns the new nesting count if the lock is successfully set; otherwise, it returns zero.
00223  */
00224 int othread_test_nest_lock(
00225  othread_nest_lock_t *lock)                          /**< (In) Address of the lock to set */
00226 {
00227   return pthread_mutex_trylock(&lock->mutex);
00228 }
00229
00230 /* S P I N   L O C K S */
00231
00232 /**
00233  The othread_init_spin_lock function initializes the spin lock associated with the parameter lock
00234  for use in subsequent calls.
00235  */
00236 int othread_init_spin_lock(
00237  othread_spin_lock_t *lock)          /**< (In) The address of the variable to contain a spin lock */
00238 {
00239 #ifdef PTHREAD_MUTEX_SPINBLOCK_NP                                          /* e.g. IRIX */
00240   #warning "OMPi INFO: Using PTHREAD_MUTEX_SPINBLOCK_NP for spin locks..."
00241   static pthread_mutexattr_t spinblock_attr;
00242   static int firstCall = 1;
00243
00244   if (firstCall) {
00245     firstCall = 0;
00246     pthread_mutexattr_init(&spinblock_attr);          /* TODO: check results and halt on error */
00247     pthread_mutexattr_settype(&spinblock_attr, PTHREAD_MUTEX_SPINBLOCK_NP);
00248   }
00249   return pthread_mutex_init(&lock->mutex, &spinblock_attr);
00250 #else
00251   return pthread_mutex_init(&lock->mutex, NULL);
00252 #endif
00253 }
00254
00255 /**
00256  The othread_destroy_spin_lock function ensures that the pointed to lock variable lock is
00257  uninitialized.
00258  */
00259 int othread_destroy_spin_lock(
00260  othread_spin_lock_t *lock)                          /**< (In) Address of the lock to be destroyed */
00261 {
00262   return pthread_mutex_destroy(&lock->mutex);
00263 }
00264
00265 /**
00266  The othread_set_spin_lock function blocks the thread executing the function until the specified
00267  lock is available and then sets the lock. A spin lock is available if it is unlocked.
00268  */
00269 int othread_set_spin_lock(
00270  othread_spin_lock_t *lock)                          /**< (In) The address of the lock to set */
```

```
00271 {
00272   return pthread_mutex_lock(&lock->mutex);
00273 }
00274
00275 /**
00276  The othread_unset_spin_lock function provide the means of releasing ownership of a spin lock.
00277  */
00278 int othread_unset_spin_lock(
00279  othread_spin_lock_t *lock)                          /**< (In) Address of the lock to unset */
00280 {
00281   return pthread_mutex_unlock(&lock->mutex);
00282 }
00283
00284 /**
00285  The othread_test_spin_lock function attempts to set a lock but does not block execution of the
00286  thread. Returns a nonzero value if the lock is successfully set; otherwise, it returns zero.
00287  */
00288 int othread_test_spin_lock(
00289  othread_spin_lock_t *lock)                           /**< (In) Address of the lock to set */
00290 {
00291   return pthread_mutex_trylock(&lock->mutex);
00292 }
```

ompi-0.8.2/lib/generic_othread_types.h

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*! \file generic_othread_types.h
00024     \brief Thread types. Implementation for the generic pthread library.
00025  */
00026
00027 #ifndef __GENERIC__OTHREAD_TYPES_H__
00028 #define __GENERIC__OTHREAD_TYPES_H__
00029
00030 #include <pthread.h>
00031
00032 #define OTHREAD_LOCK_INITIALIZER {PTHREAD_MUTEX_INITIALIZER}
00033
00034 typedef pthread_t othread_t;
00035 typedef pthread_attr_t othread_attr_t;
00036 typedef pthread_key_t othread_key_t;
00037
00038 /** othread_mutex_t is equivelant to pthread_mutex_t, except for its size. It is declared as a
00039  64-byte union to ensure that TWO locks can't fit in the same cache line. If two or more locks
00040  happen to be in the same cache line, then polling gets their value from RAM instead of cache,
00041  causing high bus traffic.
00042  In Solaris threads, mutex_t is already padded to 64 bytes, so there is no size loss.
00043  DEVELOPERS NOTE: unions are initialized based on their FIRST declared member, i.e. mutex.
00044  In order for the PTHREAD_MUTEX_INITIALIZER to work, the order of members should not change.
00045  Nevertheless, in Solaris threads, padding is declared first, because no THREAD_MUTEX_INITIALIZER
00046  is defined, and initialization is (according to the manuals) done by zero-filled memory.
00047  */
00048 typedef union {
00049   pthread_mutex_t mutex;
00050   char padding[64];                                  /**< 64 bytes is >= than most cache line sizes */
```

```
00051 } othread_lock_t;
00052
00053 /**
00054  The othread_nest_lock_t is an internal type, the public type omp_nest_lock_t is declared in omp.h.
00055  A nested lock is an object type capable of represending either that a lock is available, or both
00056  the identity of the thread that owns the lock and a nesting count.
00057  */
00058 typedef othread_lock_t othread_nest_lock_t;
00059
00060 /**
00061  othread_spin_lock_t locks are faster than othread_lock_t but they keep the processor busy.
00062  They are used in cases where something must be locked for a very small period of time.
00063  (For now they are implemented as simple othread_locks, except for IRIX)
00064  */
00065 typedef othread_lock_t othread_spin_lock_t;
00066
00067 #endif
```

ompi-0.8.2/lib/Makefile.am

```
00001 #
00002 #    OMPi OpenMP Compiler
00003 #    Copyright 2001-2003 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas
00004 #
00005 #    This file is part of OMPi.
00006 #
00007 #    OMPi is free software; you can redistribute it and/or modify
00008 #    it under the terms of the GNU General Public License as published by
00009 #    the Free Software Foundation; either version 2 of the License, or
00010 #    (at your option) any later version.
00011 #
00012 #    OMPi is distributed in the hope that it will be useful,
00013 #    but WITHOUT ANY WARRANTY; without even the implied warranty of
00014 #    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00015 #    GNU General Public License for more details.
00016 #
00017 #    You should have received a copy of the GNU General Public License
00018 #    along with OMPi; if not, write to the Free Software
00019 #    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00020 #
00021 ##   This file must be processed by automake
00022
00023 lib_LIBRARIES = libompi.a
00024 AM_CFLAGS = @DEBUGFLAG@ @DEFINETHREADLIB@
00025
00026 if SOLARISTHREADS
00027 libompi_a_SOURCES = ompi.c omp.c sparc_othread.c
00028 EXTRA_libompi_a_SOURCES = generic_othread.c generic_othread_types.h
00029 include_HEADERS = omp.h ompi.h othread.h sparc_othread_types.h
00030 else
00031 libompi_a_SOURCES = ompi.c omp.c generic_othread.c
00032 EXTRA_libompi_a_SOURCES = sparc_othread.c sparc_othread_types.h
00033 include_HEADERS = omp.h ompi.h othread.h generic_othread_types.h
00034 endif
```

ompi-0.8.2/lib/omp.c

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
```

```
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022 /*! \file omp.c
00023     \brief The openmp specification functions implementations.
00024  */
00025
00026 #include <omp.h>
00027 #include <ompi.h>
00028 #include <stdlib.h>
00029 #include <time.h>                                   /* for clock_gettime, clock_getres */
00030 #include <othread.h>
00031
00032 /**
00033  Returns true if we are in a parallel section. _OMP_THREAD is another macro that returns the
00034  current thread identifier (pointer).
00035  */
00036 #define _OMP_IN_PARALLEL (_OMP_THREAD != _omp_master_thread)
00037
00038 /**
00039  The omp_in_parallel function returns a nonzero value if it is called within the dynamic extent
00040  of a parallel region executing in parallel; otherwise, it returns 0.
00041  */
00042 int omp_in_parallel(void)
00043 {
00044   return _OMP_IN_PARALLEL;
00045 }
00046
00047 /**
00048  The omp_get_thread_num function returns the thread number, within its team, of the thread executing
00049  the function. The thread number lies between 0 and omp_get_num_threads()-1, inclusive. The master
00050  thread of the team is thread 0.
00051  */
00052 int omp_get_thread_num(void)
00053 {
00054   return _OMP_THREAD->thread_num;
00055 }
00056
00057 /**
00058  The omp_set_num_threads function sets the default number of threads to use for subsequent parallel
00059  regions that do not specify a num_threads clause.
00060  */
00061 void omp_set_num_threads(int num_threads)
00062 {
00063   if (!_OMP_IN_PARALLEL && num_threads > 0) {        /* Can only be called when not in parallel */
00064     _omp_modules_resize_tp_vars(num_threads);
00065     _omp_max_threads = num_threads;
00066   }
00067 }
00068
00069 /**
00070  The omp_get_num_threads function returns the number of threads currently in the team executing the
00071  parallel region from which it is called.
00072  */
00073 int omp_get_num_threads(void)
00074 {
00075   return _OMP_THREAD->parent->num_children;
00076 }
00077
00078 /**
00079  The omp_get_max_threads function returns an integer that is guaranteed to be at least as large as
00080  the number of threads that would be used to form a team if a parallel region without a num_threads
00081  clause were to be encountered at that point in the code.
00082  */
00083 int omp_get_max_threads(void)
00084 {
00085   return _omp_max_threads;
00086 }
00087
00088 /**
00089  The omp_get_num_procs function returns the number of processors that are available to the program
00090  at the time the function is called.
00091 */
00092 int omp_get_num_procs(void)
00093 {
00094   return _omp_num_procs;
```

```
00095 }
00096
00097 /**
00098  The omp_set_dynamic function enables or disables dynamic adjustment of the number of threads
00099  available for execution of parallel regions. No implementation required.
00100  */
00101 void omp_set_dynamic(int dynamic_threads)
00102 {
00103   return;
00104 }
00105
00106 /**
00107  The omp_get_dynamic function returns a nonzero value if dynamic adjustment of threads is enabled,
00108  and returns 0 otherwise.
00109  */
00110 int  omp_get_dynamic(void)
00111 {
00112   return 0;
00113 }
00114
00115 /**
00116  The omp_set_nested function enables or disables nested parallelism. No implementation required.
00117  */
00118 void omp_set_nested(int nested)
00119 {
00120   return;
00121 }
00122
00123 /**
00124  The omp_get_nested function returns a nonzero value if nested parallelism is enabled and 0 if it
00125  is disabled.
00126  */
00127 int omp_get_nested(void)
00128 {
00129   return 0;
00130 }
00131
00132 /**
00133  The omp_init_lock function initializes the simple lock associated with the parameter lock for use
00134  in subsequent calls.
00135  */
00136 void omp_init_lock(omp_lock_t *lock)
00137 {
00138   othread_init_lock(lock);
00139 }
00140
00141 /**
00142  The omp_destroy_lock function ensures that the pointed to lock variable lock is uninitialized.
00143  */
00144 void omp_destroy_lock(omp_lock_t *lock)
00145 {
00146   othread_destroy_lock(lock);
00147 }
00148
00149 /**
00150  The omp_set_lock function blocks the thread executing the function until the specified lock is
00151  available and then sets the lock. A simple lock is available if it is unlocked.
00152  */
00153 void omp_set_lock(omp_lock_t *lock)
00154 {
00155   othread_set_lock(lock);
00156 }
00157
00158 /**
00159  The omp_unset_lock function provide the means of releasing ownership of a simple lock.
00160  */
00161 void omp_unset_lock(omp_lock_t *lock)
00162 {
00163   othread_unset_lock(lock);
00164 }
00165
00166 /**
00167  The omp_test_lock function attempts to set a lock but does not block execution of the thread.
00168  Returns a nonzero value if the lock is successfully set; otherwise, it returns zero.
00169  */
00170 int omp_test_lock(omp_lock_t *lock)
00171 {
```

```
00172   return othread_test_lock(lock);
00173 }
00174
00175 /**
00176  The omp_init_nest_lock function initializes the nested lock associated with the parameter lock for
00177  use in subsequent calls.
00178  */
00179 void omp_init_nest_lock(omp_nest_lock_t *lock)
00180 {
00181   othread_init_nest_lock(lock);
00182 }
00183
00184 /**
00185  The omp_destroy_nest_lock function ensures that the pointed to lock variable lock is uninitialized.
00186  */
00187 void omp_destroy_nest_lock(omp_nest_lock_t *lock)
00188 {
00189   othread_destroy_nest_lock(lock);
00190 }
00191
00192 /**
00193  The omp_set_nest_lock function blocks the thread executing the function until the specified lock is
00194  available and then sets the lock. A nestable lock is available if it is unlocked or if it is
00195  already owned by the thread executing the function.
00196  */
00197 void omp_set_nest_lock(omp_nest_lock_t *lock)
00198 {
00199   othread_set_nest_lock(lock);
00200 }
00201
00202 /**
00203  The omp_unset_nest_lock function provide the means of releasing ownership of a nestable lock.
00204  */
00205 void omp_unset_nest_lock(omp_nest_lock_t *lock)
00206 {
00207   othread_unset_nest_lock(lock);
00208 }
00209
00210 /**
00211  The omp_test_nest_lock function attempts to set a lock but does not block execution of the thread.
00212  Returns the new nesting count if the lock is successfully set; otherwise, it returns zero.
00213  */
00214 int omp_test_nest_lock(omp_nest_lock_t *lock)
00215 {
00216   return othread_test_nest_lock(lock);
00217 }
00218
00219 #ifdef CLOCK_SGI_CYCLE                                                    /* Irix */
00220   #define CLOCK_OMPI CLOCK_SGI_CYCLE
00221   #warning "OMPi INFO: Using clock_gettime(CLOCK_SGI_CYCLE) for timings..."
00222 #else
00223   #ifdef CLOCK_HIGHRES                                                  /* Solaris */
00224     #define CLOCK_OMPI CLOCK_HIGHRES
00225     #warning "OMPi INFO: Using clock_gettime(CLOCK_HIGHRES) for timings..."
00226   #else
00227     #ifdef CLOCK_REALTIME                                          /* General solution */
00228       #define CLOCK_OMPI CLOCK_REALTIME
00229       #warning "OMPi INFO: Using clock_gettime(CLOCK_REALTIME) for timings..."
00230     #else
00231       #define CLOCK_OMPI 0                                          /* Last option */
00232       #warning "OMPi INFO: Using clock_gettime(0) for timings..."
00233     #endif
00234   #endif
00235 #endif
00236 /**
00237  The omp_get_wtime function returns a double-precision floating point value equal to the elapsed
00238  wall clock time in seconds since some "time in the past". The actual "time in the past" is
00239  arbitrary, but it is guaranteed not to change during the execution of the application program
00240  */
00241 double omp_get_wtime(void)
00242 {
00243   struct timespec ts;
00244   double t;
00245
00246   clock_gettime(CLOCK_OMPI, &ts);
00247   t = (double)ts.tv_sec + (double) ts.tv_nsec * 1.0E-9;
00248   return t;
```

```
00249 }
00250
00251 /**
00252  The omp_get_wtick function returns a double-precision floating point value equal to the number of
00253  seconds between successive clock ticks.
00254  */
00255 double omp_get_wtick(void)
00256 {
00257   struct timespec ts;
00258   double t;
00259
00260   clock_getres(CLOCK_OMPI, &ts);
00261   t = (double) ts.tv_sec + (double) ts.tv_nsec * 1.0E-9;
00262   return t;
00263 }
```

ompi-0.8.2/lib/omp.h

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*! \file omp.h
00024     \brief The openmp specification functions declarations.
00025  */
00026
00027 #ifndef __OMP_H__
00028 #define __OMP_H__
00029
00030 /*
00031   The following definitions are just for "safety", since they are already declared in the command
00032   line: gcc -D_OPENMP=200203 -D_POMP=200203
00033   This is so that user code like
00034     #ifdef _OPENMP
00035     #include <omp.h>
00036     #endif
00037   works OK.
00038  */
00039 #ifndef _OPENMP
00040 #define _OPENMP 200203
00041 #endif
00042
00043 #ifndef _POMP
00044 #define _POMP 200203
00045 #endif
00046
00047 #include <stdlib.h>
00048 #include <string.h>
00049 #include "othread.h"
00050
00051 #ifdef __cplusplus
00052 extern "C" {
00053 #endif
00054
00055 /* execution environment functions */
00056 int  omp_in_parallel(void);
00057 int  omp_get_thread_num(void);
```

```
00058 void omp_set_num_threads(int num_threads);
00059 int  omp_get_num_threads(void);
00060 int  omp_get_max_threads(void);
00061 int  omp_get_num_procs(void);
00062 void omp_set_dynamic(int dynamic_threads);
00063 int  omp_get_dynamic(void);
00064 void omp_set_nested(int nested);
00065 int  omp_get_nested(void);
00066
00067 /* lock functions */
00068 typedef othread_lock_t omp_lock_t;
00069
00070 void omp_init_lock(omp_lock_t *lock);
00071 void omp_destroy_lock(omp_lock_t *lock);
00072 void omp_set_lock(omp_lock_t *lock);
00073 void omp_unset_lock(omp_lock_t *lock);
00074 int  omp_test_lock(omp_lock_t *lock);
00075
00076 /* nestable lock fuctions */
00077 typedef othread_nest_lock_t omp_nest_lock_t;
00078
00079 void omp_init_nest_lock(omp_nest_lock_t *lock);
00080 void omp_destroy_nest_lock(omp_nest_lock_t *lock);
00081 void omp_set_nest_lock(omp_nest_lock_t *lock);
00082 void omp_unset_nest_lock(omp_nest_lock_t *lock);
00083 int  omp_test_nest_lock(omp_nest_lock_t *lock);
00084
00085 /* timing routines */
00086 double omp_get_wtime(void);
00087 double omp_get_wtick(void);
00088
00089 #ifdef __cplusplus
00090 }
00091 #endif
00092
00093 #endif
```

ompi-0.8.2/lib/ompi.c

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*! \file ompi.c
00024     \brief Private runtime library implementation.
00025  */
00026
00027 /* This is just for debugging: */
00028 /*#define DD fprintf(stderr, "LINE: %d, thread = %d\n", __LINE__, _OMP_THREAD->thread_num)*/
00029
00030 #include <stdio.h>
00031 #include <stdlib.h>
00032 #include <string.h>
00033 #include <unistd.h>
00034 #include <stdarg.h>
00035 #include <time.h>
00036 #include "othread.h"
```

```
00037 #pragma weak ompi_barrier_init = _omp_barrier_init
00038 #pragma weak ompi_barrier_wait = _omp_barrier_wait
00039 #pragma weak ompi_barrier_destroy = _omp_barrier_destroy
00040 #include "ompi.h"
00041
00042 /*
00043  Threads created by another thread become children of the latter. Consequently, they form a tree
00044  structure where the parent points to the first child, and the rest are linked in a list.
00045  Each node in the thread tree (TT) contains info and necessary data for the thread.
00046  */
00047
00048 _ompi_thread_t *_omp_sentinel_thread;          /**< Not a thread - just the root node of the TT */
00049 _ompi_thread_t *_omp_master_thread;  /**< The main thread, the only child of _omp_sentinel_thread */
00050 othread_key_t _ompi_thread_key;      /**< Key for thread-specific data; the data is a pointer to the
00051                                                                                  thread's node in the TT */
00052 int _omp_max_threads;                                                  /**< Maximum number of threads */
00053 int _omp_num_procs;                                                /**< Number of available processors */
00054
00055 int _omp_num_sections = 0;     /**< Counter of total number of section constructs in source files */
00056 int _omp_num_for = 0;             /**< Counter of total number of for constructs in source files */
00057 int _omp_num_single = 0;        /**< Counter of total number of single constructs in source files */
00058
00059 int _omp_serialize = 0;                                  /**< True if a parallel if evaluates to false;
00060                                                                    nested parallels will be serialized */
00061 othread_lock_t _omp_atomic_lock;                                    /**< Global lock for atomic */
00062 othread_lock_t _omp_flush_lock;                                      /**< Global lock for flush */
00063
00064 /**
00065  The pool of threads (linked list); all created threads join the pool waiting for work;
00066  the nodes in the TT actually correspond to threads in the pool.
00067  */
00068 _ompi_execenv_t *_omp_thread_pool;
00069 othread_lock_t  _omp_pool_lock;
00070
00071 int _ompi_initialized = 0;                              /**< Initialization should be done once */
00072
00073 /*
00074  Support functions
00075  */
00076
00077 /**
00078  Reports an error to stderr and exits.
00079  */
00080 _ompi_error(int exitcode, char *format, ...)
00081 {
00082   va_list ap;
00083
00084   va_start(ap, format);
00085   vfprintf(stderr, format, ap);
00086   va_end(ap);
00087
00088   exit(exitcode);
00089 }
00090
00091 /**
00092  Alloc with error handling.
00093  */
00094 void *_omp_alloc(int size)
00095 {
00096   void *a;
00097
00098   if ((a = malloc(size)) == NULL)
00099     _ompi_error(1, "libomp: memory allocation failed\n");
00100   return (a);
00101 }
00102
00103 /**
00104  Calloc with error handling.
00105  */
00106 void *_omp_calloc(int size)
00107 {
00108   void *a;
00109
00110   if ((a = calloc(1, size)) == NULL)
00111     _ompi_error(1, "libomp: memory allocation failed\n");
00112   return (a);
00113 }
```

```
00114
00115 /*
00116  Thread & pool functions
00117  */
00118
00119 /**
00120  The function executed by a thread. Passed to othread_create.
00121  */
00122 void *_ompi_thread_func(void *arg1)                    /**< The execution environment of the thread */
00123 {
00124   _ompi_thread_t *p;                                          /* My parent from my node in TT */
00125   _ompi_execenv_t *arg = (_ompi_execenv_t *) arg1;                      /* Just a typecast */
00126   int alive;                                            /* # running children of my parent */
00127
00128   while (1) {
00129     othread_set_lock(&arg->lock);                                 /* Block waiting for work */
00130     (*(arg->start_routine))(arg->arg);                            /* Execute requested code */
00131     othread_set_lock(&_omp_pool_lock);                            /* Done; reenter the pool */
00132     arg->next = _omp_thread_pool;
00133     _omp_thread_pool = arg;
00134     othread_unset_lock(&_omp_pool_lock);
00135
00136     p = _OMP_THREAD->parent;                                    /* My parent from my node in TT */
00137
00138 /* Decrement number of running siblings */
00139     othread_set_lock(&p->children_count_lock);
00140     alive = --p->children_alive;
00141     othread_unset_lock(&p->children_count_lock);
00142
00143     if (alive == 1)                                 /* If I am the last one, wake up my parent */
00144       othread_unset_lock(&p->join_lock);
00145   }
00146 }
00147
00148 /**
00149  Wake up the head of the pool and give it work to do;
00150  if pool is empty, a new thread will be created.
00151  */
00152 void _ompi_start_thread(void *(*start_routine)(void *), void *arg)
00153 {
00154   _ompi_execenv_t *t;
00155
00156   othread_set_lock(&_omp_pool_lock);
00157   t = _omp_thread_pool;
00158   if (t == NULL) {                                              /* No threads left in the pool */
00159     othread_t thrid;                                  /* So create a new one, OUT of the pool */
00160
00161     othread_unset_lock(&_omp_pool_lock);
00162     t = (_ompi_execenv_t *) _omp_alloc(sizeof(_ompi_execenv_t));      /* Create a new environment */
00163     othread_init_lock(&t->lock);
00164     othread_set_lock(&t->lock);                          /* Block the thread while initializing */
00165     othread_create(&thrid, NULL, _ompi_thread_func, (void *) t);
00166   }
00167   else {                                                        /* Take it out of the pool */
00168     _omp_thread_pool = _omp_thread_pool->next;
00169     othread_unset_lock(&_omp_pool_lock);
00170   }
00171
00172   t->start_routine = start_routine;                          /* Fill up the execution environment */
00173   t->arg = arg;
00174   othread_unset_lock(&t->lock);                                       /* Wake up the thread */
00175 }
00176
00177 /**
00178  Creates all threads and puts them in the pool.
00179  */
00180 void _ompi_create_thread_pool(int num)
00181 {
00182   _ompi_execenv_t *t;
00183   int i;
00184   othread_t thrid;
00185
00186   _omp_thread_pool = NULL;
00187   othread_init_lock(&_omp_pool_lock);
00188   for (i = 0; i < num; i++) {
00189     t = (_ompi_execenv_t *) _omp_alloc(sizeof(_ompi_execenv_t));
00190 /* Lock t->lock so that child sleeps upon entry */
```

```
00191     othread_init_lock(&t->lock);
00192     othread_set_lock(&t->lock);
00193
00194 /* othread_set_lock(&_omp_pool_lock); NOT NEEDED because only the master thread is active. */
00195     t->next = _omp_thread_pool;
00196     _omp_thread_pool = t;
00197 /* othread_unset_lock(&_omp_pool_lock); NOT NEEDED because only the master thread is active. */
00198
00199     othread_create(&thrid, NULL, _ompi_thread_func, (void *) t);      /* Create the actual thread */
00200   }
00201 }
00202
00203 /* Single/sections/for constructs with nowait clauses require bookkeeping to keep track of child
00204  threads that passed through; orphaned cases, too. For each such construct we keep a list of
00205  'rounds' (i.e. in a round all threads pass through - the same construct (e.g. when orphaned) may be
00206  executed many times, hence the need for rounds bookkeeping); in each round every thread that passed
00207  through the construct decrements a counter by 1. The last one removes the info on this round.
00208  Threads may be in different rounds at a given time. All this info is stored in the parent of the
00209  team. Each thread maintains only info about its current round so as to know if it is the last one
00210  to complete the round.
00211  */
00212
00213 /**
00214  Allocate memory for info on single/sections/for (SSF) rounds at the parent of the team.
00215  */
00216 void _ompi_init_ssf_rounds(_ompi_thread_t *parent,                    /**< The parent of the thread */
00217                            int con_type,              /**< type of construct (single/sections/for) */
00218                            int size)                 /**< number of such directives in the program */
00219 {
00220   int i;
00221
00222   parent->con_lock[con_type] = (othread_lock_t *) _omp_alloc(size * sizeof(othread_lock_t));
00223   parent->con_head[con_type] = (_ompi_ssfround_t **) _omp_alloc(size * sizeof(_ompi_ssfround_t *));
00224   parent->con_run[con_type]  = (_ompi_ssfround_t **) _omp_alloc(size * sizeof(_ompi_ssfround_t *));
00225   for (i = 0; i < size; i++) {
00226     othread_init_lock(&parent->con_lock[con_type][i]);
00227     parent->con_run[con_type][i] = parent->con_head[con_type][i] =
00228      (_ompi_ssfround_t *) _omp_alloc(sizeof(_ompi_ssfround_t));
00229     parent->con_head[con_type][i]->next = NULL;
00230     parent->con_head[con_type][i]->ordered_info = NULL;
00231   }
00232 }
00233
00234 /**
00235  Allocate memory for info on SSF rounds for a child thread. This info just points to the current
00236  round (allocated in the parent) that the child executes.
00237  */
00238 void _ompi_init_child_ssf_round(_ompi_thread_t *child,                    /**< The child thread */
00239                                 int con_type,         /**< _OMP_SINGLE, _OMP_SECTIONS or _OMP_FOR */
00240                                 int size)
00241 {
00242   int i;
00243
00244   child->con_encounter[con_type] = (_ompi_ssfround_t **)_omp_alloc(size*sizeof(_ompi_ssfround_t *));
00245   for (i = 0; i < size; i++)
00246     child->con_encounter[con_type][i] = child->parent->con_head[con_type][i];
00247 }
00248
00249 /**
00250  Free the SSF round infos from the parent
00251  */
00252 void _ompi_free_ssf_rounds(_ompi_thread_t *parent, int con_type, int size)
00253 {
00254   int i;
00255
00256   /* TODO: We probably need othread_destroy_lock for every lock !!! */
00257   free(parent->con_lock[con_type]);
00258
00259   for (i = 0; i < size; i++) {
00260     if (parent->con_head[con_type][i]->next != 0) {
00261       if (parent->con_head[con_type][i]->next->ordered_info != 0) {
00262         othread_destroy_lock(&parent->con_head[con_type][i]->next->ordered_info->lock);
00263         {                                          /* this is an alternative without conditions */
00264           int sl;
00265
00266           for (sl = omp_get_num_threads() - 1; sl >= 0; sl--)          /* downwards it is faster */
00267             othread_destroy_lock(
```

```
00268                 &parent->con_head[con_type][i]->next->ordered_info->sleep_locks[sl]);
00269             free(parent->con_head[con_type][i]->next->ordered_info->sleep_locks);
00270           }
00271           free(parent->con_head[con_type][i]->next->ordered_info);
00272         }
00273         free(parent->con_head[con_type][i]->next);
00274       }
00275       free(parent->con_head[con_type][i]);
00276     }
00277     free(parent->con_head[con_type]);
00278     free(parent->con_run[con_type]);
00279 }
00280
00281 /**
00282  This is called upon entry in a parallel region; it creates TT nodes as children of the parent and
00283  starts the threads (which will be taken out of the thread pool).
00284  If num_threads = -1, the team will have _omp_max_threads threads.
00285  */
00286 void _omp_create_team(int num_threads, _ompi_thread_t *parent, void *(*func)(void *), void *shared)
00287 {
00288   _ompi_thread_t *p, *p0;
00289   int i, total;
00290
00291   if (num_threads == -1)
00292     num_threads = _omp_max_threads;
00293
00294   total = (parent == _omp_master_thread && _omp_serialize == 0) ? num_threads : 1;
00295
00296 /* This one is created dynamically during linking and calls _omp_module_resize_tp_vars() for each
00297  source module. This is needed to resize the threadprivate variables array in case the requested
00298  num_threads is > current # threads in the pool. */
00299   _omp_modules_resize_tp_vars(num_threads);
00300
00301   parent->num_children = total;
00302   parent->shared_data = shared;
00303   ompi_barrier_init((ompi_barrier_t *)&parent->barrier, total);
00304   if (_omp_num_single)
00305     _ompi_init_ssf_rounds(parent, _OMP_SINGLE, _omp_num_single);
00306   if (_omp_num_sections)
00307     _ompi_init_ssf_rounds(parent, _OMP_SECTIONS, _omp_num_sections);
00308   if (_omp_num_for)
00309     _ompi_init_ssf_rounds(parent, _OMP_FOR, _omp_num_for);
00310
00311   othread_init_lock(&parent->copyprivate.lock);
00312   othread_init_lock(&parent->join_lock);
00313   othread_init_lock(&parent->children_count_lock);
00314   othread_set_lock(&parent->join_lock);
00315   parent->children_alive = total;
00316
00317 /* Create nodes in TT */
00318   p0 = (_ompi_thread_t *) _omp_alloc(total * sizeof(_ompi_thread_t));
00319   for (i = total-1; i >= 0; i--) {
00320     p = p0 + i;
00321     p->parent = parent;
00322     p->child = NULL;
00323     p->next = parent->child;
00324     parent->child = p;
00325     p->thread_num = i;                                    /* Thread id within the team */
00326     if (parent == _omp_master_thread)
00327       p->real_thread_num = i;
00328     else p->real_thread_num = parent->real_thread_num;
00329     p->num_children = 0;
00330     p->shared_data = NULL;
00331     p->sdn = parent;
00332     p->thrid = parent->thrid;
00333     if (_omp_num_single)
00334       _ompi_init_child_ssf_round(p, _OMP_SINGLE, _omp_num_single);
00335     if (_omp_num_sections)
00336       _ompi_init_child_ssf_round(p, _OMP_SECTIONS, _omp_num_sections);
00337     if (_omp_num_for) {
00338       _ompi_init_child_ssf_round(p, _OMP_FOR, _omp_num_for);
00339       p->for_data = NULL;
00340     }
00341   }
00342
00343 /* Start the threads (except myself - I am the first child, too) */
00344   for (p = parent->child->next; p != 0; p = p->next)
```

```
00345     _ompi_start_thread(func, (void *) p);
00346
00347 /* I am the master and the first child, the rest of the children get their key when they execute
00348  their function */
00349   othread_setspecific(_ompi_thread_key, (void *) parent->child);
00350
00351   if (func != NULL)                                          /* I also run the function */
00352     (*func)((void *) parent->child);
00353 }
00354
00355
00356 /**
00357  Wait till children finish and free all children nodes in the TT.
00358  */
00359 void _omp_destroy_team(_ompi_thread_t *parent)
00360 {
00361   _ompi_thread_t *p, *t;
00362
00363 /* Sleep till all children finish (last one will unlock the join_lock) */
00364   if (parent->num_children > 1)
00365     othread_set_lock(&parent->join_lock);
00366   othread_unset_lock(&parent->join_lock);
00367
00368 /* Free child SSF infos */
00369   if (_omp_num_single)
00370     for (p = parent->child; p != NULL; p = p->next)
00371       free(p->con_encounter[_OMP_SINGLE]);
00372   if (_omp_num_sections)
00373     for (p = parent->child; p != NULL; p = p->next)
00374       free(p->con_encounter[_OMP_SECTIONS]);
00375   if (_omp_num_for)
00376     for (p = parent->child; p != NULL; p = p->next)
00377       free(p->con_encounter[_OMP_FOR]);
00378
00379   free(parent->child);
00380
00381   parent->child = NULL;
00382   parent->num_children = 0;
00383
00384 /* TODO: Do we need this if we are the master_thread ? */
00385   othread_destroy_lock(&parent->join_lock);
00386   othread_destroy_lock(&parent->children_count_lock);
00387   ompi_barrier_destroy((ompi_barrier_t *)&parent->barrier);
00388
00389 /* Free SSF infos from the parent */
00390   if (_omp_num_single)
00391     _ompi_free_ssf_rounds(parent, _OMP_SINGLE, _omp_num_single);
00392   if (_omp_num_sections)
00393     _ompi_free_ssf_rounds(parent, _OMP_SECTIONS, _omp_num_sections);
00394   if (_omp_num_for)
00395     _ompi_free_ssf_rounds(parent, _OMP_FOR, _omp_num_for);
00396
00397 /* I am no longer a child of myself in TT; I assume again my parent node */
00398   othread_setspecific(_ompi_thread_key, (void *) parent);
00399 }
00400
00401 int _omp_assign_key(void *p)
00402 {
00403   return othread_setspecific(_ompi_thread_key, p);
00404 }
00405
00406 /**
00407  This is called from _omp_modules_resize_tp_vars() for each module.
00408  */
00409 void _omp_module_resize_tp_vars(_omp_mod_t *module, int newlen)
00410 {
00411   if (module->tp_vars_structsize > 0 && newlen > module->len_tp_vars) {
00412     char *thp;
00413     int i;
00414
00415     /* could be realloc() */
00416
00417     module->len_tp_vars = newlen;
00418     thp = (char *) _omp_alloc(module->tp_vars_structsize);
00419     memcpy(thp, module->tp_vars, module->tp_vars_structsize);
00420     free(module->tp_vars);
00421     module->tp_vars = malloc(module->len_tp_vars * module->tp_vars_structsize);
```

```
00422     for (i = 0; i < module->len_tp_vars; i++)
00423       memcpy(((char *) module->tp_vars) + i*module->tp_vars_structsize, thp,
00424        module->tp_vars_structsize);
00425     free(thp);
00426   }
00427 }
00428
00429 /**
00430  Called from _omp_modules_init() for each module dynamically
00431  */
00432 void _omp_module_init(_omp_mod_t *module, int num_for, int num_single, int num_sections,
00433  int num_reduction, int g_threadprivate, /* int tp_vars_structsize, */
00434  void (*_init_global_threadprivate_func)())
00435 {
00436   int i;
00437
00438   if (num_reduction)
00439     module->reduction_lock = _omp_alloc(num_reduction * sizeof(othread_lock_t));
00440
00441   for (i = 0; i < num_reduction; i++)
00442     othread_init_lock(&module->reduction_lock[i]);
00443
00444   module->len_tp_vars = _omp_max_threads;
00445 /*  module->tp_vars_structsize = tp_vars_structsize;
00446   if (module->tp_vars_structsize > 0 )
00447     module->tp_vars = _omp_calloc(module->len_tp_vars * module->tp_vars_structsize);*/
00448   if (g_threadprivate) (*_init_global_threadprivate_func)();
00449
00450   module->for_ofs = _omp_num_for;
00451   _omp_num_for += num_for;
00452
00453   module->single_ofs = _omp_num_single;
00454   _omp_num_single += num_single;
00455
00456   module->sections_ofs = _omp_num_sections;
00457   _omp_num_sections = num_sections;
00458 }
00459
00460 /**
00461  Get the number of processors online.
00462  */
00463 int _omp_get_num_procs()
00464 {
00465   int np;
00466 #if defined(__sgi)
00467   np = sysconf(_SC_NPROC_ONLN);
00468 #else
00469   np = sysconf(_SC_NPROCESSORS_ONLN);
00470   if (np == 0) {
00471     fprintf(stderr, "libompi: could not get the number of processors; assuming 1\n");
00472     np = 1;
00473   }
00474 #endif
00475   return np;
00476 }
00477
00478 /**
00479  First function called.
00480  */
00481 int _omp_initialize()
00482 {
00483   char *s;
00484   int i;
00485
00486   if (_ompi_initialized)
00487     return 0;
00488
00489   _omp_max_threads = _omp_num_procs = _omp_get_num_procs();
00490
00491   s = (char *) getenv("OMP_NUM_THREADS");
00492   if (s != 0 && sscanf(s, "%d", &i) == 1 && i > 0)
00493     _omp_max_threads = i;
00494
00495   _omp_modules_initialize();              /* Created during linking, sets _omp_num_single, etc. */
00496
00497   othread_setconcurrency(_omp_num_procs);
00498
```

```
00499   othread_init_lock(&_omp_atomic_lock);
00500   othread_init_lock(&_omp_flush_lock);
00501
00502   _omp_sentinel_thread = (_ompi_thread_t *) _omp_alloc(sizeof(_ompi_thread_t));
00503   _omp_master_thread = (_ompi_thread_t *) _omp_alloc(sizeof(_ompi_thread_t));
00504
00505   _omp_sentinel_thread->child = _omp_master_thread;
00506   _omp_sentinel_thread->next = 0;
00507   _omp_sentinel_thread->parent = 0;
00508   _omp_sentinel_thread->sdn = 0;
00509   _omp_sentinel_thread->num_children = 1;
00510   _omp_sentinel_thread->thread_num = 0;
00511   _omp_sentinel_thread->thrid = othread_self();
00512   _omp_sentinel_thread->shared_data = 0;
00513
00514   if (_omp_num_single)
00515     _ompi_init_ssf_rounds(_omp_sentinel_thread, _OMP_SINGLE, _omp_num_single);
00516   if (_omp_num_sections)
00517     _ompi_init_ssf_rounds(_omp_sentinel_thread, _OMP_SECTIONS, _omp_num_sections);
00518   if (_omp_num_for) {
00519     _ompi_init_ssf_rounds(_omp_sentinel_thread, _OMP_FOR, _omp_num_for);
00520     _omp_sentinel_thread->for_data = 0;
00521   }
00522   ompi_barrier_init((ompi_barrier_t *)&_omp_sentinel_thread->barrier, 1);
00523
00524   _omp_master_thread->child = 0;
00525   _omp_master_thread->next = 0;
00526   _omp_master_thread->parent = _omp_sentinel_thread;
00527   _omp_master_thread->sdn = _omp_master_thread;
00528   _omp_master_thread->num_children = 0;
00529   _omp_master_thread->thread_num = 0;
00530   _omp_master_thread->real_thread_num = 0;
00531   _omp_master_thread->thrid = othread_self();
00532   _omp_master_thread->shared_data = 0;
00533
00534   if (_omp_num_single)
00535     _ompi_init_child_ssf_round(_omp_master_thread, _OMP_SINGLE, _omp_num_single);
00536   if (_omp_num_sections)
00537     _ompi_init_child_ssf_round(_omp_master_thread, _OMP_SECTIONS, _omp_num_sections);
00538   if (_omp_num_for) {
00539     _ompi_init_child_ssf_round(_omp_master_thread, _OMP_FOR, _omp_num_for);
00540     _omp_master_thread->for_data = 0;
00541   }
00542
00543 /* In this key we store the node in TT for each thread to access */
00544   othread_key_create(&_ompi_thread_key, 0);
00545   othread_setspecific(_ompi_thread_key, (void *) _omp_master_thread);
00546
00547   _ompi_create_thread_pool(_omp_max_threads);
00548
00549   _ompi_initialized = 1;
00550   return 1;
00551 }
00552
00553 /*
00554  * Barrier
00555  */
00556
00557 /**
00558  Initializes a barrier.
00559  */
00560 void _omp_barrier_init(ompi_barrier_t *void_bar, int n)
00561 {
00562 #define bar ((_omp_barrier_t *)void_bar)                    /* just a typecast */
00563   int i;
00564
00565   bar->lock = (othread_lock_t *) _omp_alloc(n * sizeof(othread_lock_t));
00566   for (i = 0; i < n; i++) {
00567     othread_init_lock(&bar->lock[i]);
00568     othread_set_lock(&bar->lock[i]);
00569   }
00570   othread_init_lock(&bar->c_lock);
00571   bar->total = bar->left = n;
00572 #undef bar
00573 }
00574
00575 /**
```

```
00576  Called by all threads at explicit and implicit barriers. The last one to arrive unlocks all the
00577  waiting threads.
00578  */
00579 void _omp_barrier_wait(ompi_barrier_t *void_bar)
00580 {
00581 #define bar ((_omp_barrier_t *)void_bar)                    /* just a typecast */
00582   _ompi_thread_t *th = _OMP_THREAD;
00583   int i;
00584
00585   othread_set_lock(&bar->c_lock);
00586   if (--bar->left == 0) {
00587     bar->left = bar->total;
00588     othread_unset_lock(&bar->c_lock);
00589     for (i = 0; i < th->thread_num; i++)          /* Unlock all the threads except for the caller */
00590        othread_unset_lock(&bar->lock[i]);                    /* first the "lower" ones, */
00591     for (i++; i < bar->total; i++)                           /* then the "higher" ones */
00592        othread_unset_lock(&bar->lock[i]);
00593   } else {
00594     othread_unset_lock(&bar->c_lock);
00595     othread_set_lock(&bar->lock[th->thread_num]);
00596   }
00597 #undef bar
00598 }
00599
00600 /**
00601  Destroys a barrier.
00602  */
00603 void _omp_barrier_destroy(ompi_barrier_t *void_bar)
00604 {
00605 #define bar ((_omp_barrier_t *)void_bar)                    /* just a typecast */
00606   int i;
00607
00608   for (i = 0; i < bar->total; i++)
00609     othread_destroy_lock(&bar->lock[i]);
00610   free(bar->lock);
00611   othread_destroy_lock(&bar->c_lock);
00612 #undef bar
00613 }
00614
00615 /**
00616  Flush should instruct the compiler to restore all register contents back to memory and the
00617  hardware to flush its buffers. Only the latter is supported. The compiler could be implicitly
00618  instructed to flush all the registers with a fake setjmp.
00619  */
00620 void _omp_flush_all()
00621 {
00622 #if defined(__sgi)
00623 /* for ORIGIN: nothing, it has sequential consistency */
00624 #elif defined(__sparc)
00625   asm("stbar");
00626 #else
00627 /* General solution from OdinMP: lock & unlock something */
00628   othread_set_lock(&_omp_flush_lock);
00629   othread_unset_lock(&_omp_flush_lock);
00630 #endif
00631 }
00632
00633 /**
00634   This should only flush the variable x, resulting in better performance. Compiler & hardware
00635   specific.
00636  */
00637 void _omp_flush(void *x)
00638 {
00639   _omp_flush_all();
00640 }
00641
00642 /**
00643  Initializations (per-thread) needed for single/sections/for (SSF) directives.
00644  Returns 1 if I am the first to start a new SSF round, 0 otherwise.
00645  */
00646 int _omp_init_directive(int type,                    /**< type = OMP_FOR, OMP_SECTIONS, OMP_SINGLE */
00647  int k,                              /**< Id of the corresponding directive (0 <= k < _omp_num_xxx) */
00648  int data,                  /**< Next section to execute (for SINGLE) or initial lower bound (FOR) */
00649  int incr,                                                    /**< FOR's increment */
00650  int has_ordered,
00651  char sched)
00652 {
```

```
00653   int res = 0;
00654   _ompi_ssfround_t *p;
00655   _ompi_thread_t *th = _OMP_THREAD;
00656
00657   othread_set_lock(&th->parent->con_lock[type][k]);
00658
00659 /* I am the first thread to start a new round */
00660   if (th->con_encounter[type][k] == th->parent->con_run[type][k]) {
00661 /* Add a new round node in the SSF info */
00662     p = (_ompi_ssfround_t *) _omp_alloc(sizeof(_ompi_ssfround_t));
00663     th->parent->con_run[type][k]->next = p;
00664     p->left = 0;                                          /* No thread passed through yet */
00665     p->data = data;
00666
00667     if (type == _OMP_FOR && has_ordered) {
00668       p->ordered_info = (_omp_ordered_info_t *)
00669       _omp_alloc(sizeof(_omp_ordered_info_t));
00670       p->ordered_info->ordered_lb = data;
00671       p->ordered_info->incr = incr;
00672       {                                                   /* This is an alternative without conditions */
00673         int sl;
00674
00675         p->ordered_info->sleep_locks = (othread_lock_t *)
00676           malloc(omp_get_num_threads() * sizeof(othread_lock_t));
00677         for (sl = omp_get_num_threads() - 1; sl >= 0; sl--) {          /* downwards it is faster */
00678           othread_init_lock(&p->ordered_info->sleep_locks[sl]);
00679         }
00680       }
00681       othread_init_lock(&p->ordered_info->lock);
00682     }
00683     else
00684       p->ordered_info = NULL;
00685
00686     if (type == _OMP_FOR && sched == 'r') {
00687       char *tmp, tmp1[80], tmp2[80], *c;
00688       tmp = getenv("OMP_SCHEDULE");
00689       tmp1[0] = 0;
00690       if (tmp != NULL) {
00691         strcpy(tmp1, tmp);
00692         if ((c = strchr(tmp1, ',')) != NULL)
00693           *c = ' ';
00694       }
00695       p->sched_info.chunksize = -1;
00696       if (tmp == NULL)
00697         p->sched_info.sched = _OMP_STATIC;
00698       else {
00699         sscanf(tmp1, "%s %d", tmp2, &p->sched_info.chunksize);
00700         if (strcmp(tmp2, "static") == 0)
00701           p->sched_info.sched = _OMP_STATIC;
00702         else if (strcmp(tmp2, "dynamic") == 0)
00703           p->sched_info.sched = _OMP_DYNAMIC;
00704         else if (strcmp(tmp2, "guided") == 0)
00705           p->sched_info.sched = _OMP_GUIDED;
00706           else {
00707             fprintf(stderr, "WARNING: libompi: incorrect value of environment variable "
00708              "OMP_SCHEDULE, `static' assumed\n");
00709             p->sched_info.sched = _OMP_STATIC;
00710           }
00711         }
00712       }
00713
00714       th->parent->con_run[type][k] = p;
00715       res = 1;
00716     }
00717
00718 /* Every thread */
00719     p = th->con_encounter[type][k];
00720     p->left++;                                            /* One more passes through this round */
00721     th->con_encounter[type][k] = p->next;
00722
00723 /* I am the last thread in this round */
00724     if (p != th->parent->con_head[type][k] && p->left == th->parent->num_children) {
00725 /* Remove head node from the SSF round list */
00726       th->parent->con_head[type][k]->next = p->next;
00727
00728     if (p->ordered_info != 0) {
00729       othread_destroy_lock(&p->ordered_info->lock);
```

```
00730       {                                          /* this is an alternative without conditions */
00731         int sl;
00732
00733         for (sl = omp_get_num_threads() - 1; sl >= 0; sl--)          /* downwards it is faster */
00734           othread_destroy_lock(&p->ordered_info->sleep_locks[sl]);
00735         free(p->ordered_info->sleep_locks);
00736       }
00737       free(p->ordered_info);
00738       p->ordered_info = NULL;
00739     }
00740     free(p);
00741   }
00742
00743   othread_unset_lock(&th->parent->con_lock[type][k]);
00744   return res;
00745 }
00746
00747 /*
00748  * SECTIONS
00749  */
00750
00751 int _omp_get_next_section(int k)
00752 {
00753   int res;
00754   _ompi_thread_t *th = _OMP_THREAD;
00755
00756   othread_set_lock(&th->parent->con_lock[_OMP_SECTIONS][k]);
00757   res = --th->con_encounter[_OMP_SECTIONS][k]->data;
00758   othread_unset_lock(&th->parent->con_lock[_OMP_SECTIONS][k]);
00759   return res;
00760 }
00761
00762 /*
00763  * ORDERED
00764  */
00765
00766 /**
00767   Plan: all sleep_locks are initially UNLOCKED. So when the threads go in, they check if it's
00768   their turn. If so, they go on and do their job without sleeping.
00769   When they are done, they "atomically" increase lb.
00770   If it was not their turn, they should sleep. But they should do this while lb is locked;
00771   otherwise they might [check, (lb is increased by some other thread), sleep for ever].
00772   So they
00773     1. Test-set their sleep lock,
00774     2. Set struct access lock, Check lb, Unset struct access lock,
00775     3. Lock their sleep lock.
00776   If the "working" thread changed lb between steps (1,2) no problem, because it unlocked their
00777   lock, invalidating step (1) and they won't sleep when reaching step (3). So the next thread
00778   immediately starts "working".
00779   If the "working" thread changed lb between steps (2,3) no problem, because it unlocked their
00780   lock, invalidating step (1) and they won't sleep when reaching step (3).  So the next thread
00781   immediately starts "working".
00782 */
00783 void _omp_ordered_begin()
00784 {
00785   _ompi_thread_t *th = _OMP_THREAD;
00786   _ompi_ssfround_t *enc = th->con_encounter[_OMP_FOR][th->for_data->id];
00787
00788   if (enc->ordered_info != 0) {
00789 /* implied flush */
00790     othread_set_lock(&enc->ordered_info->lock);
00791 /* These "lb > ordered_lb" OR "lb < ordered_lb" checks are neccesary, because for example
00792    thread #0 could handle loops 0 to 24 while all other threads are asleep */
00793     while (enc->ordered_info->incr > 0 ?              /* return one of the following conditions: */
00794       th->for_data->lb > enc->ordered_info->ordered_lb :
00795       th->for_data->lb < enc->ordered_info->ordered_lb) {                    /* while not my turn */
00796 /* before sleeping, check WELL! */
00797       othread_test_lock(&enc->ordered_info->sleep_locks[omp_get_thread_num()]);
00798       othread_unset_lock(&enc->ordered_info->lock);
00799 /* lb could be changed by another thread here. No harm done, I won't sleep */
00800       othread_set_lock(&enc->ordered_info->sleep_locks[omp_get_thread_num()]);      /* SLEEP HERE */
00801 /* lb could be changed by another thread here. No harm done, I will wake up */
00802 /* OK, now may be my turn. Safely unlock my sleep lock */
00803       othread_test_lock(&enc->ordered_info->sleep_locks[omp_get_thread_num()]);
00804       othread_unset_lock(&enc->ordered_info->sleep_locks[omp_get_thread_num()]);
00805       othread_set_lock(&enc->ordered_info->lock);
00806     }
```

```
00807 /* Now it IS my turn. I unlock the struct lock and enter "ordered" */
00808     othread_unset_lock(&enc->ordered_info->lock);
00809   }
00810 }
00811
00812 void _omp_ordered_end()
00813 {
00814   _ompi_thread_t *th = _OMP_THREAD;
00815   _ompi_ssfround_t *enc = th->con_encounter[_OMP_FOR][th->for_data->id];
00816
00817   if (enc->ordered_info != 0) {
00818     othread_set_lock(&enc->ordered_info->lock);
00819     enc->ordered_info->ordered_lb += enc->ordered_info->incr;
00820 /* implied flush */
00821     {  //alternative wake up method without using conditions
00822       int sl;
00823
00824         for (sl = omp_get_num_threads() - 1; sl >= 0; sl--) {          /* downwards it is faster */
00825           /* safely unlock all locks */
00826           othread_test_lock(&enc->ordered_info->sleep_locks[sl]);
00827           othread_unset_lock(&enc->ordered_info->sleep_locks[sl]);
00828         }
00829     }
00830     othread_unset_lock(&enc->ordered_info->lock);
00831   }
00832 }
00833
00834 /*
00835  * FOR
00836  */
00837
00838 /**
00839  Necessary for nested ordered.
00840  */
00841 void _omp_push_for_data(int k, int lb)
00842 {
00843   _omp_for_data_t *p;
00844   _ompi_thread_t *th = _OMP_THREAD;
00845
00846   p = (_omp_for_data_t *) _omp_alloc(sizeof(_omp_for_data_t));
00847   p->id = k;
00848   p->lb = lb;
00849   p->next = th->for_data;
00850   th->for_data = p;
00851 }
00852
00853 void _omp_pop_for_data()
00854 {
00855   _omp_for_data_t *p;
00856   _ompi_thread_t *th = _OMP_THREAD;
00857
00858   p = th->for_data;
00859   th->for_data = p->next;
00860   free(p);
00861 }
00862
00863 int _omp_static_bounds(int ub, int lb, int k, int step, int chunksize, int *start, int *end,
00864  int nchunks, int init_start, int *i)
00865 {
00866   if (chunksize == -1) {
00867     if (*i == 0) {
00868       (*i)++;
00869       if (((step >= 0) && (*end > ub)) || ((step < 0) && (*end < ub)))
00870         *end = ub;
00871       return 1;
00872     }
00873     else return 0;
00874   }
00875
00876   if (*i < nchunks) {
00877     *start = init_start + chunksize * step * (*i);
00878     *end = *start + chunksize*step;
00879     if (((step >= 0) && (*end > ub)) || ((step < 0) && (*end < ub)))
00880       *end = ub;
00881     (*i)+= _OMP_THREAD->parent->num_children;
00882     return 1;
00883   }
```

```
00884   else return 0;
00885 }
00886
00887 int _omp_dynamic_bounds(int ub, int lb, int k, int step, int chunksize, int *start, int *end,
00888  int d1, int d2, int *d3)
00889 {
00890   int newlb;
00891   _ompi_thread_t *th = _OMP_THREAD;
00892   int *data = &th->con_encounter[_OMP_FOR][k]->data;
00893
00894   othread_set_lock(&th->parent->con_lock[_OMP_FOR][k]);
00895   newlb = *data;
00896   (*data) += step *chunksize;
00897   othread_unset_lock(&th->parent->con_lock[_OMP_FOR][k]);
00898
00899   if (step >= 0) {
00900     if (newlb < ub) {
00901       *start = newlb;
00902       *end = *start + chunksize*step;
00903       if (*end > ub)
00904         *end = ub;
00905       return 1;
00906     }
00907     else return 0;
00908   }
00909   else {
00910     if (newlb > ub) {
00911       *start = newlb;
00912       *end = *start + chunksize*step;
00913       if (*end < ub)
00914         *end = ub;
00915       return 1;
00916   }
00917     else return 0;
00918   }
00919 }
00920
00921 int _omp_guided_bounds(int ub, int lb, int k, int step, int chunksize, int *start, int *end,
00922  int d1, int d2, int *d3)
00923 {
00924   int newlb, iters;
00925   div_t d;
00926   _ompi_thread_t *th = _OMP_THREAD;
00927   int *data = &th->con_encounter[_OMP_FOR][k]->data;
00928
00929   othread_set_lock(&th->parent->con_lock[_OMP_FOR][k]);
00930   newlb = *data;
00931   d = div(ub - newlb, step * th->parent->num_children);
00932   iters = d.quot;
00933   if (d.rem != 0)
00934     iters++;
00935   if (iters > chunksize)
00936     chunksize = iters;
00937
00938   (*data) += step * chunksize;
00939   othread_unset_lock(&th->parent->con_lock[_OMP_FOR][k]);
00940
00941   if (step >= 0) {
00942     if (newlb < ub) {
00943       *start = newlb;
00944       *end = *start + chunksize*step;
00945       if (*end > ub)
00946         *end = ub;
00947       return 1;
00948     }
00949     else return 0;
00950   }
00951   else {
00952     if (newlb > ub) {
00953       *start = newlb;
00954       *end = *start + chunksize*step;
00955       if (*end < ub)
00956         *end = ub;
00957       return 1;
00958     }
00959     else return 0;
00960   }
```

```
00961 }
00962
00963 void _omp_static_bounds_default(int ub, int lb, int step, int *start, int *end)
00964 {
00965   int chunksize, iters, nchunks;
00966   div_t diters, dchunksize;
00967   int N = _OMP_THREAD->parent->num_children;
00968   int myid = _OMP_THREAD->thread_num;
00969
00970   diters = div(ub - lb, step);                          /* total number of iterations */
00971   iters = diters.quot;
00972   if (diters.rem != 0) iters++;
00973
00974   if (iters <= N) nchunks = iters, chunksize = 1;
00975   else {
00976     nchunks = N;
00977     dchunksize = div(iters, N);
00978     chunksize = dchunksize.quot;                        /* iterations in a chunk */
00979     iters = dchunksize.rem;
00980     if (iters)
00981       chunksize++;                                      /* first iters threads get this chunksize */
00982   }
00983
00984   if (myid < nchunks) {
00985     if (myid < iters || iters == 0) {                   /* I get a full chunk */
00986       *start = lb + chunksize*step*myid;
00987       *end = *start + chunksize*step;
00988     }
00989     else {                                              /* I get a smaller chunk */
00990       *start = lb + chunksize*step*iters + (chunksize - 1)*step*(myid - iters);
00991       *end = *start + (chunksize - 1)*step;
00992     }
00993   }
00994 /* otherwise, there is nothing for me */
00995   else *start = *end = 0;                               /* force no computations */
00996 }
00997
00998 void _omp_static_bounds_chunk(int ub, int lb, int step, int chunksize, int *nchunks,
00999  int *init_start)
01000 {
01001   div_t diters, dnchunks;
01002   int iters;
01003
01004   diters = div(ub - lb, step);
01005   iters = diters.quot;
01006   if (diters.rem != 0)
01007     iters++;
01008
01009   dnchunks = div(iters, chunksize);
01010   *nchunks = dnchunks.quot;
01011   if (dnchunks.rem != 0)
01012     (*nchunks)++;
01013
01014   *init_start = lb + _OMP_THREAD->thread_num*step*chunksize;
01015 }
```

ompi-0.8.2/lib/ompi.h

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
```

```
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*! \file ompi.h
00024     \brief Private runtime library interface.
00025  */
00026
00027 #ifndef __OMPI_H__
00028 #define __OMPI_H__
00029
00030 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
00031  *                                                             *
00032  *  TYPES                                                      *
00033  *                                                             *
00034  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
00035
00036 /**
00037  Holds the data of copyprivate vars as pure bytes.
00038  */
00039 typedef struct {
00040   char *data;
00041   othread_t owner;
00042   int del_counter;
00043   othread_lock_t lock;
00044 } _omp_cpriv_t;
00045
00046 /**
00047  Ordered data.
00048  */
00049 typedef struct {
00050   int ordered_lb;               /**< The lower bound of the chunk that should be next (serially) */
00051   int incr;                                          /**< Is > 0 is FOR's step is positive */
00052   othread_lock_t *sleep_locks;          /**< Threads sleep in these locks until their order comes */
00053   othread_lock_t lock;
00054 } _omp_ordered_info_t;
00055
00056 /**
00057  FOR schedule info.
00058  */
00059 typedef struct {
00060   int sched;
00061   int chunksize;
00062 } _omp_sched_info_t;
00063
00064 /**
00065  Directive data : node in the SSF rounds list.
00066  */
00067 typedef struct _ompi_ssfround_s {
00068   int data;           /**< Used in for (original lb) and in sections (section id to execute next) */
00069   int left;                                    /**< # threads that have passed through this round */
00070   _omp_ordered_info_t *ordered_info;
00071   _omp_sched_info_t sched_info;
00072   struct _ompi_ssfround_s *next;
00073 } _ompi_ssfround_t;
00074
00075 /**
00076  This structure is only used to keep track of ordered fors.
00077  It is a stack for the case of nested ordereds.
00078  */
00079 typedef struct _omp_for_data_s {
00080   int id;                                                              /**< Directive number */
00081   int lb;                          /**< The lower bound of the chunk that I am currently executing */
00082   struct _omp_for_data_s *next;
00083 } _omp_for_data_t;
00084
00085 /**
00086  Barrier structure
00087  */
00088 typedef struct {
00089   othread_lock_t *lock;
00090   othread_lock_t c_lock;
00091   int total, left;
00092 } _omp_barrier_t;
00093
00094 /**
```

```
00095  The execution environment of ompi's threads;
00096  The pool of threads is a linked list of such structures.
00097  */
00098 typedef struct _ompi_execenv_s {
00099   void *(*start_routine)(void *);                          /**< What function to execute */
00100   void *arg;                                               /**< The function's argument */
00101   othread_lock_t lock;                          /**< To wait for work; parent will unlock it */
00102   struct _ompi_execenv_s *next;                            /**< Next node in the pool */
00103 } _ompi_execenv_t;
00104
00105 /**
00106  Thread tree (TT) nodes.
00107  */
00108 typedef struct _ompi_thread_s {
00109   struct _ompi_thread_s *child, *next, *parent;
00110   int num_children;
00111   int thread_num;                                          /**< Thread id within the team */
00112   int real_thread_num;                    /**< Serial thread id used only to index threadprivate */
00113   othread_t thrid;                                         /**< The othread id */
00114   void *shared_data;                           /**< Pointer to shared struct of current function */
00115 /* Where we will get shared data from; normally our parent, except at a false parallel if where we
00116  get it from ourselves since we are the only thread to execute the region. */
00117   struct _ompi_thread_s *sdn;
00118   _omp_barrier_t barrier;                                  /**< Barrier for my children */
00119   othread_lock_t *con_lock[3];                             /**< For SSF rounds handling */
00120   _ompi_ssfround_t **con_head[3];                /**< Start of SSF round list (for my children) */
00121   _ompi_ssfround_t **con_run[3];                   /** End of SSF round list (for my children) */
00122   _ompi_ssfround_t **con_encounter[3];     /**< For me; my current SSF round in parent's SSF list */
00123   _omp_cpriv_t copyprivate;     /**< For copyprivate; owner will store data here and the rest of the
00124                                                                  children will take it from here */
00125   othread_lock_t join_lock;    /**< Parent sleeps on this lock waiting for all children to finish */
00126   int children_alive;                                      /**< Children not finished yet */
00127   othread_lock_t children_count_lock;
00128   _omp_for_data_t *for_data;                                   /**< My own ordered info */
00129 } _ompi_thread_t;
00130
00131 /**
00132  Module info
00133  */
00134 typedef struct {
00135   othread_lock_t *reduction_lock;
00136
00137   void *tp_vars;                                       /**< Struct with threadprivate vars */
00138   int tp_vars_structsize;
00139   int len_tp_vars;                                  /**< Length of array (number of elements) */
00140   int for_ofs, single_ofs, sections_ofs;
00141 } _omp_mod_t;
00142
00143
00144 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
00145  *                                                             *
00146  *  FUNCTION PROTOTYPES                                        *
00147  *                                                             *
00148  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
00149
00150
00151 void _omp_create_team(int num_threads, _ompi_thread_t *parent, void *(*func)(void *), void *shared);
00152 void _omp_destroy_team(_ompi_thread_t *parent);
00153
00154 /**
00155   ompi_barrier_t is declared here as void, to enable the user to define his own barrier type
00156  */
00157 #define ompi_barrier_t void
00158
00159 /* These functions are declared as weak symbols, so that they can be overriden by a user defined
00160    function */
00161 void ompi_barrier_init(ompi_barrier_t *bar, int n);
00162 void ompi_barrier_wait(ompi_barrier_t *bar);
00163 void ompi_barrier_destroy(ompi_barrier_t *bar);
00164
00165 void _omp_module_init(_omp_mod_t *module, int num_for, int num_single, int num_sections,
00166  int num_reduction, int g_threadprivate, void (*_init_global_threadprivate_func)());
00167
00168 int _omp_initialize();
00169
00170 int _omp_assign_key(void *p);
00171
```

```
00172 void _omp_flush(void *);
00173 void _omp_flush_all();
00174
00175 int _omp_init_directive(int type, int k, int data, int incr, int has_ordered, char sched);
00176
00177 int  _omp_get_next_section(int k);
00178
00179 void _omp_ordered_begin();
00180 void _omp_ordered_end();
00181
00182 void _omp_push_for_data(int k, int lb);
00183 void _omp_pop_for_data();
00184 int  _omp_get_for_schedule();
00185
00186 int _omp_guided_bounds(int ub, int lb, int k, int step, int chunksize,
00187  int *start, int *end, int, int, int *);
00188 int _omp_dynamic_bounds(int ub, int lb, int k, int step, int chunksize,
00189  int *start, int *end, int, int, int *);
00190 int _omp_static_bounds(int ub, int lb, int k, int step, int chunksize,
00191  int *start, int *end, int nchunks, int init_start, int *i);
00192 void _omp_static_bounds_default(int ub, int lb, int step, int *start, int *end);
00193 void _omp_static_bounds_chunk(int ub, int lb, int step, int chunksize, int *nchunks,
00194  int *init_start);
00195
00196 void _omp_modules_initialize();
00197 void _omp_modules_init_threadprivate();
00198 void _omp_module_resize_tp_vars(_omp_mod_t *module, int newlen);
00199 void _omp_modules_resize_tp_vars(int newlen);
00200
00201 extern _ompi_thread_t *_omp_sentinel_thread, *_omp_master_thread;
00202 extern othread_key_t _ompi_thread_key;
00203 extern int _omp_max_threads;
00204
00205 extern int _omp_num_sections, _omp_num_for, _omp_num_single;
00206 extern int _omp_serialize;
00207
00208 extern int _omp_num_procs;
00209
00210 extern othread_lock_t _omp_atomic_lock;
00211
00212 extern void *_omp_alloc(int size);
00213 extern void *_omp_calloc(int size);
00214
00215 /* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
00216  *                                                       *
00217  *  DEFINITIONS USED IN GENERATED CODE                   *
00218  *                                                       *
00219  * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
00220
00221
00222 #ifndef _OPENMP
00223 #define _OPENMP       200203
00224 #endif
00225
00226 #define _OMP_SINGLE   0
00227 #define _OMP_SECTIONS 1
00228 #define _OMP_FOR      2
00229
00230 #define _OMP_STATIC   0
00231 #define _OMP_DYNAMIC  1
00232 #define _OMP_GUIDED   2
00233 #define _OMP_RUNTIME  3
00234
00235 #define _omp_run_single(k) _omp_init_directive(_OMP_SINGLE, k, 0, 0, 0, 0)
00236 #define _omp_init_sections(k,d) _omp_init_directive(_OMP_SECTIONS, k, d, 0, 0, 0)
00237
00238 /** Gets the thread's node in the Thread Tree (TT) */
00239 #define _OMP_THREAD ((_ompi_thread_t *) othread_getspecific(_ompi_thread_key))
00240
00241 /** Shared variable reference */
00242 #define _OMP_VARREF(f,i) (*((f ## _vars *) (_OMP_THREAD->sdn->shared_data))->i)
00243
00244 /** Declaration of a structure holding the shared variables */
00245 #define _OMP_PARALLEL_DECL_VARSTRUCT(s) \
00246   s ## _vars s ## _var
00247
00248 /** Initialization of a shared variable in the structure */
```

```
00249 #define _OMP_PARALLEL_INIT_VAR(s,v) \
00250   s ## _var. v = &(v)
00251
00252
00253 #endif      /* __OMPI_H__ */
```

ompi-0.8.2/lib/othread.h

```
00001 /*
00002  OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*! \file othread.h
00024     \brief The wrapper functions implementations for pthread.h or other libraries.
00025  */
00026
00027 #ifndef __OTHREAD_H__
00028 #define __OTHREAD_H__
00029
00030 /* If you want to use Solaris threads you should "./configure --enable-solaristhreads=yes" */
00031
00032 /* First include the machine-dependent types */
00033 #ifdef SOLARISTHREADS
00034 #include "sparc_othread_types.h"
00035 #else /* _GENERIC */
00036 #include "generic_othread_types.h"
00037 #endif
00038 /* And then declare below the machine-independent types */
00039
00040 /** Create Thread.
00041  The othread_create() function creates a thread with the specified attributes and runs the C
00042  function start_routine in the thread with the single pointer argument specified. The new thread
00043  may, but does not always, begin running before othread_create() returns. If othread_create()
00044  completes successfully, the othread handle is stored in the contents of the location referred to
00045  by thread. Returns 0 if successful.
00046 */
00047 int othread_create(
00048  othread_t *thread,                                  /**< (Out) Othread handle to the created thread */
00049  const othread_attr_t *attr,     /**< (In) The thread attributes object containing the attributes to
00050     be associated with the newly created thread. If NULL, the default thread attributes are used. */
00051  void *(*start_routine)(void *),/**< (In) The function to be run as the new threads start routine */
00052  void *arg);                    /**< (In) An address for the argument for the threads start routine */
00053
00054 /** Compare Two Threads.
00055  The othread_equal() function compares two Othread handles for equality.
00056  Returns 1 if the Othread handles refer to the same thread, 0 otherwise.
00057 */
00058 int othread_equal(
00059  othread_t t1,                                                /**< (In) Othread handle for thread 1 */
00060  othread_t t2);                                               /**< (In) Othread handle for thread 2 */
00061
00062 /** Get Thread Local Storage Value by Key.
00063  The othread_getspecific() function retrieves the thread local storage value associated with the
00064  key. othread_getspecific() may be called from a data destructor. The thread local storage value is
00065  a variable of type void * that is local to a thread, but global to all of the functions called
00066  within that thread. It is accessed via the key. Returns NULL if key out of range.
00067 */
```

```
00068 void *othread_getspecific(
00069  othread_key_t key);    /**< (In) The thread local storage key returned from othread_key_create() */
00070
00071 /** Create Thread Local Storage Key.
00072  The othread_key_create() function creates a thread local storage key for the process and associates
00073  the destructor function with that key. After a key is created, that key can be used to set and get
00074  per-thread data pointer. When othread_key_create() completes, the value associated with the newly
00075  created key is NULL. Returns 0 if successful.
00076 */
00077 int othread_key_create(
00078  othread_key_t *key,/**<(Out) The address of the variable to contain the thread local storage key */
00079  void (*destructor)(void *));       /**< (In) The address of the function to act as a destructor for
00080                                                                    this thread local storage key */
00081 /** Get Othread Handle.
00082  The othread_self() function returns the Othread handle of the calling thread. The othread_self()
00083  function does NOT return the integral thread of the calling thread. You must use
00084  othread_getthreadid_np() to return an integral identifier for the thread.
00085 */
00086 othread_t othread_self(void);
00087
00088 /** Set Process Concurrency Level.
00089  The othread_setconcurrency() function sets the current concurrency level for the process.
00090  A concurrency value of zero indicates that the threads implementation chooses the concurrency
00091  level that best suits the application. A concurrency level greater than zero indicates that the
00092  application wants to inform the system of its desired concurrency level.
00093  Returns 0 if successful.
00094 */
00095 int othread_setconcurrency(
00096  int concurrency);                          /**< (In) The new concurrency level for the process */
00097
00098 /** Set Thread Local Storage by Key.
00099  The othread_setspecific() function sets the thread local storage value associated with a key. The
00100  othread_setspecific() function may be called from within a data destructor.
00101 */
00102 int othread_setspecific(
00103  othread_key_t key,     /**< (In) The thread local storage key returned from othread_key_create(). */
00104  const void *value);    /**< (In) The pointer to store at the key location for the calling thread. */
00105
00106 /* S I M P L E   L O C K S */
00107
00108 /**
00109  The othread_init_lock function initializes the simple lock associated with the parameter lock for
00110  use in subsequent calls.
00111  */
00112 int othread_init_lock(
00113  othread_lock_t *lock);          /**< (In) The address of the variable to contain a lock object. */
00114
00115 /**
00116  The othread_destroy_lock function ensures that the pointed to lock variable lock is uninitialized.
00117  */
00118 int othread_destroy_lock(
00119  othread_lock_t *lock);                      /**< (In) Address of the lock to be destroyed */
00120
00121 /**
00122  The othread_set_lock function blocks the thread executing the function until the specified lock is
00123  available and then sets the look. A simple lock is available if it is unlocked.
00124  */
00125 int othread_set_lock(
00126  othread_lock_t *lock);                        /**< (In) The address of the look to set */
00127
00128 /**
00129  The othread_unset_lock function provide the means of releasing ownership of a simple lock.
00130  */
00131 int othread_unset_lock(
00132  othread_lock_t *lock);                        /**< (In) Address of the lock to unset */
00133
00134 /**
00135  The othread_test_lock function attempts to set a lock but does not block execution of the thread.
00136  Returns a nonzero value if the lock is successfully set; otherwise, it returns zero.
00137  */
00138 int othread_test_lock(
00139  othread_lock_t *lock);                        /**< (In) Address of the lock to set */
00140
00141 /* N E S T E D   L O C K S */
00142
00143 /**
00144  The othread_init_nest_lock function initializes the nested lock associated with the parameter lock
```

```
00145  for use in subsequent calls.
00146  */
00147 int othread_init_nest_lock(
00148  othread_nest_lock_t *lock);       /**< (In) The address of the variable to contain a nested lock */
00149
00150 /**
00151  The othread_destroy_nest_lock function ensures that the pointed to lock variable lock is
00152  uninitialized.
00153  */
00154 int othread_destroy_nest_lock(
00155  othread_nest_lock_t *lock);                          /**< (In) Address of the lock to be destroyed */
00156
00157 /**
00158  The othread_set_nest_lock function blocks the thread executing the function until the specified
00159  lock is available and then sets the lock. A nestable lock is available if it is unlocked or if it
00160  is already owned by the thread executing the function.
00161  */
00162 int othread_set_nest_lock(
00163  othread_nest_lock_t *lock);                            /**< (In) The address of the lock to set */
00164
00165 /**
00166  The othread_unset_nest_lock function provide the means of releasing ownership of a nestable lock.
00167  */
00168 int othread_unset_nest_lock(
00169  othread_nest_lock_t *lock);                             /**< (In) Address of the lock to unset */
00170
00171 /**
00172  The othread_test_nest_lock function attempts to set a lock but does not block execution of the
00173  thread. Returns the new nesting count if the lock is successfully set; otherwise, it returns zero.
00174  */
00175 int othread_test_nest_lock(
00176  othread_nest_lock_t *lock);                               /**< (In) Address of the lock to set */
00177
00178 /* S P I N   L O C K S */
00179
00180 /**
00181  The othread_init_spin_lock function initializes the spni lock associated with the parameter lock
00182  for use in subsequent calls.
00183  */
00184 int othread_init_spin_lock(
00185  othread_spin_lock_t *lock);        /**< (In) The address of the variable to contain a spin lock */
00186
00187 /**
00188  The othread_destroy_spin_lock function ensures that the pointed to lock variable lock is
00189  uninitialized.
00190  */
00191 int othread_destroy_spin_lock(
00192  othread_spin_lock_t *lock);                          /**< (In) Address of the lock to be destroyed */
00193
00194 /**
00195  The othread_set_spin_lock function blocks the thread executing the function until the specified
00196  lock is available and then sets the lock. A spin lock is available if it is unlocked.
00197  */
00198 int othread_set_spin_lock(
00199  othread_spin_lock_t *lock);                            /**< (In) The address of the lock to set */
00200
00201 /**
00202  The othread_unset_spin_lock function provide the means of releasing ownership of a spin lock.
00203  */
00204 int othread_unset_spin_lock(
00205  othread_spin_lock_t *lock);                               /**< (In) Address of the lock to
unset */
00206
00207 /**
00208  The othread_test_spin_lock function attempts to set a lock but does not block execution of the
00209  thread. Returns a nonzero value if the lock is successfully set; otherwise, it returns zero.
00210  */
00211 int othread_test_spin_lock(
00212  othread_spin_lock_t *lock);                                /**< (In) Address of the lock to
set */
00213
00214 #endif
```

ompi-0.8.2/lib/sparc_othread.c

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*! \file sparc_othread.c
00024     \brief The wrapper function implementations for solaris threads.
00025  */
00026
00027 #include <othread.h>
00028 #include <thread.h>
00029
00030 /** Create Thread.
00031  The othread_create() function creates a thread with the specified attributes and runs the C
00032  function start_routine in the thread with the single pointer argument specified. The new thread
00033  may, but does not always, begin running before othread_create() returns. If othread_create()
00034  completes successfully, the othread handle is stored in the contents of the location referred to
00035  by thread. Returns 0 if successful.
00036 */
00037 int othread_create(
00038  othread_t *thread,                              /**< (Out) othread handle to the created thread */
00039  const othread_attr_t *attr,     /**< (In) The thread attributes object containing the attributes to
00040     be associated with the newly created thread. If NULL, the default thread attributes are used. */
00041  void *(*start_routine)(void *),/**< (In) The function to be run as the new threads start routine */
00042  void *arg)                    /**< (In) An address for the argument for the threads start routine */
00043 {
00044   return thr_create(
00045     NULL,                                                                  /* stack_base */
00046     0,                                                                     /* stack_size */
00047     start_routine,
00048     arg,
00049     0,                                              /* TODO: flags do not match but are unused */
00050     thread);
00051 }
00052
00053 /** Compare Two Threads.
00054  The othread_equal() function compares two othread handles for equality.
00055  Returns 1 if the othread handles refer to the same thread, 0 otherwise.
00056 */
00057 int othread_equal(
00058  othread_t t1,                                          /**< (In) othread handle for thread 1 */
00059  othread_t t2)                                          /**< (In) othread handle for thread 2 */
00060 {
00061   return t1 == t2;                                /**< There is no thr_equal in Solaris threads */
00062 }
00063
00064 /** Get Thread Local Storage Value by Key.
00065  The othread_getspecific() function retrieves the thread local storage value associated with the
00066  key. othread_getspecific() may be called from a data destructor. The thread local storage value is
00067  a variable of type void * that is local to a thread, but global to all of the functions called
00068  within that thread. It is accessed via the key. Returns NULL if key out of range.
00069 */
00070 void *othread_getspecific(
00071  othread_key_t key)     /**< (In) The thread local storage key returned from othread_key_create() */
00072 {
00073   void *result;
00074
00075   thr_getspecific(key, &result);                    /* Returns int (error code), not the pointer */
```

```
00076
00077   return result;
00078 }
00079
00080 /** Create Thread Local Storage Key.
00081  The othread_key_create() function creates a thread local storage key for the process and associates
00082  the destructor function with that key. After a key is created, that key can be used to set and get
00083  per-thread data pointer. When othread_key_create() completes, the value associated with the newly
00084  created key is NULL. Returns 0 if successful.
00085 */
00086 int othread_key_create(
00087  othread_key_t *key,/**<(Out) The address of the variable to contain the thread local storage key */
00088  void (*destructor)(void *))     /**< (In) The address of the function to act as a destructor for */
00089 {                                                                    /* this thread local storage key */
00090   return thr_keycreate(key, destructor);              /* TODO: descructor unused, can be removed */
00091 }
00092
00093 /** Get othread Handle.
00094  The othread_self() function returns the othread handle of the calling thread. The othread_self()
00095  function does NOT return the integral thread of the calling thread. You must use
00096  othread_getthreadid_np() to return an integral identifier for the thread.
00097 */
00098 othread_t othread_self(void)
00099 {
00100   return thr_self();
00101 }
00102
00103 /** Set Process Concurrency Level.
00104  The othread_setconcurrency() function sets the current concurrency level for the process.
00105  A concurrency value of zero indicates that the threads implementation chooses the concurrency
00106  level that best suits the application. A concurrency level greater than zero indicates that the
00107  application wants to inform the system of its desired concurrency level.
00108  Returns 0 if successful.
00109 */
00110 int othread_setconcurrency(
00111  int concurrency)                                /**< (In) The new concurrency level for the process */
00112 {
00113   return thr_setconcurrency(concurrency);/* actually it does nothing, all threads attached in LWP */
00114 }
00115
00116 /** Set Thread Local Storage by Key.
00117  The othread_setspecific() function sets the thread local storage value associated with a key. The
00118  othread_setspecific() function may be called from within a data destructor.
00119 */
00120 int othread_setspecific(
00121  othread_key_t key,     /**< (In) The thread local storage key returned from othread_key_create(). */
00122  const void *value)     /**< (In) The pointer to store at the key location for the calling thread. */
00123 {
00124   return thr_setspecific((thread_key_t)key, (void *)value);
00125 }
00126
00127 /* S I M P L E   L O C K S */
00128
00129 /**
00130  The othread_init_lock function initializes the simple lock associated with the parameter lock for
00131  use in subsequent calls.
00132  */
00133 int othread_init_lock(
00134  othread_lock_t *lock)              /**< (In) The address of the variable to contain a lock object. */
00135 {
00136   return mutex_init(&lock->mutex,
00137   USYNC_THREAD,               /* The lock can be used to synchronize threads in this process only */
00138   NULL);                                                          /* void *arg, not currently used */
00139 }
00140
00141 /**
00142  The othread_destroy_lock function ensures that the pointed to lock variable lock is uninitialized.
00143  */
00144 int othread_destroy_lock(
00145  othread_lock_t *lock)                                 /**< (In) Address of the lock to be destroyed */
00146 {
00147   return mutex_destroy(&lock->mutex);
00148 }
00149
00150 /**
00151  The othread_set_lock function blocks the thread executing the function until the specified lock is
00152  available and then sets the lock. A simple lock is available if it is unlocked.
```

```
00153  */
00154 int othread_set_lock(
00155  othread_lock_t *lock)                                  /**< (In) The address of the lock to set */
00156 {
00157   return mutex_lock(&lock->mutex);
00158 }
00159
00160 /**
00161  The othread_unset_lock function provide the means of releasing ownership of a simple lock.
00162  */
00163 int othread_unset_lock(
00164  othread_lock_t *lock)                                   /**< (In) Address of the lock to unset */
00165 {
00166   return mutex_unlock(&lock->mutex);
00167 }
00168
00169 /**
00170  The othread_test_lock function attempts to set a lock but does not block execution of the thread.
00171  Returns a nonzero value if the lock is successfully set; otherwise, it returns zero.
00172  */
00173 int othread_test_lock(
00174  othread_lock_t *lock)                                   /**< (In) Address of the lock to set */
00175 {
00176   return mutex_trylock(&lock->mutex);
00177 }
00178
00179 /* N E S T E D   L O C K S */
00180
00181 /**
00182  The othread_init_nest_lock function initializes the nested lock associated with the parameter lock
00183  for use in subsequent calls.
00184  */
00185 int othread_init_nest_lock(
00186  othread_nest_lock_t *lock)        /**< (In) The address of the variable to contain a nested lock */
00187 {
00188   mutex_init(&lock->ilock, USYNC_THREAD, NULL);                          /* initialize fields */
00189 /* mutex_lock(lock->ilock); */
00190   mutex_init(&lock->lock, USYNC_THREAD, NULL);
00191   lock->count = 0;
00192   cond_init(&lock->cond, USYNC_THREAD, NULL);          /* Synchonize threads in this process only */
00193 /* mutex_unlock(&lock->ilock); */
00194
00195   return 0;
00196 }
00197
00198 /**
00199  The othread_destroy_nest_lock function ensures that the pointed to lock variable lock is
00200  uninitialized.
00201  */
00202 int othread_destroy_nest_lock(
00203  othread_nest_lock_t *lock)                          /**< (In) Address of the lock to be destroyed */
00204 {
00205 /* mutex_lock(&lock->ilock); */
00206   mutex_destroy(&lock->lock);                                    /* destroy the mutex fields */
00207   cond_destroy(&lock->cond);
00208 /* mutex_unlock(&lock->ilock); */
00209   mutex_destroy(&lock->ilock);
00210 }
00211
00212 /**
00213  The othread_set_nest_lock function blocks the thread executing the function until the specified
00214  lock is available and then sets the lock. A nestable lock is available if it is unlocked or if it
00215  is already owned by the thread executing the function.
00216  */
00217 int othread_set_nest_lock(
00218  othread_nest_lock_t *lock)                             /**< (In) The address of the lock to set */
00219 {
00220   mutex_lock(&lock->ilock);                                /* get exclusive access to the struct */
00221   if (mutex_trylock(&lock->lock)==0) {                                  /* if not already locked */
00222     lock->owner = othread_self();                                                /* lock it */
00223     lock->count++;
00224   }                                              /* else check if already locked by the same thread */
00225   else if (othread_equal(lock->owner, othread_self()))
00226     lock->count++;                                                 /* if yes, just increase counter */
00227   else {                                      /* else it's locked by another thread, we have to wait */
00228     while (mutex_trylock(&lock->lock) != 0)
00229       cond_wait(&lock->cond, &lock->ilock);
```

```
00230     lock->owner = othread_self();                          /* done waiting, now lock it */
00231     lock->count++;
00232   }
00233   mutex_unlock(&lock->ilock);                              /* unlock the struct access lock */
00234 }
00235
00236 /**
00237  The othread_unset_nest_lock function provide the means of releasing ownership of a nestable lock.
00238  */
00239 int othread_unset_nest_lock(
00240  othread_nest_lock_t *lock)                          /**< (In) Address of the lock to unset */
00241 {
00242   mutex_lock(&lock->ilock);                           /* get exclusive access to the struct */
00243   if (othread_equal(lock->owner, othread_self())         /* if locked by the calling thread */
00244    && lock->count > 0) {                                                  /* and count > 0 */
00245     lock->count--;
00246     if (lock->count == 0) {                                            /* if count reaches zero */
00247       mutex_unlock(&lock->lock);                                      /* then completely unlock */
00248       cond_signal(&lock->cond);                                        /* and signal condition */
00249     }                                        /* because someone might be waiting to set the lock */
00250   }
00251   mutex_unlock(&lock->ilock);                              /* unlock the struct access lock */
00252 }
00253
00254 /**
00255  The othread_test_nest_lock function attempts to set a lock but does not block execution of the
00256  thread. Returns the new nesting count if the lock is successfully set; otherwise, it returns zero.
00257  */
00258 int othread_test_nest_lock(
00259  othread_nest_lock_t *lock)                           /**< (In) Address of the lock to set */
00260 {
00261   int res;
00262
00263   mutex_lock(&lock->ilock);                           /* get exclusive access to the struct */
00264   if (mutex_trylock(&lock->lock)==0) {                        /* if not already locked */
00265     lock->owner = othread_self();                                                 /* lock it */
00266     res = ++lock->count;                                                        /* return 1 */
00267   }
00268   else                                          /* else check if already locked by the same thread */
00269     if (othread_equal(lock->owner, othread_self()))
00270       res = ++lock->count;                                 /* if yes, return the nesting counter */
00271     else
00272       res = 0;
00273   mutex_unlock(&lock->ilock);                              /* unlock the struct access lock */
00274
00275   return res;
00276 }
00277
00278 /* S P I N   L O C K S */
00279
00280 /**
00281  The othread_init_spin_lock function initializes the spin lock associated with the parameter lock
00282  for use in subsequent calls.
00283  */
00284 int othread_init_spin_lock(
00285  othread_spin_lock_t *lock)          /**< (In) The address of the variable to contain a spin lock */
00286 {
00287   return mutex_init(&lock->mutex,
00288   USYNC_THREAD,                /* The lock can be used to synchronize threads in this process only */
00289   NULL);                                                          /* void *arg, not currently used */
00290 }
00291
00292 /**
00293  The othread_destroy_spin_lock function ensures that the pointed to lock variable lock is
00294  uninitialized.
00295  */
00296 int othread_destroy_spin_lock(
00297  othread_spin_lock_t *lock)                        /**< (In) Address of the lock to be destroyed */
00298 {
00299   return mutex_destroy(&lock->mutex);
00300 }
00301
00302 /**
00303  The othread_set_spin_lock function blocks the thread executing the function until the specified
00304  lock is available and then sets the lock. A spin lock is available if it is unlocked.
00305  */
00306 int othread_set_spin_lock(
```

```
00307  othread_spin_lock_t *lock)                          /**< (In) The address of the lock to set */
00308 {
00309   return mutex_lock(&lock->mutex);
00310 }
00311
00312 /**
00313  The othread_unset_spin_lock function provide the means of releasing ownership of a spin lock.
00314  */
00315 int othread_unset_spin_lock(
00316  othread_spin_lock_t *lock)                          /**< (In) Address of the lock to unset */
00317 {
00318   return mutex_unlock(&lock->mutex);
00319 }
00320
00321 /**
00322  The othread_test_spin_lock function attempts to set a lock but does not block execution of the
00323  thread. Returns a nonzero value if the lock is successfully set; otherwise, it returns zero.
00324  */
00325 int othread_test_spin_lock(
00326  othread_spin_lock_t *lock)                          /**< (In) Address of the lock to set */
00327 {
00328   return mutex_trylock(&lock->mutex);
00329 }
```

ompi-0.8.2/lib/sparc_othread_types.h

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006  This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*! \file othread_types.h
00024     \brief Thread types. Implementation for the sparc thread library.
00025  */
00026
00027 #ifndef __SPARC__OTHREAD_TYPES_H__
00028 #define __SPARC__OTHREAD_TYPES_H__
00029
00030 #include <thread.h>
00031
00032 #define OTHREAD_LOCK_INITIALIZER {DEFAULTMUTEX}
00033
00034 typedef thread_t othread_t;
00035 typedef long othread_attr_t;            /* corresponds to parameter "flags" of thr_create, unused */
00036 typedef thread_key_t othread_key_t;
00037
00038 /** othread_mutex_t is equivelant to mutex_t, except for its size. It is declared as a
00039  64-byte union to ensure that TWO locks can't fit in the same cache line. If two or more locks
00040  happen to be in the same cache line, then polling gets their value from RAM instead of cache,
00041  causing high bus traffic.
00042  In Solaris threads, mutex_t is already padded to 64 bytes, so there is no size loss.
00043  DEVELOPERS NOTE: unions are initialized based on their FIRST declared member, i.e. mutex.
00044  In order for the OTHREAD_LOCK_INITIALIZER to work, the order of members should not change.
00045  Nevertheless, in Solaris threads, padding is declared first, because no THREAD_MUTEX_INITIALIZER
00046  is defined, and initialization is (according to the manuals) done by zero-filled memory.
00047  */
00048 typedef union {
00049   mutex_t mutex;
```

```
00050   char padding[64];                                   /**< 64 bytes is >= than most cache line sizes */
00051 } othread_lock_t;
00052
00053 /* TODO: check: in SunOS >= 5.9 recursive mutexes work ok, but not in 5.8.
00054    So it would be better to use #if os version... */
00055 /**
00056  The othread_nest_lock_t is an internal type, the public type omp_nest_lock_t is declared in omp.h.
00057  A nested lock is an object type capable of represending either that a lock is available, or both
00058  the identity of the thread that owns the lock and a nesting count.
00059  */
00060 typedef struct {
00061   mutex_t lock;                                              /**< The corresponding simple lock */
00062   othread_t owner;                                       /**< Which thread owns the nestable lock */
00063   int count;                                        /**< How many times it is locked by the same thread */
00064   cond_t cond;                                   /**< Used to conditionally wait until lock is available */
00065   mutex_t ilock;                                                    /**< Struct access lock */
00066 } othread_nest_lock_t;
00067
00068 /**
00069  othread_spin_lock_t locks are faster than othread_lock_t but they keep the processor busy.
00070  They are used in cases where something must be locked for a very small period of time.
00071  (For now they are implemented as simple othread_locks, except for IRIX)
00072  */
00073 typedef othread_lock_t othread_spin_lock_t;
00074
00075 #endif
```

ompi-0.8.2/ompi/Makefile.am

```
00001 #
00002 #    OMPi OpenMP Compiler
00003 #    Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas
00004 #    Alkis Georgopoulos
00005 #
00006 #    This file is part of OMPi.
00007 #
00008 #    OMPi is free software; you can redistribute it and/or modify
00009 #    it under the terms of the GNU General Public License as published by
00010 #    the Free Software Foundation; either version 2 of the License, or
00011 #    (at your option) any later version.
00012 #
00013 #    OMPi is distributed in the hope that it will be useful,
00014 #    but WITHOUT ANY WARRANTY; without even the implied warranty of
00015 #    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016 #    GNU General Public License for more details.
00017 #
00018 #    You should have received a copy of the GNU General Public License
00019 #    along with OMPi; if not, write to the Free Software
00020 #    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 #
00022 ##   This file must be processed by automake
00023
00024 bin_PROGRAMS = ompi ompicc
00025 AM_CFLAGS = @DEBUGFLAG@ -I. -DSYSCOMPILER=\"@CC@\" -DINCLUDEDIR=\"@includedir@\" -
DLIBDIR=\"@libdir@\" @DEFINETHREADLIB@
00026 AM_YFLAGS = -d
00027 #AM_YFLAGS = -d -v --debug
00028
00029 ompi_SOURCES = parser.y scanner.l
00030 EXTRA_ompi_SOURCES = parser.h
00031 ompicc_SOURCES = ompicc.c
00032
00033 #CLEANFILES = parser.output parser.h parser.c scanner.c
```

ompi-0.8.2/ompi/ompicc.c

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
```

```
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*!
00024  \file ompicc.c
00025  \brief The ompi compiler command line interface.
00026  Usage: ompicc [-k] [--pomp-enable] [--pomp-disable=...] <gcc-args>
00027    -k: keep intermediate files
00028    --pomp-enable: enable POMP instrumentation\n"
00029    --pomp-disable: disable specific POMP regions\n"
00030  1. The input files are passed to the preprocessor (gcc -e).
00031  2. Then ompi.c transforms #pragma omp ...
00032  3. The output files are concatenated and passed to gcc.
00033 */
00034
00035 #include <stdio.h>
00036 #include <stdlib.h>
00037 #include <string.h>
00038 #include <ctype.h>
00039 #include <sys/types.h>
00040 #include <sys/wait.h>
00041 #include <sys/stat.h>
00042 #include <unistd.h>
00043
00044 #include "config.h"
00045
00046 #define LEN 4096
00047 #define SLEN 256
00048
00049 /* The following variables are used to REMEMBER which thread library the user selected during
00050    ./configure and pass it on to
00051    1) The compiler, as #define, so that othread.h can select the appropriate othread_types.h, and
00052    2) The linker, to link the selected library with the user executable
00053  */
00054 #ifdef SOLARISTHREADS
00055   #warning "OMPi INFO: Compiling ompicc with Solaris threads library..."
00056   char *threadlib="thread";
00057   char *threadlib_define="-DSOLARISTHREADS";
00058 #else
00059   #warning "OMPi INFO: Compiling ompicc with pthread library..."
00060   char *threadlib="pthread";
00061   char *threadlib_define="";
00062 #endif
00063
00064 typedef struct arg_s {
00065   char opt;
00066   char val[SLEN];
00067   struct arg_s *next;
00068 } arg_t;
00069
00070 typedef struct {
00071    arg_t *head, *tail;
00072 } arglist_t;
00073
00074 char *get_basename(char *path)
00075 {
00076   char *s;
00077
00078   if (path == NULL || *path == 0)
00079     return ".";
00080   else if (path[0] == '/' && path[1] == 0)
00081     return path;
00082   else {
00083     s = path;
00084     while (*s)
00085       s++;
```

```
00086     s--;
00087     while (*s == '/' && s != path)
00088       *s-- = 0;
00089     if (s == path)
00090       return path;
00091     while (*s != '/' && s != path)
00092       s--;
00093     if (*s == '/')
00094       return s + 1;
00095     else return path;
00096   }
00097 }
00098
00099 char *get_dirname(char *path)
00100 {
00101   char *s;
00102
00103   if (path == NULL || *path == 0)
00104     return ".";
00105   else if (path[0] == '/' && path[1] == 0)
00106     return path;
00107   else {
00108     s = path;
00109     while (*s)
00110       s++;
00111     s--;
00112     while (*s == '/' && s != path)
00113       *s-- = 0;
00114     if (s == path)
00115       return "/";
00116     while (*s != '/' && s != path)
00117       s--;
00118     if (*s == '/') {
00119       if (s != path)
00120         *s = 0;
00121       else return "/";
00122     }
00123     else return ".";
00124     return path;
00125   }
00126 }
00127
00128 arg_t *new_arg(char opt, char *val)
00129 {
00130   arg_t *p;
00131
00132   p = (arg_t *) malloc(sizeof(arg_t));
00133   if (p == NULL) {
00134     fprintf(stderr, "malloc(): out of memory\n");
00135     _exit(-1);
00136   }
00137   p->opt = opt;
00138   if (val != NULL)
00139     strcpy(p->val, val);
00140   else p->val[0] = 0;
00141   p->next = NULL;
00142   return p;
00143 }
00144
00145 void arglist_init(arglist_t *l)
00146 {
00147   l->head = l->tail = NULL;
00148 }
00149
00150 void arglist_add(arglist_t *l, arg_t *arg)
00151 {
00152   if (l->head == NULL)
00153     l->head = l->tail = arg;
00154   else {
00155     l->tail->next = arg;
00156     l->tail = arg;
00157   }
00158 }
00159
00160 int append_arg(arglist_t *l, int argc, char **argv, int proceed)
00161 {
00162   char opt, val[SLEN];
```

```
00163   arg_t *p;
00164
00165   val[0] = 0;
00166   if (argv[0][0] == 0)
00167     return 0;
00168   if (argv[0][0] != '-') {
00169     p = new_arg(0, *argv);
00170     arglist_add(l, p);
00171     return 0;
00172   }
00173   opt = argv[0][1];
00174
00175   if (argv[0][2] != 0) {
00176     strcpy(val, &argv[0][2]);
00177     p = new_arg(opt, val);
00178     arglist_add(l, p);
00179     return 0;
00180   }
00181   else {
00182     if (proceed && argc > 1)
00183       strcpy(val, &argv[1][0]);
00184     p = new_arg(opt, val);
00185     arglist_add(l, p);
00186     return proceed && argc > 1;
00187   }
00188 }
00189
00190 void strarglist(char *dest, arglist_t *l)
00191 {
00192   arg_t *p;
00193   *dest = 0;
00194
00195   for (p = l->head; p != NULL; p = p->next) {
00196     if (p->opt != 0) {
00197       sprintf(dest, "-%c ", p->opt);
00198       if (p->opt == 'o')
00199         dest += 3;
00200       else dest += 2;
00201     }
00202     sprintf(dest, "%s ", p->val);
00203     dest += strlen(p->val) + 1;
00204   }
00205 }
00206
00207 int fok(char *fname)
00208 {
00209   struct stat buf;
00210
00211   return stat(fname, &buf) == 0;
00212 }
00213
00214 int linking = 1;                                  /**< Becomes 0 if -c parameter is specified */
00215 int keep = 0;                                     /**< Becomes 1 if -k parameter is specified */
00216 /**< Collection of bits, enables/disables some instrumentation.
00217  But always using POMP is annoying. So we defined an unofficial extra parameter "--pomp-enable" to
00218  explicitly enable POMP. So if a user wants to enable POMP but disable barrier monitoring he should
00219  specify --pomp-enable --pomp-disable=barrier */
00220 int pomp_disable_mask = 0;
00221
00222 arglist_t files;                                  /**< Files to be compiled/linked */
00223 arglist_t goutfile;                               /**< Output, -o XXX */
00224 arglist_t prep_args;                              /**< Preprocessor args */
00225 arglist_t link_args;                              /**< Linker args */
00226 arglist_t gcc_args;                               /**< Remaining compiler args */
00227
00228 char curdir[SLEN];
00229
00230 void parse_args(int argc, char **argv)
00231 {
00232   int c, d, i;
00233   char tmp[SLEN];
00234   char *parameter;
00235   static char *pomp_enable = "--pomp-enable";
00236   static char *pomp_disable = "--pomp-disable=";
00237   static char *pomp_region_name[11] = {
00238     "atomic", "barrier", "critical", "for", "master",
00239     "parallel", "region", "section", "sections", "single", "sync"};
```

```
00240   static int pomp_region_mask[11] = {
00241     1, 2, 4, 8, 16, 32, 64, 128, 256, 512, -1};        /* sync blocks ALL performance monitoring */
00242   int pomp_enabled = 0;                                  /* POMP is initially disabled */
00243
00244   getcwd(curdir, SLEN);
00245
00246   arglist_init(&files);
00247   arglist_init(&goutfile);
00248   arglist_init(&prep_args);
00249   arglist_init(&link_args);
00250   arglist_init(&gcc_args);
00251   argv++;
00252   argc--;
00253
00254   while (argc) {
00255     d = 0;
00256     parameter = argv[0];
00257     if (strncmp(parameter, pomp_enable, strlen(pomp_enable)) == 0) {
00258       pomp_enabled = 1;
00259     }
00260     else if (strncmp(parameter, pomp_disable, strlen(pomp_disable)) == 0) {
00261       parameter += strlen(pomp_disable);                  /* pass the "--pomp-disable=" part */
00262       do {
00263         for (i = 0; i < 11; i++)
00264           if (strncmp(parameter, pomp_region_name[i], strlen(pomp_region_name[i])) == 0) {
00265             pomp_disable_mask |= pomp_region_mask[i];                    /* set the mask */
00266             parameter += strlen(pomp_region_name[i]);
00267             if (*parameter == ',')                    /* e.g. --pomp-disable=atomic,region */
00268               parameter++;
00269             break;                                                      /* exit for */
00270           }
00271       } while (i < 11 && *parameter != '\0');        /* if a match was found, try again for more */
00272     }
00273     else if (argv[0][0] == '-')                                         /* option */
00274       switch (argv[0][1]) {
00275         case 'c': linking = 0; break;
00276         case 'l': d = append_arg(&link_args, argc, argv, 1); break;
00277         case 'L': d = append_arg(&link_args, argc, argv, 1); break;
00278         case 'I': d = append_arg(&prep_args, argc, argv, 1); break;
00279         case 'D': d = append_arg(&prep_args, argc, argv, 1); break;
00280         case 'U': d = append_arg(&prep_args, argc, argv, 1); break;
00281         case 'o': d = append_arg(&goutfile, argc, argv, 1); break;
00282         case 'k': keep = 1; d = 0; break;
00283         default:
00284           d = append_arg(&gcc_args, argc, argv, 0);
00285           if (strcmp(argv[0], "--version") == 0) {
00286             printf("%s ", VERSION);
00287             fflush(stdout);
00288             if (strcmp(SYSCOMPILER, "gcc") == 0)
00289               system("gcc --version");
00290             else printf("\n");
00291             _exit(0);
00292           }
00293       }
00294     else {
00295       d = append_arg(&files, argc, argv, 0);
00296       if (!fok(files.tail->val)) {
00297         fprintf(stderr, "file %s does not exist\n", files.tail->val);
00298         _exit(1);
00299       }
00300     }
00301
00302     argc = argc - 1 - d;
00303     argv = argv + 1 + d;
00304   }
00305   if (!pomp_enabled)          /* If the user didn't explicitly enable pomp, disable it completely */
00306     pomp_disable_mask = -1;
00307 }
00308
00309 char *fext_fun(char *fname)
00310 {
00311   char *s;
00312
00313   s = strrchr(fname, '.');
00314   if (s == NULL)
00315     return "";
00316   else return s + 1;
```

```
00317 }
00318
00319 void fzapext(char *fname)
00320 {
00321   char *s;
00322
00323   s = strrchr(fname, '.');
00324   if (s != NULL) *s = 0;
00325 }
00326
00327 void process_file(char *fname)
00328 {
00329   char tmp[SLEN], bfile[SLEN], filedir[SLEN], curdir2[SLEN];
00330   char outfile[SLEN], outdir[SLEN];
00331   char cmd[LEN], strgcc_args[LEN], strgoutfile[LEN], rm_outdir[LEN];
00332   char tmp2[SLEN];
00333   int samedir;
00334   int res;
00335   FILE *f;
00336
00337   if (strcmp(fext_fun(fname), "o") == 0)                                    /* skip .o */
00338     return;
00339
00340   strcpy(tmp, fname);
00341   strcpy(bfile, get_basename(tmp));
00342   fzapext(bfile);
00343
00344   strcpy(tmp, fname);
00345   strcpy(filedir, get_dirname(tmp));
00346
00347   sprintf(outfile, "%s_ompi.c", bfile);
00348   sprintf(outdir, "%s_ompi", bfile);
00349
00350   sprintf(rm_outdir, "rm -rf %s", outdir);
00351
00352   chdir(filedir);
00353   getcwd(curdir2, SLEN);
00354   samedir = strcmp(curdir2, curdir) == 0;
00355   mkdir(outdir, 0755);
00356
00357   /* preprocess */
00358   strarglist(tmp2, &prep_args);
00359   sprintf(cmd, "%s -E -U__GNUC__ -D_OPENMP=200203 -D_POMP=200203 %s -I%s %s %s.c >> %s/%s.c",
00360       SYSCOMPILER, threadlib_define, INCLUDEDIR, tmp2, bfile, outdir, bfile);
00361   res = system(cmd);
00362   if (res > 0) {
00363     if (!keep)
00364       system(rm_outdir);
00365     chdir(curdir);
00366     _exit(res);
00367   }
00368   chdir(outdir);
00369
00370   /* transform */
00371 #ifdef DEBUGGING
00372   fprintf(stderr, "pomp_disable_mask == %d\n", pomp_disable_mask);
00373 #endif
00374   sprintf(cmd, "ompi %s.c __ompi__ %d > ../%s", bfile, pomp_disable_mask, outfile);
00375   res = system(cmd);
00376   res = WEXITSTATUS(res);
00377   chdir("..");
00378   if (!keep)
00379     system(rm_outdir);
00380   if (res == 33) {                                          /* no pragma omp directives */
00381     sprintf(cmd, "cp %s.c %s", bfile, outfile);
00382     system(cmd);
00383   }
00384   else if (res > 0) {
00385     if (!keep)
00386       unlink(outfile);
00387     chdir(curdir);
00388     _exit(res);
00389   }
00390
00391   /* call indent here */
00392
00393   chdir(curdir);
```

```
00394
00395   strarglist(strgcc_args, &gcc_args);
00396   sprintf(cmd, "%s %s/%s -c -D_OPENMP=200203 -D_POMP=200203 %s -I%s %s %s",
00397     SYSCOMPILER, filedir, outfile, threadlib_define, INCLUDEDIR, tmp2, strgcc_args);
00398   res = system(cmd);
00399
00400   if (!keep) {
00401     chdir(filedir);
00402     unlink(outfile);
00403     chdir(curdir);
00404   }
00405
00406   if (res > 0)
00407     _exit(res);
00408
00409   strarglist(strgoutfile, &goutfile);
00410
00411   if (goutfile.head != NULL && linking == 0) {
00412     sprintf(cmd, "mv %s_ompi.o %s", bfile, goutfile.head->val);
00413     system(cmd);
00414   }
00415   else {
00416     sprintf(cmd, "mv %s_ompi.o %s.o", bfile, bfile);
00417     system(cmd);
00418   }
00419 }
00420
00421 void do_linking()
00422 {
00423   arg_t *p;
00424   char cur_obj[SLEN], tmp[SLEN], cmd[LEN];
00425   char objects[LEN], *obj;
00426   char strgccargs[LEN], strlinkargs[LEN], strgoutfile[LEN], strprepargs[LEN];
00427   int res, len;
00428   int is_tmp;
00429   char rm_obj[LEN];
00430   char module_id[SLEN];
00431   int num_for, num_single, num_sections, num_reduction, g_threadprivate;
00432   FILE *linkh_fp, *link_fp, *rtp_fp;
00433
00434   linkh_fp = fopen("_omp_modinit.h", "w");
00435   fprintf(linkh_fp, "#include <othread.h>\n"
00436                     "#include <ompi.h>\n"
00437                     "#include <pomp_lib.h>\n");
00438
00439   link_fp = fopen("_omp_modinit.c", "w");
00440   fprintf(link_fp, "#include\"_omp_modinit.h\"\n\n"
00441                    "void _omp_modules_initialize()\n"
00442                    "{\n");
00443
00444   rtp_fp = fopen("_omp_resztpv.c", "w");
00445   fprintf(rtp_fp,  "void _omp_modules_resize_tp_vars(int newlen)\n"
00446                    "{\n");
00447
00448   obj = objects;
00449   *obj = 0;
00450   strcpy(rm_obj, "rm -f ");
00451   for (p = files.head; p != NULL; p = p->next) {
00452     strcpy(cur_obj, p->val);
00453
00454     is_tmp = 0;
00455     len = strlen(cur_obj);
00456     if (cur_obj[len-1] == 'c')
00457       is_tmp = 1;
00458
00459     if (is_tmp) {
00460       cur_obj[len-2] = 0;
00461       strcpy(tmp, cur_obj);
00462       strcpy(cur_obj, get_basename(tmp));
00463       strcat(cur_obj, ".o");
00464       strcat(rm_obj, cur_obj);
00465       strcat(rm_obj, " ");
00466     }
00467
00468     sprintf(obj, "%s ", cur_obj);
00469     obj += strlen(cur_obj)+1;
00470
```

```
00471     {
00472       FILE *f;
00473       char pstr[LEN], crit[LEN], *s, *e;
00474
00475       fflush(NULL);
00476       module_id[0] = 0;
00477       sprintf(cmd, "nm %s | grep _omp", cur_obj);
00478       f = popen(cmd, "r");
00479       while (fgets(pstr, LEN, f) != NULL) {
00480         if ((s = strstr(pstr, "_omp_mi_id_")) != NULL) {
00481           e = strchr(s+11, '_'); *e = 0;
00482           strcpy(module_id, s+11);
00483         }
00484         else if ((s = strstr(pstr, "_omp_mi_for_")) != NULL)
00485           sscanf(s+12, "%d", &num_for);
00486         else if ((s = strstr(pstr, "_omp_mi_sec_")) != NULL)
00487           sscanf(s+12, "%d", &num_sections);
00488         else if ((s = strstr(pstr, "_omp_mi_sin_")) != NULL)
00489           sscanf(s+12, "%d", &num_single);
00490         else if ((s = strstr(pstr, "_omp_mi_red_")) != NULL)
00491           sscanf(s+12, "%d", &num_reduction);
00492         else if ((s = strstr(pstr, "_omp_mi_gtp_")) != NULL)
00493           sscanf(s+12, "%d", &g_threadprivate);
00494         else if ((s = strstr(pstr, "_omp_critical_lock_")) != NULL) {
00495           crit[0] = 0;
00496           sscanf(s+19, "%s", crit);
00497           fprintf(linkh_fp,
00498            "#ifndef __OMP_CRITICAL_LOCK_%1$s__\n"
00499            "#define __OMP_CRITICAL_LOCK_%1$s__\n"
00500            "othread_lock_t _omp_critical_lock_%1$s = OTHREAD_LOCK_INITIALIZER;\n"
00501            "#endif\n", crit);
00502         }
00503       }
00504       pclose(f);
00505     }
00506
00507     if (module_id[0]) {
00508       fprintf(link_fp, "  _omp_module_init(&_omp_module%s, %d, %d, %d, %d, %d,\n"
00509                        "                   _omp_init_gtp%s);\n",
00510         module_id, num_for, num_single, num_sections, num_reduction, g_threadprivate, module_id);
00511
00512       fprintf(linkh_fp, "\nvoid _omp_init_gtp%s();\n", module_id);
00513       fprintf(linkh_fp, "extern _omp_mod_t _omp_module%s;\n", module_id);
00514
00515       fprintf(rtp_fp, "  _omp_module_resize_tp_vars(&_omp_module%s, newlen);\n", module_id);
00516     }
00517     }
00518   fclose(linkh_fp);
00519   fprintf(link_fp, "}\n\n\n");
00520   fprintf(link_fp, "#include\"_omp_resztpv.c\"\n");
00521   fclose(link_fp);
00522   fprintf(rtp_fp, "}\n");
00523   fclose(rtp_fp);
00524
00525   strarglist(strgccargs, &gcc_args);
00526   strarglist(strlinkargs, &link_args);
00527   strarglist(strgoutfile, &goutfile);
00528   strarglist(strprepargs, &prep_args);
00529
00530   sprintf(cmd, "%s %s _omp_modinit.c %s -I%s %s %s %s -L%s %s -lompi %s -lrt -l%s",
00531    SYSCOMPILER, objects, threadlib_define, INCLUDEDIR, strprepargs, strgccargs, strgoutfile, LIBDIR,
00532     strlinkargs, pomp_disable_mask == -1 ? "" : "-lpomp", /* Only link with pomp if it is enabled */
00533     threadlib);                                           /* This is either "pthread" or "thread" */
00534 /*fprintf(stderr,"command line: %s\n", cmd); */
00535   system(cmd);
00536
00537   if (!keep) {
00538     unlink("_omp_resztpv.c");
00539     unlink("_omp_modinit.c");
00540     unlink("_omp_modinit.i");
00541   }
00542   system(rm_obj);
00543 }
00544
00545 int main(int argc, char **argv)
00546 {
00547   arg_t *p;
```

```
00548
00549   if (argc == 1) {
00550     printf("This is %s V.%s using compiler \"%s\" and thread library \"%s\"\n",
00551       PACKAGE, VERSION, SYSCOMPILER, threadlib);
00552     fprintf(stderr, "usage: %s [-k] [--pomp-enable] [--pomp-disable=...] <%s-args>\n",
00553       argv[0], SYSCOMPILER);
00554     fprintf(stderr,
00555       "              -k: keep intermediate files\n"
00556       "   --pomp-enable: enable POMP instrumentation\n"
00557       "  --pomp-disable: disable specific POMP regions\n");
00558     _exit(0);
00559   }
00560   parse_args(argc, argv);
00561   for (p = files.head; p != NULL; p = p->next)
00562     process_file(p->val);
00563   if (linking == 0)
00564     return 0;
00565   do_linking();
00566
00567   return 0;
00568 }
```

ompi-0.8.2/ompi/parser.y

```
00001 %{
00002 /*
00003   OMPi OpenMP Compiler
00004   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00005   Alkis Georgopoulos.
00006
00007   This file is part of OMPi.
00008
00009   OMPi is free software; you can redistribute it and/or modify
00010   it under the terms of the GNU General Public License as published by
00011   the Free Software Foundation; either version 2 of the License, or
00012   (at your option) any later version.
00013
00014   OMPi is distributed in the hope that it will be useful,
00015   but WITHOUT ANY WARRANTY; without even the implied warranty of
00016   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017   GNU General Public License for more details.
00018
00019   You should have received a copy of the GNU General Public License
00020   along with OMPi; if not, write to the Free Software
00021   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00022 */
00023
00024 /*! \file parser.y
00025     \brief Bison input file.
00026  */
00027
00028 #include <unistd.h>
00029 #include <stdio.h>
00030 #include <stdlib.h>
00031 #include <string.h>
00032 #include <sys/time.h>
00033 #include <stdarg.h>
00034
00035 /* define if empty structs are not allowed */
00036 /* #define NO_EMPTY_STRUCT */
00037
00038 #define STR_MAX_LEN     1024
00039
00040 /* identifier length */
00041 #define IDL 32
00042 /* declaration depth */
00043 #define DDEPTH 16
00044
00045 int  is_typedef = 0;                          /* 1 if it is a typedef statement */
00046 int  is_seu = 0;                                      /* struct, enum or union */
00047 int  was_seu = 0;
00048 char *SP = "\0";                                          /* handy empty string */
00049 char dec_spec[DDEPTH][STR_MAX_LEN];           /* $$ from declaration specifiers */
00050 char last_dec_spec[DDEPTH][STR_MAX_LEN];
00051 char func_name[STR_MAX_LEN];                                  /* function name */
```

```
00052 int  n_parblock = 0;                                          /* parallel block counter */
00053 int  n_threadprivate = 0;                          /* number of threadprivate vars in current block */
00054 int  g_threadprivate = 0;                                /* number of global threadprivate vars */
00055 char after_barrier_string[STR_MAX_LEN] = "\0"; /**< usually == "{\n", See remove_redudant_barrier */
00056 int last_barrier_start = -1;       /**< file position where the last implicit barrier was spitted */
00057 int last_barrier_end = -1;  /**< file position where the last implicit barrier + abs string ended */
00058 int last_barrier_out_fp_pos = -1;        /**< out_fp file position for the last implicit barrier */
00059
00060 int orig_line = 0;
00061 char orig_fname[STR_MAX_LEN] = {0};
00062 int new_line = 0;
00063 extern int line_no;
00064 int __has_omp = 0;
00065 int has_main = 0;
00066
00067 char init_statement[16384] = {0};        /* buffer containing initialization statements of vars in a
00068                                                                                              block */
00069 char *ist = init_statement;
00070 #define IST_SEEK() while (*ist) ist++
00071 #define IST_RESET() init_statement[0] = 0, ist = init_statement
00072
00073 int in_parallel = 0;
00074 int nest_parallel = 0;
00075 int in_sections = 0;
00076 int block_level = 0;
00077 int parallel_block_level = -1;
00078 int parallel_dir_depth = 0;
00079 int in_func_def = 0;
00080
00081 FILE *out_fp;                /**< (*.c0) The main output file, all code except parallel regions */
00082 FILE *thread_fun_fp;        /**< (threadfuncs.c0) Thread functions (moved code), before 2nd pass */
00083 FILE *thread_fun_fp2;              /**< (funcs.c) Thread functions (moved code), after 2nd pass */
00084 FILE *thread_fun_fp_h;                                /**< (funcs.h) Thread function prototypes */
00085 FILE *par_var_fp;           /**< (vars.h) Structure declarations with parallel region variables */
00086 FILE *modh_fp;            /**< (mod.h) Variables generated for threadprivate, initialization etc */
00087 FILE *shared_var_fp;                                    /**< (svars.dat) Shared variables */
00088 FILE *typedef_fp;                          /**< (types.h) All typedefs are moved into this file */
00089 FILE *threadp_fp;                                /**< (tpvars.h) Thread private variables */
00090 FILE *copyin_fp;                                       /**< (copyin.c) Copy-in code */
00091 FILE *copyin_fp_h;                                     /**< (copyin.h) Copy-in variables */
00092
00093 char ompc_func[STR_MAX_LEN];              /**< name of the func we are currently parsing */
00094
00095 typedef enum {
00096   D_NONE, D_PARALLEL, D_SECTIONS, D_FOR, D_SINGLE, D_CRITICAL, D_THREADPRIVATE, D_FLUSH
00097 } directive_e;
00098
00099 typedef struct directive_s {
00100   directive_e type;
00101   unsigned int defshared : 1;
00102   unsigned int has_reduction : 2;                                        /* 0 = no reduction
00103                                                                        1 = reduction code pending
00104                                                                      2 = reduction code spitted */
00105   unsigned int has_nowait: 1;
00106   unsigned int is_parallel: 1;                                        /* 1 = #pragma omp parallel for
00107                                                                      or #pragma omp parallel sections
00108                                                          (both parallel and a work-sharing construct) */
00109   unsigned int has_cpriv: 1;
00110   int copyin;                        /* value of num_copyin, or -1 if there aren't any copyin vars */
00111
00112   int depth;
00113   int block_level;
00114   char if_expr[STR_MAX_LEN];
00115   char numth_expr[STR_MAX_LEN];
00116   struct directive_s *next;
00117 } directive_t;
00118
00119 #define V_FIRST      4
00120 #define V_LAST       8
00121 typedef enum {
00122   V_NONE = 0, V_SHARED = 1, V_PRIVATE = 2, V_FIRSTPRIVATE = 6, V_LASTPRIVATE = 10,
00123   V_FIRSTLASTPRIVATE = 14, V_THREADPRIVATE = 16, V_COPYIN = 32, V_CPRIV = 64
00124 } directive_vis_e;
00125
00126 typedef enum {
00127   R_NONE, R_ADD, R_MULT, R_SUB, R_BAND, R_BOR, R_XOR, R_LAND, R_LOR
00128 } reduction_op_t;
```

```
00129 char *reduction_init_val[9] = {NULL, "0", "1", "0", "-0", "0", "0", "1", "0"};
00130 char *reduction_op_str[9] = {NULL, "+", "*", "-", "&", "|", "^", "&&", "||"};
00131
00132 /** Info about the #pragma omp line we are currently parsing */
00133 typedef struct {
00134   directive_e active_directive;
00135   directive_vis_e data_clause_type;
00136   reduction_op_t reduction_op;
00137   unsigned int def_found: 1;
00138   unsigned int expr: 1;                    /**< Parse expression with variable transformations */
00139 } directive_line_t;
00140
00141 typedef struct {
00142   char var[STR_MAX_LEN];
00143   char incr[STR_MAX_LEN];
00144   char lb[STR_MAX_LEN];
00145   char b[STR_MAX_LEN];
00146   char chunksize[STR_MAX_LEN];
00147   char schedule[STR_MAX_LEN];
00148   unsigned int eq_op : 2;
00149   unsigned int has_ordered : 1;
00150   unsigned int has_schedule : 1;
00151   char log_op;
00152 } for_t;
00153
00154 #define DT_BASETYPE 0
00155 #define DT_ARRAY 1
00156 #define DT_FUNCTION 2
00157
00158 struct {
00159   char name[STR_MAX_LEN];
00160   int id_h, id_l;                                          /**< Identifier index */
00161   unsigned int type : 2;
00162 } declared_var[DDEPTH];
00163
00164 int dec_depth = 0;
00165
00166 directive_line_t directive_line;
00167 directive_t *Directive;
00168
00169 for_t for_info;
00170
00171 int num_sections    = -1;
00172 int num_for         = -1;
00173 int num_single      = -1;
00174 int num_parallel    = -1;
00175
00176 int num_reduction   = 0;
00177 int num_copyin      = 0;
00178
00179 /* max directive depth */
00180 #define DIRECTIVE_DEPTH 16
00181
00182 /**
00183   Node for a list of critical regions. Used to keep track of which critical locks have already
00184   been declared.
00185  */
00186 typedef struct critical_id_s {
00187   char phrase[STR_MAX_LEN];                          /**< The name of the critical region */
00188   struct critical_id_s *next;                /**< Pointer to the previous critical region found */
00189 } critical_id_t;
00190
00191 typedef struct entity_s {
00192   char type[STR_MAX_LEN];                          /**< Type as declared in source file */
00193   char name[STR_MAX_LEN];                              /**< May contain '[]()*....' */
00194   char *new_id;                                    /**< New name for threadprivate var */
00195   int id_h;                                /**< Index in name (where pure name begins) */
00196   int id_l;                                                /**< Pure name length */
00197   unsigned int dtype : 2;                    /**< DataType = base, array or function */
00198   struct entity_s *next;
00199   struct {
00200     unsigned int used[DIRECTIVE_DEPTH];
00201     unsigned int var_reffed : 1;         /**< 1 = declaration has been put in struct and var will be
00202                                                                          accessed by _OMP_VARREF() */
00203     unsigned int arg : 1;                                  /**< 1 if it is a function argument */
00204     unsigned int reprivate : 1;
00205     unsigned int threadprivate : 1;
```

```
00206     unsigned int cpriv[DIRECTIVE_DEPTH];                    /**< copyprivate */
00207     directive_vis_e vis[DIRECTIVE_DEPTH];
00208     reduction_op_t reduction_op[DIRECTIVE_DEPTH];
00209   } flags;
00210 } entity;
00211
00212 /** List of scopes; each one has a list of entities. */
00213 typedef struct scope_s {
00214   entity *head;
00215   struct scope_s *next;
00216   int level;
00217 } scope;
00218
00219 typedef struct type_entity_s {
00220   char name[IDL];                                   /**< Original name of typedef */
00221   char new_name[STR_MAX_LEN];                       /**< New name when moved to top of file */
00222   struct type_entity_s *next;
00223 } type_entity;
00224
00225 /** List of scopes for new types. */
00226 typedef struct type_scope_s {
00227   type_entity *head;
00228   struct type_scope_s *next;
00229   int level;
00230 } type_scope;
00231
00232 /** Just a list of integeres; needed in two places. */
00233 typedef struct int_list_s {
00234   int count;
00235   struct int_list_s *next, *prev;
00236 } int_list;
00237
00238 /*
00239 typedef struct strlistelem_s {
00240   char data[IDL];
00241   struct strlistelem_s *next;
00242 } strlistelem_t;
00243
00244 typedef struct {
00245   strlistelem_t *head, *tail;
00246 } strlist_t;
00247 */
00248
00249 /*
00250 void strlist_init(strlist_t *l);
00251 void strlist_add(strlist_t *l, char *item);
00252 void strlist_free(strlist_t *l);
00253 */
00254
00255 int add_critical(char *s);
00256 int is_const(entity *e);
00257 int is_static(entity *e);
00258 void del_word(char *dest, char *word, int len, char *src);
00259 void gen_local_type(char *dest, char *src);
00260 int var_in_struct(entity *e, int level);
00261 void initialize_parallel_block();
00262 void finalize_parallel_block();
00263 void handle_function_definition(char *declarator);
00264 void handle_identifier(char *result, char *identifier);
00265 void handle_variable_list(char *identifier);
00266
00267 void spit_implicit_barrier(directive_t *d, char *abs);
00268 void spit_barrier_code();
00269 int remove_redudant_barrier();
00270 void spit_vis_initializations();
00271 void spit_reduction_code();
00272 void spit_lastprivate_assign();
00273 void spit_copyprivate_store();
00274 void spit_copyprivate_assign();
00275
00276 /* malloc() with error handling */
00277 void *safe_malloc(size_t size, char *file_name, int line_number);
00278 /* a macro to automatically include file name and line number */
00279 #define SAFE_MALLOC(size) safe_malloc(size, "__FILE__", __LINE__)
00280 /* strdup() with error handling */
00281 char *safe_strdup(char *s, char *file_name, int line_number);
00282 /* a macro to automatically include file name and line number */
```

```
00283 #define SAFE_STRDUP(s) safe_strdup(s, "__FILE__", __LINE__)
00284
00285 void show_error(char *format, ...);
00286 void show_warning(char *format, ...);
00287 FILE *spit_in_which_file();
00288 void spitcode(char *format, ...);
00289
00290 /*
00291  POMP specific types, variables and functions.
00292  Remember that the pomp library is not needed and therefore NOT #included in the parser.
00293  */
00294
00295 /**
00296  Enumeration with all the pomp-monitored regions.
00297  Also serves as index for the string arrays pomp_region_name etc.
00298  */
00299 typedef enum {
00300   PR_ATOMIC = 0, PR_BARRIER, PR_CRITICAL, PR_FOR, PR_MASTER,
00301   PR_PARALLEL, PR_USER, PR_SECTION, PR_SECTIONS, PR_SINGLE
00302 } pomp_region;
00303
00304 /**
00305   The names of the respective "pomp_region" regions. Should be lowercase, example "pomp_lib.c"
00306   implementation checks for "barrier" using [if ( r->name[0] == 'b' )].
00307  */
00308 char *pomp_region_name[10] = {
00309   "atomic", "barrier", "critical", "for", "master",
00310   "parallel", "region", "section", "sections", "single"};
00311
00312 /**
00313   Which function to call for each "pomp_region" entrance. For example,
00314   POMP_Atomic_enter(&omp_rd_14) is spitted for atomic directive.
00315  */
00316 char *pomp_region_enter_function[10] = {
00317   "Atomic_enter", "Barrier_enter", "Critical_enter", "For_enter", "Master_begin",
00318   "Parallel_fork", "Begin", "Section_begin", "Sections_enter", "Single_enter"};
00319
00320 /**
00321   Which function to call for each "pomp_region" exit. For example,
00322   POMP_Atomic_exit(&omp_rd_14) is spitted for atomic directive.
00323  */
00324 char *pomp_region_exit_function[10] = {
00325   "Atomic_exit", "Barrier_exit", "Critical_exit", "For_exit", "Master_end",
00326   "Parallel_join", "End", "Section_end", "Sections_exit", "Single_exit"};
00327
00328 /**
00329   Differences from struct ompregdescr declared in pomp_lib.h:
00330   1) pomp_region is kept instead of name of construct
00331   2) file_name is not kept, it is always the same when parsing a single file
00332   3) begin_first_line, begin_last_line are assumed to be the same and are replaced with begin_line
00333   4) end_first_line, end_last_line are assumed to be the same and are replaced with end_line
00334   5) *stack_previous is added because we need to keep a stack of the active directives
00335   6) *next is used to keep a list of all directives, so it was renamed to *list_next
00336   If instrumentation is on at the time a region is entered, then, even if it is afterwards
00337   switched off, the already "started" regions SHOULD BE instrumented. For example, if a parallel
00338   directive is started and instrumentation is on, and then a #pragma omp inst off is encountered,
00339   the rest (closing) pomp calls for the PARALLEL directive should NOT be omitted.
00340   7) depth field is used to keep the number of encapsulated region at the start of the region.
00341      Example:
00342        #parallel depth == 1
00343          #pomp off
00344            #parallel depth == 2
00345            #end parallel depth == 2      ==> we know that this does not refer to #parallel depth == 1
00346        #end parallel depth == 1          ==> ok, instrument this one
00347   8) index field is used to keep the number of the struct, e.g. omp_rd_14.
00348  */
00349 struct ompregdescr_parsing {
00350   pomp_region pr;                                          /**< instead of name of construct */
00351   char* sub_name;                                                /**< optional: region name */
00352   int   num_sections;                                  /**< sections only: number of sections */
00353   int   begin_line;                                      /**< line number opening pragma */
00354   int   end_line;                                        /**< line number closing pragma */
00355   int   depth;                          /**< depth of encapsulated pomp regions including this one */
00356   int   index;                                  /**< == pomp_regions value upon region entrance */
00357   struct ompregdescr_parsing *stack_previous;       /**< stack = encapsulated active directives */
00358   struct ompregdescr_parsing *list_next;            /**< list of all the directives encountered */
00359 };
```

```
00360
00361 int pomp_regions = 0;          /**< counter of pomp regions, EXCLUDING the ones not instrumented */
00362 int pomp_depth = 0;  /**< depth of encapsulated pomp regions, INCLUDING the ones not instrumented */
00363 int pomp_instrument = 1;                                  /**< keeps track of #pragma INST ON / OFF */
00364 int pomp_disable = 0;      /**< It is a collection of 10 one-bit booleans. Bit #0 <=> PR_ATOMIC etc.
00365                                 If any bits are set, monitoring for these directives is disabled.
00366                                                            See also compiler option --pomp-disable */
00367 int pomp_sync = 0;  /**< if "--pomp-disable=sync" parameter was passed, don't spit any pomp staff */
00368 struct ompregdescr_parsing *pomp_rd_list_root = NULL; /**< the first (pomp) directive encountered */
00369 struct ompregdescr_parsing *pomp_rd_list_last = NULL;  /**< the last (pomp) directive encountered */
00370 struct ompregdescr_parsing *pomp_rd_stack = NULL;       /**< actually the active (pomp) directive */
00371
00372 void pomp_spit_code(char *format, ...);          /* spits code only if instrumentation is enabled */
00373 void pomp_spit_region_code(pomp_region pr, char *format);          /* checks if POMP_DEPTHS_MATCH */
00374 int  pomp_region_instr_on(pomp_region pr);              /* checks pomp_instrument and pomp_disable */
00375 void pomp_region_enter(pomp_region pr, char *sub_name);     /* spits code, updates list and stack */
00376 void pomp_region_exit(pomp_region pr);                      /* spits code, updates list and stack */
00377 void pomp_spit_regdescr(FILE *f);    /* called at the end of parsing to spit all the rd variables */
00378
00379 void free_entity_list(entity **e);
00380 entity *get_entity(char *name, int *level);
00381 type_entity *get_type_entity(char *name, int *level);
00382 void open_directive_scope(directive_e directive_type);
00383 void close_directive_scope();
00384 void open_scope();
00385 void close_scope();
00386 void add_entity(entity **dest, int arg, char *type, char *name, int head, int len, int dtype);
00387 void print_Scope();
00388
00389 void open_type_scope();
00390 void close_type_scope();
00391 void gen_unique_name(char *dest, char *src);
00392 void add_type_entity(char *name);
00393
00394 scope *Scope = NULL;
00395 entity *func_args = NULL;
00396 type_scope *Type_Scope = NULL;
00397 critical_id_t *Critical = NULL;                  /**< The root of the named critical regions list */
00398 int_list *bt_head, *bt_tail;                                                  /**< Block tree code */
00399 int_list *sec_head, *sec_tail;          /**< Section jobs (# of 'section' in each 'sections') */
00400 /* strlist_t idlist;                     /**< For 'expr' evaluation at schedule(kind, chunksize) */
00401
00402 char module_id[30];
00403
00404 %}
00405
00406 %union {
00407   char name[1024];
00408   int type;
00409 }
00410
00411 %token <name>  IDENTIFIER TYPE_NAME CONSTANT STRING_LITERAL
00412 %token <name>  PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
00413 %token <name>  AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
00414 %token <name>  SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
00415 %token <name>  XOR_ASSIGN OR_ASSIGN SIZEOF
00416
00417 %token <name>  TYPEDEF EXTERN STATIC AUTO REGISTER
00418 %token <name>  CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
00419 %token <name>  STRUCT UNION ENUM ELIPSIS RANGE
00420
00421 %token <name>  CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
00422
00423 %token <name>  PRAGMA_OMP PRAGMA_OMP_THREADPRIVATE OMP_PARALLEL OMP_SECTIONS OMP_NOWAIT OMP_ORDERED
00424 %token <name>  OMP_SCHEDULE OMP_STATIC OMP_DYNAMIC OMP_GUIDED OMP_RUNTIME OMP_SECTION
00425 %token <name>  OMP_SINGLE OMP_MASTER OMP_CRITICAL OMP_BARRIER OMP_ATOMIC OMP_FLUSH
00426 %token <name>  OMP_PRIVATE OMP_FIRSTPRIVATE
00427 %token <name>  OMP_LASTPRIVATE OMP_SHARED OMP_DEFAULT OMP_NONE OMP_REDUCTION
00428 %token <name>  OMP_COPYIN OMP_NUMTHREADS OMP_COPYPRIVATE OMP_FOR OMP_IF
00429
00430 /* additional POMP tokens */
00431 %token <name>  POMP_INST POMP_INIT POMP_FINALIZE POMP_ON POMP_OFF
00432 %token <name>  POMP_BEGIN POMP_END POMP_NOINSTRUMENT POMP_INSTRUMENT
00433
00434 %type <name> primary_expr
00435 %type <name> postfix_expr
00436 %type <name> argument_expr_list
```

```
00437 %type <name> unary_expr
00438 %type <name> unary_operator
00439 %type <name> cast_expr
00440 %type <name> multiplicative_expr
00441 %type <name> additive_expr
00442 %type <name> shift_expr
00443 %type <name> relational_expr
00444 %type <name> equality_expr
00445 %type <name> and_expr
00446 %type <name> exclusive_or_expr
00447 %type <name> inclusive_or_expr
00448 %type <name> logical_and_expr
00449 %type <name> logical_or_expr
00450 %type <name> conditional_expr
00451 %type <name> assignment_expr
00452 %type <name> assignment_operator
00453 %type <name> expr
00454 %type <name> constant_expr
00455 %type <name> declaration
00456 %type <name> declaration_specifiers
00457 %type <name> init_declarator_list
00458 %type <name> init_declarator
00459 %type <name> storage_class_specifier
00460 %type <name> type_specifier
00461 %type <name> struct_or_union_specifier
00462 %type <name> struct_identifier
00463 %type <name> struct_or_union
00464 %type <name> struct_declaration_list
00465 %type <name> struct_declaration
00466 %type <name> struct_declarator_list
00467 %type <name> struct_declarator
00468 %type <name> enum_
00469 %type <name> enum_specifier
00470 %type <name> enumerator_list
00471 %type <name> enumerator
00472 %type <name> declarator
00473 %type <name> declarator_lock
00474 %type <name> declarator2
00475 %type <name> pointer
00476 %type <name> type_specifier_list
00477 %type <name> parameter_identifier_list
00478 %type <name> identifier_list
00479 %type <name> parameter_type_list
00480 %type <name> parameter_list
00481 %type <name> parameter_declaration
00482 %type <name> type_name
00483 %type <name> abstract_declarator
00484 %type <name> abstract_declarator2
00485 %type <name> initializer
00486 %type <name> initializer_list
00487 %type <name> statement
00488 %type <name> openmp_construct
00489 %type <name> openmp_directive
00490 %type <name> structured_block
00491
00492 /* additional POMP construct type */
00493 %type <name> pomp_construct
00494
00495 %type <name> parallel_construct
00496 %type <name> parallel_clause_optseq
00497 %type <name> parallel_directive
00498 %type <name> parallel_clause
00499 %type <name> unique_parallel_clause
00500 %type <name> for_construct
00501 %type <name> for_clause_optseq
00502 %type <name> for_directive
00503 %type <name> for_clause
00504 %type <name> unique_for_clause
00505 %type <name> schedule_kind
00506 %type <name> sections_construct
00507 %type <name> sections_clause_optseq
00508 %type <name> sections_directive
00509 %type <name> sections_clause
00510 %type <name> section_scope
00511 %type <name> section_sequence
00512 %type <name> section_directive
00513 %type <name> single_construct
```

```
00514 %type <name> single_clause_optseq
00515 %type <name> single_directive
00516 %type <name> single_clause
00517 %type <name> parallel_for_construct
00518 %type <name> parallel_for_clause_optseq
00519 %type <name> parallel_for_directive
00520 %type <name> parallel_for_clause
00521 %type <name> parallel_sections_construct
00522 %type <name> parallel_sections_clause_optseq
00523 %type <name> parallel_sections_directive
00524 %type <name> parallel_sections_clause
00525 %type <name> master_construct
00526 %type <name> master_directive
00527 %type <name> critical_construct
00528 %type <name> critical_directive
00529 %type <name> region_phrase_opt
00530 %type <name> barrier_directive
00531 %type <name> atomic_construct
00532 %type <name> atomic_directive
00533 %type <name> flush_directive
00534 %type <name> flush_vars_opt
00535 %type <name> ordered_construct
00536 %type <name> ordered_directive
00537 %type <name> threadprivate_directive
00538 %type <name> data_clause
00539 %type <name> reduction_operator
00540 %type <name> variable_list
00541 %type <name> labeled_statement
00542 %type <name> compound_statement
00543 %type <name> statement_list2
00544 %type <name> compound_statement2
00545 %type <name> declaration_list
00546 %type <name> statement_list
00547 %type <name> expression_statement
00548 %type <name> else_statement
00549 %type <name> selection_statement
00550 %type <name> init_expr
00551 %type <name> incr_expr
00552 %type <name> logical_op
00553 %type <name> logical_expr
00554 %type <name> parallel_for_statement
00555 %type <name> iteration_statement
00556 %type <name> jump_statement
00557 %type <name> file
00558 %type <name> external_definition
00559 %type <name> function_definition
00560 %type <name> function_body
00561 %type <name> identifier
00562 %type <name> string_literal_seq
00563 %type <name> open_brace
00564 /*%type <name> print_declaration_specifiers*/
00565
00566 %start file
00567 %%
00568
00569 string_literal_seq:
00570     STRING_LITERAL
00571     {
00572       sprintf($$,"%s",$1);
00573     }
00574   | STRING_LITERAL string_literal_seq
00575     {
00576       sprintf($$, "%s %s", $1, $2);
00577     }
00578 ;
00579
00580 primary_expr:
00581     identifier
00582     {
00583       strcpy($$, $1);
00584     }
00585   | CONSTANT
00586     {
00587       strcpy($$, $1);
00588     }
00589   | string_literal_seq
00590     {
```

```
00591       strcpy($$, $1);
00592     }
00593   | '(' expr ')'
00594     {
00595       sprintf($$,"(%s)",$2);
00596     }
00597 ;
00598
00599 postfix_expr:
00600     primary_expr
00601     {
00602       strcpy($$,$1);
00603     }
00604   | postfix_expr '[' expr ']'
00605     {
00606       sprintf($$,"%s[%s]",$1,$3);
00607     }
00608   | IDENTIFIER '(' ')'
00609     {
00610       sprintf($$,"%s()",$1);
00611     }
00612   | postfix_expr '(' ')'
00613     {
00614       sprintf($$,"%s()",$1);
00615     }
00616   | IDENTIFIER '(' argument_expr_list ')'
00617     {
00618       sprintf($$,"%s(%s)",$1,$3);
00619     }
00620   | postfix_expr '(' argument_expr_list ')'
00621     {
00622       sprintf($$,"%s(%s)",$1,$3);
00623     }
00624   | postfix_expr '.' IDENTIFIER
00625     {
00626       sprintf($$,"%s.%s",$1,$3);
00627     }
00628   | postfix_expr PTR_OP IDENTIFIER
00629     {
00630       sprintf($$,"%s->%s",$1,$3);
00631     }
00632   | postfix_expr INC_OP
00633     {
00634       sprintf($$,"%s++";$1);
00635     }
00636   | postfix_expr DEC_OP
00637     {
00638       sprintf($$,"%s--",$1);
00639     }
00640 ;
00641
00642 argument_expr_list:
00643     assignment_expr
00644     {
00645       strcpy($$,$1);
00646     }
00647   | argument_expr_list ',' assignment_expr
00648     {
00649       sprintf($$,"%s, %s",$1,$3);
00650     }
00651 ;
00652
00653 unary_expr:
00654     postfix_expr
00655     {
00656       strcpy($$,$1);
00657     }
00658   | INC_OP unary_expr
00659     {
00660       sprintf($$,"++%s",$2);
00661     }
00662   | DEC_OP unary_expr
00663     {
00664       sprintf($$,"--%s",$2);
00665     }
00666   | unary_operator cast_expr
00667     {
```

```
00668       sprintf($$,"%s %s",$1,$2);
00669     }
00670   | SIZEOF unary_expr
00671     {
00672       sprintf($$,"sizeof(%s)",$2);
00673     }
00674   | SIZEOF '(' type_name ')'
00675     {
00676       sprintf($$,"sizeof(%s)",$3);
00677     }
00678 ;
00679
00680 unary_operator:
00681     '&'
00682     {
00683       strcpy($$,"&");
00684     }
00685   | '*'
00686     {
00687       strcpy($$,"*");
00688     }
00689   | '+'
00690     {
00691       strcpy($$,"+");
00692     }
00693   | '-'
00694     {
00695       strcpy($$,"-");
00696     }
00697   | '~'
00698     {
00699       strcpy($$,"~");
00700     }
00701   | '!'
00702     {
00703       strcpy($$,"!");
00704     }
00705 ;
00706
00707 cast_expr:
00708     unary_expr
00709     {
00710       strcpy($$,$1);
00711     }
00712   | '(' type_name ')' cast_expr
00713     {
00714       sprintf($$,"(%s) %s",$2,$4);
00715     }
00716 ;
00717
00718 multiplicative_expr:
00719     cast_expr
00720     {
00721       strcpy($$,$1);
00722     }
00723   | multiplicative_expr '*' cast_expr
00724     {
00725       sprintf($$,"%s * %s",$1,$3);
00726     }
00727   | multiplicative_expr '/' cast_expr
00728     {
00729       sprintf($$,"%s / %s",$1,$3);
00730     }
00731   | multiplicative_expr '%' cast_expr
00732     {
00733       sprintf($$,"%s %% %s",$1,$3);
00734     }
00735 ;
00736
00737 additive_expr:
00738     multiplicative_expr
00739     {
00740       strcpy($$,$1);
00741     }
00742   | additive_expr '+' multiplicative_expr
00743     {
00744       sprintf($$,"%s + %s",$1,$3);
```

```
00745     }
00746   | additive_expr '-' multiplicative_expr
00747     {
00748       sprintf($$,"%s - %s",$1,$3);
00749     }
00750 ;
00751
00752 shift_expr:
00753     additive_expr
00754     {
00755       strcpy($$,$1);
00756     }
00757   | shift_expr LEFT_OP additive_expr
00758     {
00759       sprintf($$,"%s << %s",$1,$3);
00760     }
00761   | shift_expr RIGHT_OP additive_expr
00762     {
00763       sprintf($$,"%s >> %s",$1,$3);
00764     }
00765 ;
00766
00767 relational_expr:
00768     shift_expr
00769     {
00770       strcpy($$,$1);
00771     }
00772   | relational_expr '<' shift_expr
00773     {
00774       sprintf($$,"%s < %s",$1,$3);
00775     }
00776   | relational_expr '>' shift_expr
00777     {
00778       sprintf($$,"%s > %s",$1,$3);
00779     }
00780   | relational_expr LE_OP shift_expr
00781     {
00782       sprintf($$,"%s <= %s",$1,$3);
00783      }
00784   | relational_expr GE_OP shift_expr
00785     {
00786       sprintf($$,"%s >= %s",$1,$3);
00787     }
00788 ;
00789
00790 equality_expr:
00791     relational_expr
00792     {
00793       strcpy($$,$1);
00794     }
00795   | equality_expr EQ_OP relational_expr
00796     {
00797       sprintf($$,"%s == %s",$1,$3);
00798     }
00799   | equality_expr NE_OP relational_expr
00800     {
00801       sprintf($$,"%s != %s",$1,$3);
00802     }
00803 ;
00804
00805 and_expr:
00806     equality_expr
00807     {
00808       strcpy($$,$1);
00809     }
00810   | and_expr '&' equality_expr
00811     {
00812       sprintf($$,"%s & %s",$1,$3);
00813     }
00814 ;
00815
00816 exclusive_or_expr:
00817     and_expr
00818     {
00819       strcpy($$,$1);
00820     }
00821   | exclusive_or_expr '^' and_expr
```

```
00822     {
00823       sprintf($$,"%s ^ %s",$1,$3);
00824     }
00825 ;
00826
00827 inclusive_or_expr:
00828     exclusive_or_expr
00829     {
00830       strcpy($$,$1);
00831     }
00832   | inclusive_or_expr '|' exclusive_or_expr
00833     {
00834       sprintf($$,"%s | %s",$1,$3);
00835     }
00836 ;
00837
00838 logical_and_expr:
00839     inclusive_or_expr
00840     {
00841       strcpy($$,$1);
00842     }
00843   | logical_and_expr AND_OP inclusive_or_expr
00844     {
00845       sprintf($$,"%s && %s",$1,$3);
00846     }
00847 ;
00848
00849 logical_or_expr:
00850     logical_and_expr
00851     {
00852       strcpy($$,$1);
00853     }
00854   | logical_or_expr OR_OP logical_and_expr
00855     {
00856       sprintf($$,"%s || %s",$1,$3);
00857     }
00858 ;
00859
00860 conditional_expr:
00861     logical_or_expr
00862     {
00863       strcpy($$,$1);
00864     }
00865   | logical_or_expr '?' logical_or_expr ':' conditional_expr
00866     {
00867       sprintf($$,"%s? %s: %s",$1,$3,$5);
00868     }
00869 ;
00870
00871 assignment_expr:
00872     conditional_expr
00873     {
00874       strcpy($$,$1);
00875     }
00876   | unary_expr assignment_operator assignment_expr
00877     {
00878       sprintf($$,"%s %s %s",$1,$2,$3);
00879     }
00880 ;
00881
00882 assignment_operator:
00883     '='
00884     {
00885       strcpy($$,"=");
00886     }
00887   | MUL_ASSIGN
00888     {
00889       strcpy($$,"*=");
00890     }
00891   | DIV_ASSIGN
00892     {
00893       strcpy($$,"/=");
00894     }
00895   | MOD_ASSIGN
00896     {
00897       strcpy($$,"%=");
00898     }
```

```
00899     | ADD_ASSIGN
00900       {
00901         strcpy($$,"+=");
00902       }
00903     | SUB_ASSIGN
00904       {
00905         strcpy($$,"-=");
00906       }
00907     | LEFT_ASSIGN
00908       {
00909         strcpy($$,"<<=");
00910       }
00911     | RIGHT_ASSIGN
00912       {
00913         strcpy($$,">>=");
00914       }
00915     | AND_ASSIGN
00916       {
00917         strcpy($$,"&=");
00918       }
00919     | XOR_ASSIGN
00920       {
00921         strcpy($$,"^=");
00922       }
00923     | OR_ASSIGN
00924       {
00925         strcpy($$,"|=");
00926       }
00927 ;
00928
00929 expr:
00930       assignment_expr
00931       {
00932         strcpy($$, $1);
00933       }
00934     | expr ',' assignment_expr
00935       {
00936         sprintf($$,"%s, %s",$1, $3);
00937       }
00938 ;
00939
00940 constant_expr:
00941       conditional_expr
00942       {
00943         strcpy($$, $1);
00944       }
00945 ;
00946
00947 declaration:
00948       declaration_specifiers ';'
00949       {
00950         last_dec_spec[dec_depth][0] = 0;
00951         if (was_seu && block_level == 0) {
00952           fprintf(typedef_fp, "%s;\n", $1);
00953           was_seu = 0;
00954         }
00955         else
00956           spitcode("%s;\n", $1);
00957         if (!in_func_def)
00958           free_entity_list(&func_args);
00959         strcpy($$, SP);
00960       }
00961     | declaration_specifiers init_declarator_list ';'
00962       {
00963         last_dec_spec[dec_depth][0] = 0;
00964         spitcode(";\n");
00965         if (is_typedef && !is_seu)
00966           is_typedef = 0;
00967         if (!in_func_def)
00968           free_entity_list(&func_args);
00969         strcpy($$, SP);
00970       }
00971     | threadprivate_directive
00972       {
00973         spitcode("%s\n", $1);
00974         strcpy($$, SP);
00975       }
```

```
00976 ;
00977
00978 declaration_specifiers:
00979     storage_class_specifier
00980     {
00981       strcpy($$, $1);
00982       strcpy(dec_spec[dec_depth], $$);
00983     }
00984   | storage_class_specifier declaration_specifiers
00985     {
00986       sprintf($$,"%s %s", $1, $2);
00987       strcpy(dec_spec[dec_depth], $$);
00988     }
00989   | type_specifier
00990     {
00991       strcpy($$, $1);
00992       strcpy(dec_spec[dec_depth], $$);
00993     }
00994   | type_specifier declaration_specifiers
00995     {
00996       sprintf($$, "%s %s", $1, $2);
00997       strcpy(dec_spec[dec_depth], $$);
00998     }
00999 ;
01000
01001 init_declarator_list:
01002     init_declarator
01003     {
01004       strcpy($$, SP);
01005     }
01006   | init_declarator_list ',' { spitcode(", "); } init_declarator
01007     {
01008       strcpy($$, SP);
01009     }
01010 ;
01011
01012 init_declarator:
01013     declarator
01014     {
01015       /*
01016       char name[STR_MAX_LEN] = {0};
01017       strncpy(name, &declared_var.name[declared_var.id_h], declared_var.id_l);
01018       */
01019       if (dec_spec[dec_depth][0]) {
01020         strcpy(last_dec_spec[dec_depth], dec_spec[dec_depth]);
01021         spitcode("%s ", dec_spec[dec_depth]);
01022         dec_spec[dec_depth][0] = 0;
01023       }
01024       spitcode("%s", $1);
01025       if (!is_typedef && dec_depth == 0)
01026         if (!in_func_def)
01027           add_entity(&Scope->head, 0, last_dec_spec[dec_depth], $1, declared_var[dec_depth].id_h,
01028             declared_var[dec_depth].id_l, declared_var[dec_depth].type);
01029         else
01030           add_entity(&func_args, 1, last_dec_spec[dec_depth], $1, declared_var[dec_depth].id_h,
01031             declared_var[dec_depth].id_l, declared_var[dec_depth].type);
01032     /* if (is_seu) is_seu--; */
01033       strcpy($$, SP);
01034     }
01035   | declarator '='
01036     {
01037       if (dec_spec[dec_depth][0]) {
01038         strcpy(last_dec_spec[dec_depth], dec_spec[dec_depth]);
01039         spitcode("%s ", dec_spec[dec_depth]);
01040         dec_spec[dec_depth][0] = 0;
01041       }
01042       if (dec_depth == 0)
01043         if (!in_func_def)
01044           add_entity(&Scope->head, 0, last_dec_spec[dec_depth], $1, declared_var[dec_depth].id_h,
01045             declared_var[dec_depth].id_l, declared_var[dec_depth].type);
01046         else
01047           add_entity(&func_args, 1, last_dec_spec[dec_depth], $1, declared_var[dec_depth].id_h,
01048             declared_var[dec_depth].id_l, declared_var[dec_depth].type);
01049       spitcode("%s = ", $1);
01050     }
01051     initializer
01052     {
```

```
01053       strcpy($$, SP);
01054     /* if (is_seu) is_seu--; */
01055      }
01056 ;
01057
01058 storage_class_specifier:
01059     TYPEDEF
01060     {
01061       strcpy($$, "typedef");
01062       is_typedef = 1;
01063     }
01064   | EXTERN
01065     {
01066       strcpy($$, "extern");
01067     }
01068   | STATIC
01069     {
01070       strcpy($$, "static");
01071     }
01072   | AUTO
01073     {
01074       strcpy($$, "auto");
01075     }
01076   | REGISTER
01077     {
01078       strcpy($$, "register");
01079     }
01080 ;
01081
01082 type_specifier:
01083     CHAR
01084     {
01085       strcpy($$,$1);
01086     }
01087   | SHORT
01088     {
01089       strcpy($$,$1);
01090     }
01091   | INT
01092     {
01093       strcpy($$,$1);
01094     }
01095   | LONG
01096     {
01097       strcpy($$,$1);
01098     }
01099   | SIGNED
01100     {
01101       strcpy($$,$1);
01102     }
01103   | UNSIGNED
01104     {
01105       strcpy($$,$1);
01106     }
01107   | FLOAT
01108     {
01109       strcpy($$,$1);
01110     }
01111   | DOUBLE
01112     {
01113       strcpy($$,$1);
01114     }
01115   | CONST
01116     {
01117       strcpy($$,$1);
01118     }
01119   | VOLATILE
01120     {
01121       strcpy($$,$1);
01122     }
01123   | VOID
01124     {
01125       strcpy($$,$1);
01126     }
01127   | struct_or_union_specifier
01128     {
01129       strcpy($$,$1);
```

```
01130     }
01131   | enum_specifier
01132     {
01133       strcpy($$,$1);
01134     }
01135   | TYPE_NAME
01136     {
01137       type_entity *e;
01138       int level;
01139
01140       e = get_type_entity($1, &level);
01141       strcpy($$, e->new_name);
01142    /* fprintf(stderr, "==> %s\n", $1); */
01143     }
01144 ;
01145
01146 struct_identifier:
01147     IDENTIFIER
01148     {
01149       strcpy($$, $1);
01150     }
01151   | TYPE_NAME
01152     {
01153       strcpy($$, $1);
01154     }
01155 ;
01156
01157 struct_or_union_specifier:
01158     struct_or_union struct_identifier '{' struct_declaration_list '}'
01159     {
01160       sprintf($$, "%s %s { %s }", $1, $2, $4);
01161       is_seu--;
01162       if (is_seu == 0)
01163         was_seu = 1;
01164     }
01165   | struct_or_union '{' struct_declaration_list '}'
01166     {
01167       sprintf($$, "%s { %s }", $1, $3);
01168       is_seu--;
01169       if (is_seu == 0)
01170         was_seu = 1;
01171     }
01172   | struct_or_union struct_identifier
01173     {
01174       sprintf($$, "%s %s", $1, $2);
01175       is_seu--;
01176       if (is_seu == 0)
01177         was_seu = 1;
01178     }
01179 ;
01180
01181 struct_or_union:
01182     STRUCT
01183     {
01184       strcpy($$, "struct");
01185       is_seu++;
01186     }
01187   | UNION
01188     {
01189       strcpy($$, "union");
01190       is_seu++;
01191     }
01192 ;
01193
01194 struct_declaration_list:
01195     struct_declaration
01196     {
01197       strcpy($$, $1);
01198     }
01199   | struct_declaration_list struct_declaration
01200     {
01201       sprintf($$,"%s%s", $1, $2);
01202     }
01203 ;
01204
01205 struct_declaration:
01206     type_specifier_list struct_declarator_list ';'
```

```
01207     {
01208       sprintf($$, "%s %s;\n", $1, $2);
01209     }
01210 ;
01211
01212 struct_declarator_list:
01213     struct_declarator
01214     {
01215       strcpy($$,$1);
01216     }
01217   | struct_declarator_list ',' struct_declarator
01218     {
01219       sprintf($$,"%s, %s",$1,$3);
01220     }
01221 ;
01222
01223 struct_declarator:
01224     declarator
01225     {
01226       strcpy($$,$1);
01227     }
01228   | ':' constant_expr
01229     {
01230       sprintf($$,":%s",$2);
01231     }
01232   | declarator ':' constant_expr
01233     {
01234       sprintf($$,"%s:%s",$1,$3);
01235     }
01236 ;
01237
01238 enum_:
01239     ENUM
01240     {
01241       is_seu++;
01242     }
01243 ;
01244
01245 enum_specifier:
01246     enum_ '{'
01247     {
01248       spitcode("enum {\n");
01249     }
01250     enumerator_list '}'
01251     {
01252       spitcode("\n} ");
01253       strcpy($$, SP);
01254       is_seu--;
01255       if (is_seu == 0)
01256         was_seu = 1;
01257     }
01258   | enum_ IDENTIFIER '{'
01259     {
01260       spitcode("enum %s {\n", $2);
01261     }
01262     enumerator_list '}'
01263     {
01264       spitcode("\n} ");
01265       strcpy($$, SP);
01266       is_seu--;
01267       if (is_seu == 0)
01268         was_seu = 1;
01269     }
01270   | enum_ IDENTIFIER
01271     {
01272       sprintf($$, "%s %s", $1, $2);
01273       is_seu--;
01274       if (is_seu == 0)
01275         was_seu = 1;
01276     }
01277 ;
01278
01279 enumerator_list:
01280     enumerator
01281     {
01282       spitcode($1);
01283       strcpy($$,SP);
```

```
01284     }
01285   | enumerator_list ','
01286     {
01287       spitcode(",\n");
01288     }
01289     enumerator
01290     {
01291       spitcode($4);
01292       strcpy($$, SP);
01293     }
01294 ;
01295
01296 enumerator:
01297     IDENTIFIER
01298     {
01299       strcpy($$, $1);
01300       add_entity(&Scope->head, 0, "int", $1, 0, strlen($1), DT_BASETYPE);
01301     }
01302   |  IDENTIFIER '=' constant_expr
01303     {
01304       sprintf($$,"%s= %s", $1, $3);
01305       add_entity(&Scope->head, 0, "int", $1, 0, strlen($1), DT_BASETYPE);
01306     }
01307 ;
01308
01309 declarator:
01310     declarator2
01311     {
01312       strcpy($$, $1);
01313     }
01314   | pointer declarator2
01315     {
01316       sprintf($$,"%s  %s", $1, $2);
01317       sprintf(declared_var[dec_depth].name,"%s  %s", $1, $2);
01318       declared_var[dec_depth].id_h += 2 + strlen($1);
01319     }
01320 ;
01321
01322 declarator_lock:
01323     /* empty */
01324     {
01325       dec_depth++;
01326     }
01327 ;
01328
01329 declarator2:
01330     IDENTIFIER
01331     {
01332       if (is_typedef && !is_seu) {
01333         if (dec_depth == 0) {
01334           add_type_entity($1);
01335           sprintf(declared_var[dec_depth].name, "%s", Type_Scope->head->new_name);
01336         }
01337         else
01338           strcpy(declared_var[dec_depth].name, $1);
01339       }
01340       else
01341         strcpy(declared_var[dec_depth].name, $1);
01342       declared_var[dec_depth].id_h = 0;
01343       declared_var[dec_depth].id_l = strlen(declared_var[dec_depth].name);
01344       declared_var[dec_depth].type = DT_BASETYPE;
01345       strcpy($$, declared_var[dec_depth].name);
01346     }
01347   | '(' declarator ')'
01348     {
01349       sprintf($$, "(%s)", $2);
01350       sprintf(declared_var[dec_depth].name, "(%s)", $2);
01351       declared_var[dec_depth].id_h++;
01352     }
01353   | declarator2 '[' ']'
01354     {
01355       /*
01356       sprintf($$, "%s[ ]", $1);
01357       sprintf(declared_var[dec_depth].name, "%s[ ]", $1);
01358       */
01359       sprintf($$, "(*%s)", $1);
01360       sprintf(declared_var[dec_depth].name, "(*%s)", $1);
```

```
01361       declared_var[dec_depth].id_h += 2;
01362       /*
01363       if (declared_var[dec_depth].type == DT_BASETYPE)
01364         declared_var[dec_depth].type = DT_ARRAY;
01365       */
01366     }
01367   | declarator2 '[' constant_expr ']'
01368     {
01369       sprintf($$, "%s[%s]", $1, $3);
01370       sprintf(declared_var[dec_depth].name, "s[%s]", $1, $3);
01371       if (declared_var[dec_depth].type == DT_BASETYPE)
01372         declared_var[dec_depth].type = DT_ARRAY;
01373     }
01374   | declarator2 '(' ')'
01375     {
01376       sprintf($$, "%s( )", $1);
01377       sprintf(declared_var[dec_depth].name, "%s( )", $1);
01378       if (declared_var[dec_depth].type == DT_BASETYPE)
01379         declared_var[dec_depth].type = DT_FUNCTION;
01380       if (block_level == 0 && dec_depth == 0) {
01381         n_parblock = 0;
01382         memset(func_name, 0, sizeof(func_name));
01383         strncpy(func_name, &declared_var[dec_depth].name[declared_var[dec_depth].id_h],
01384           declared_var[dec_depth].id_l);
01385       }
01386     }
01387   | declarator2 '(' declarator_lock parameter_type_list ')'
01388     {
01389       sprintf($$, "%s(%s)", $1, $4);
01390       dec_depth--;
01391       sprintf(declared_var[dec_depth].name, "%s(%s)", $1, $4);
01392       if (declared_var[dec_depth].type == DT_BASETYPE)
01393         declared_var[dec_depth].type = DT_FUNCTION;
01394       if (block_level == 0 && dec_depth == 0) {
01395         n_parblock = 0;
01396         memset(func_name, 0, sizeof(func_name));
01397         strncpy(func_name, &declared_var[dec_depth].name[declared_var[dec_depth].id_h],
01398           declared_var[dec_depth].id_l);
01399       }
01400     }
01401   | declarator2 '(' declarator_lock parameter_identifier_list ')'
01402     {
01403       sprintf($$, "%s(%s)", $1, $4);
01404       dec_depth--;
01405       sprintf(declared_var[dec_depth].name, "%s(%s)", $1, $4);
01406       if (declared_var[dec_depth].type == DT_BASETYPE)
01407         declared_var[dec_depth].type = DT_FUNCTION;
01408       if (block_level == 0 && dec_depth == 0) {
01409         n_parblock = 0;
01410         memset(func_name, 0, sizeof(func_name));
01411         strncpy(func_name, &declared_var[dec_depth].name[declared_var[dec_depth].id_h],
01412           declared_var[dec_depth].id_l);
01413       }
01414     }
01415 ;
01416
01417 pointer:
01418     '*'
01419     {
01420       sprintf($$, "*");
01421     }
01422   | '*' type_specifier_list
01423     {
01424       sprintf($$, "* %s", $2);
01425     }
01426   | '*' pointer
01427     {
01428       sprintf($$, "* %s", $2);
01429     }
01430   | '*' type_specifier_list pointer
01431     {
01432       sprintf($$, "* %s %s", $2, $3);
01433     }
01434 ;
01435
01436 type_specifier_list:
01437     type_specifier
```

```
01438     {
01439       strcpy($$,$1);
01440     }
01441   | type_specifier_list type_specifier
01442     {
01443       sprintf($$,"%s %s",$1,$2);
01444     }
01445 ;
01446
01447 parameter_identifier_list:
01448     identifier_list
01449     {
01450       strcpy($$,$1);
01451     }
01452   | identifier_list ',' ELIPSIS
01453     {
01454       sprintf($$,"%s , %s",$1,$3);
01455     }
01456 ;
01457
01458 identifier_list:
01459     IDENTIFIER
01460     {
01461       if (!is_typedef)
01462         add_entity(&func_args, 1, "int", $1, 0, strlen($1), DT_BASETYPE);
01463       strcpy($$,$1);
01464     }
01465   | identifier_list ',' IDENTIFIER
01466     {
01467       sprintf($$,"%s , %s",$1,$3);
01468     }
01469 ;
01470
01471 parameter_type_list:
01472     parameter_list
01473     {
01474       strcpy($$,$1);
01475     }
01476   | parameter_list ',' ELIPSIS
01477     {
01478       sprintf($$,"%s , %s",$1,$3);
01479     }
01480 ;
01481
01482 parameter_list:
01483     parameter_declaration
01484     {
01485       strcpy($$,$1);
01486     }
01487   | parameter_list ',' parameter_declaration
01488     {
01489       sprintf($$,"%s , %s",$1,$3);
01490     }
01491 ;
01492
01493 parameter_declaration:
01494     type_specifier_list declarator
01495     {
01496       sprintf($$,"%s  %s",$1,$2);
01497       if (dec_spec[dec_depth][0] != 0)
01498         strcpy(last_dec_spec[dec_depth], dec_spec[dec_depth]);
01499       if (dec_spec[dec_depth][0])
01500         spitcode("%s ", dec_spec[dec_depth]), dec_spec[dec_depth][0] = 0;
01501       if (!is_typedef)
01502         add_entity(&func_args, 1, $1, $2, declared_var[dec_depth].id_h,
01503           declared_var[dec_depth].id_l, declared_var[dec_depth].type);
01504     }
01505   | type_name
01506     {
01507       strcpy($$,$1);
01508     }
01509 ;
01510
01511 type_name:
01512     type_specifier_list
01513     {
01514       strcpy($$,$1);
```

```
01515       }
01516     | type_specifier_list abstract_declarator
01517       {
01518         sprintf($$,"%s  %s",$1,$2);
01519       }
01520 ;
01521
01522 abstract_declarator:
01523       pointer
01524       {
01525         strcpy($$,$1);
01526       }
01527     | abstract_declarator2
01528       {
01529         strcpy($$,$1);
01530       }
01531     | pointer abstract_declarator2
01532       {
01533         sprintf($$,"%s  %s",$1,$2);
01534       }
01535 ;
01536
01537 abstract_declarator2:
01538       '(' abstract_declarator ')'
01539       {
01540         sprintf($$,"(%s)",$2);
01541       }
01542     | '[' ']'
01543       {
01544         strcpy($$,"[ ]");
01545       }
01546     | '[' constant_expr ']'
01547       {
01548         sprintf($$,"[%s]",$2);
01549       }
01550     | abstract_declarator2 '[' ']'
01551       {
01552         sprintf($$,"%s [ ]",$1);
01553       }
01554     | abstract_declarator2 '[' constant_expr ']'
01555       {
01556         sprintf($$,"%s[%s]",$1,$3);
01557       }
01558     | '(' ')'
01559       {
01560         strcpy($$,"( )");
01561       }
01562     | '(' parameter_type_list ')'
01563       {
01564         sprintf($$,"(%s)",$2);
01565       }
01566     | abstract_declarator2 '(' ')'
01567       {
01568         sprintf($$,"%s ( )",$1);
01569       }
01570     | abstract_declarator2 '(' parameter_type_list ')'
01571       {
01572         sprintf($$,"%s (%s)",$1,$3);
01573       }
01574 ;
01575
01576 initializer:
01577       assignment_expr
01578       {
01579         spitcode("%s", $1);
01580         strcpy($$, SP);
01581       }
01582     | '{' open_brace initializer_list '}'
01583       {
01584         spitcode("}");
01585         strcpy($$, SP);
01586       }
01587     | '{' open_brace initializer_list ',' '}'
01588       {
01589         spitcode(", }");
01590         strcpy($$, SP);
01591       }
```

```
01592 ;
01593
01594 initializer_list:
01595     initializer
01596     {
01597       strcpy($$, SP);
01598     }
01599   | initializer_list ','
01600     {
01601       spitcode(", ");
01602     }
01603     initializer
01604     {
01605       strcpy($$, SP);
01606     }
01607 ;
01608
01609 statement:
01610     labeled_statement
01611     {
01612       strcpy($$, $1);
01613     }
01614   | compound_statement
01615     {
01616       strcpy($$, $1);
01617     }
01618   | expression_statement
01619     {
01620       strcpy($$, $1);
01621     }
01622   | selection_statement
01623     {
01624       strcpy($$, $1);
01625     }
01626   | iteration_statement
01627     {
01628       strcpy($$, $1);
01629     }
01630   | jump_statement
01631     {
01632       strcpy($$, $1);
01633     }
01634   | openmp_construct
01635     {
01636       strcpy($$, $1);
01637     }
01638 ;
01639
01640 openmp_construct:
01641     parallel_construct
01642     {
01643       strcpy($$, $1);
01644     }
01645   | for_construct
01646     {
01647       strcpy($$, $1);
01648     }
01649   | sections_construct
01650     {
01651       strcpy($$, $1);
01652     }
01653   | single_construct
01654     {
01655       strcpy($$, $1);
01656     }
01657   | parallel_for_construct
01658     {
01659       strcpy($$, $1);
01660     }
01661   | parallel_sections_construct
01662     {
01663       strcpy($$, $1);
01664     }
01665   | master_construct
01666     {
01667       strcpy($$, $1);
01668     }
```

```
01669    | critical_construct
01670      {
01671        strcpy($$, $1);
01672      }
01673    | atomic_construct
01674      {
01675        strcpy($$, $1);
01676      }
01677    | ordered_construct
01678      {
01679        strcpy($$, $1);
01680      }
01681 ;
01682
01683 openmp_directive:
01684      pomp_construct
01685      {
01686        strcpy($$, $1);
01687      }
01688    | barrier_directive
01689      {
01690        strcpy($$, $1);
01691      }
01692    | flush_directive
01693      {
01694        strcpy($$, $1);
01695      }
01696 ;
01697
01698 structured_block:
01699      statement
01700      {
01701        strcpy($$, SP);
01702      }
01703 ;
01704
01705 /* Additional POMP construct type: */
01706 pomp_construct:
01707      PRAGMA_OMP POMP_INST POMP_INIT '\n'
01708      {
01709        spitcode("/* #pragma omp inst init */\n");
01710     /* it is safer not to spit anything since it is automatically inserted at start of MAIN */
01711        strcpy($$, SP);
01712      }
01713    | PRAGMA_OMP POMP_INST POMP_FINALIZE '\n'
01714      {
01715        spitcode("/* #pragma omp inst finalize */\n");
01716     /* it is safer not to spit anything since it is called at_exit() */
01717        strcpy($$, SP);
01718      }
01719    | PRAGMA_OMP POMP_INST POMP_ON '\n'
01720      {
01721        spitcode("/* #pragma omp inst on */\n");
01722        pomp_spit_code("POMP_On();\n");
01723        strcpy($$, SP);
01724      }
01725    | PRAGMA_OMP POMP_INST POMP_OFF '\n'
01726      {
01727        spitcode("/* #pragma omp inst off */\n");
01728        pomp_spit_code("POMP_Off();\n");
01729        strcpy($$, SP);
01730      }
01731    | PRAGMA_OMP POMP_INST POMP_BEGIN region_phrase_opt '\n'
01732 /* region_phrase_opt is the same [optional name] in omp critical or user regions */
01733      {
01734        spitcode("/* #pragma omp inst begin (%s) */\n", $4);
01735        pomp_region_enter(PR_USER, $4);
01736        strcpy($$, SP);
01737      }
01738    | PRAGMA_OMP POMP_INST POMP_END region_phrase_opt '\n'
01739      {
01740        spitcode("/* #pragma omp inst end (%s) */\n", $4);
01741        pomp_region_exit(PR_USER);
01742        strcpy($$, SP);
01743      }
01744    | PRAGMA_OMP POMP_NOINSTRUMENT '\n'
01745      {
```

```
01746       spitcode("/* #pragma omp noinstrument */\n");
01747       pomp_instrument = 0;
01748       strcpy($$, SP);
01749     }
01750   | PRAGMA_OMP POMP_INSTRUMENT '\n'
01751     {
01752       spitcode("/* #pragma omp instrument */\n");
01753       if (!pomp_sync) /* if "--pomp-disable-sync" parameter was passed, don't reenable instrumen. */
01754         pomp_instrument = 1;
01755       strcpy($$, SP);
01756     }
01757 ;
01758
01759 parallel_construct:
01760     parallel_directive
01761     {
01762       initialize_parallel_block();
01763       spit_vis_initializations();
01764       spitcode("%s", init_statement);
01765       IST_RESET();
01766       pomp_spit_region_code(PR_PARALLEL, "{\n  POMP_Parallel_begin(&%s);\n  ");
01767     }
01768     structured_block
01769     {
01770       pomp_spit_region_code(PR_PARALLEL, "  POMP_Parallel_end(&%s);\n}\n");
01771       finalize_parallel_block();
01772       pomp_region_exit(PR_PARALLEL);
01773       strcpy($$, SP);
01774     }
01775 ;
01776
01777 parallel_clause_optseq:
01778     /* empty */
01779     {
01780       strcpy($$, SP);
01781     }
01782   | parallel_clause_optseq parallel_clause
01783     {
01784       sprintf($$, "%s %s", $1, $2);
01785     }
01786   | parallel_clause_optseq ',' parallel_clause
01787     {
01788       sprintf($$, "%s, %s", $1, $3);
01789     }
01790 ;
01791
01792 parallel_directive:
01793     PRAGMA_OMP OMP_PARALLEL
01794     {
01795       open_directive_scope(D_PARALLEL);
01796       Directive->is_parallel = 1;
01797       directive_line.active_directive = D_PARALLEL;
01798       directive_line.expr = 0;
01799       directive_line.def_found = 0;
01800       num_parallel++;
01801     }
01802     parallel_clause_optseq '\n'
01803     {
01804       spitcode("/* #pragma omp parallel %s */\n", $4);
01805       pomp_region_enter(PR_PARALLEL, "");
01806       directive_line.active_directive = D_NONE;
01807       if (Directive->copyin != -1) {
01808         num_copyin++;
01809         fprintf(copyin_fp, "  }\n}\n");
01810       }
01811       strcpy($$, SP);
01812     }
01813 ;
01814
01815 parallel_clause:
01816     unique_parallel_clause
01817     {
01818       strcpy($$, $1);
01819     }
01820   | data_clause
01821     {
01822       strcpy($$, $1);
```

```
01823     }
01824 ;
01825
01826 unique_parallel_clause:
01827     OMP_IF '(' expr ')'
01828     {
01829       if (Directive->if_expr[0] != '#')
01830         show_error("  multiple if () clauses in a parallel directive are not allowed\n");
01831       strcpy(Directive->if_expr, $3);
01832       sprintf($$, "if (%s)", $3);
01833     }
01834   | OMP_NUMTHREADS '(' expr ')'
01835     {
01836       if (strcmp(Directive->numth_expr, "-1") != 0)
01837         show_error("  multiple num_threads () clauses in a parallel directive are not allowed\n");
01838       strcpy(Directive->numth_expr, $3);
01839       sprintf($$, "num_threads (%s)", $3);
01840     }
01841 ;
01842
01843 for_construct:
01844     for_directive parallel_for_statement
01845     {
01846       close_directive_scope();
01847       pomp_region_exit(PR_FOR);
01848       strcpy($$, $1);
01849     }
01850 ;
01851
01852 for_clause_optseq:
01853     /* empty */
01854     {
01855       strcpy($$, SP);
01856     }
01857   | for_clause_optseq for_clause
01858     {
01859       sprintf($$, "%s %s", $1, $2);
01860     }
01861   | for_clause_optseq ',' for_clause
01862     {
01863       sprintf($$, "%s, %s", $1, $3);
01864     }
01865 ;
01866
01867 for_directive:
01868     PRAGMA_OMP OMP_FOR
01869     {
01870       open_directive_scope(D_FOR);
01871       directive_line.active_directive = D_FOR;
01872       directive_line.expr = 0;
01873       directive_line.def_found = 0;
01874       for_info.has_ordered = 0;
01875       for_info.has_schedule = 0;
01876       strcpy(for_info.chunksize, "#");
01877       strcpy(for_info.schedule, "_OMP_STATIC");
01878       num_for++;
01879     }
01880     for_clause_optseq '\n'
01881     {
01882       spitcode("/* #pragma omp for %s */\n", $4);
01883       pomp_region_enter(PR_FOR, "");
01884       spitcode("{\n");
01885       /* spit_vis_initializations(); */
01886       directive_line.active_directive = D_NONE;
01887       strcpy($$, SP);
01888     }
01889 ;
01890
01891 for_clause:
01892     unique_for_clause
01893     {
01894       strcpy($$, $1);
01895     }
01896   | data_clause
01897     {
01898       strcpy($$, $1);
01899     }
```

```
01900   | OMP_NOWAIT
01901     {
01902       if (Directive->has_nowait)
01903         show_error("  multiple nowait clauses are not allowed on a for directive\n");
01904       strcpy($$, "nowait");
01905       Directive->has_nowait = 1;
01906     }
01907 ;
01908
01909 unique_for_clause:
01910     OMP_ORDERED
01911     {
01912       if (for_info.has_ordered)
01913         show_error("  multiple ordered clauses are not allowed on a for directive\n");
01914       strcpy($$, "ordered"); for_info.has_ordered = 1;
01915     }
01916   | OMP_SCHEDULE '(' schedule_kind ')'
01917     {
01918       if (for_info.has_schedule)
01919         show_error("  multiple schedule clauses are not allowed on a for directive\n");
01920       sprintf($$, "%s(%s)", $1, $3);
01921       for_info.has_schedule = 1;
01922     }
01923   | OMP_SCHEDULE '(' schedule_kind ','
01924     {
01925       directive_line.expr = 1;
01926     }
01927     expr ')'
01928     {
01929       directive_line.expr = 0;
01930       if (for_info.has_schedule)
01931         show_error("  multiple schedule clauses are not allowed on a for directive\n");
01932       sprintf($$, "%s(%s, %s)", $1, $3, $6);
01933       if (for_info.schedule[5] == 'R')
01934         show_error("  chunksize must not be specified when runtime scheduling is selected\n");
01935       strcpy(for_info.chunksize, $6);
01936       for_info.has_schedule = 1;
01937     }
01938 ;
01939
01940 schedule_kind:
01941     OMP_STATIC
01942     {
01943       strcpy($$, "static");
01944       strcpy(for_info.schedule, "_OMP_STATIC");
01945     }
01946   | OMP_DYNAMIC
01947     {
01948       strcpy($$, "dynamic");
01949       strcpy(for_info.schedule, "_OMP_DYNAMIC");
01950     }
01951   | OMP_GUIDED
01952     {
01953       strcpy($$, "guided");
01954       strcpy(for_info.schedule, "_OMP_GUIDED");
01955     }
01956   | OMP_RUNTIME
01957     {
01958       strcpy($$, "runtime");
01959       strcpy(for_info.schedule, "_OMP_RUNTIME");
01960     }
01961 ;
01962
01963 sections_construct:
01964     sections_directive
01965     {
01966       in_sections = 1;
01967     }
01968     section_scope
01969     {
01970       close_directive_scope();
01971       if (pomp_rd_stack)
01972         if (pomp_rd_stack->pr == PR_SECTIONS)
01973           pomp_rd_stack->num_sections = sec_tail->count;
01974       pomp_region_exit(PR_SECTIONS);
01975       in_sections = 0;
01976       strcpy($$, SP);
```

```
01977     }
01978 ;
01979
01980 sections_clause_optseq:
01981     /* empty */
01982     {
01983       strcpy($$, SP);
01984     }
01985   | sections_clause_optseq sections_clause
01986     {
01987       sprintf($$, "%s %s", $1, $2);
01988     }
01989   | sections_clause_optseq ',' sections_clause
01990     {
01991       sprintf($$, "%s, %s", $1, $3);
01992     }
01993 ;
01994
01995 sections_directive:
01996     PRAGMA_OMP OMP_SECTIONS {
01997       open_directive_scope(D_SECTIONS);
01998       directive_line.active_directive = D_SECTIONS;
01999       directive_line.expr = 0;
02000       directive_line.def_found = 0;
02001       num_sections++;
02002     }
02003     sections_clause_optseq '\n'
02004     {
02005       spitcode("/* #pragma omp sections %s */\n", $4);
02006       pomp_region_enter(PR_SECTIONS, $4);
02007       spitcode("{\n");
02008       spit_vis_initializations();
02009       spitcode("%s", init_statement);
02010       IST_RESET();
02011       directive_line.active_directive = D_NONE;
02012       strcpy($$, SP);
02013     }
02014 ;
02015
02016 sections_clause:
02017     data_clause
02018     {
02019       strcpy($$, $1);
02020     }
02021   | OMP_NOWAIT
02022     {
02023       if (Directive->has_nowait)
02024         show_error("  multiple nowait clauses are not allowed on a sections directive\n");
02025       strcpy($$, "nowait");
02026       Directive->has_nowait = 1;
02027     }
02028 ;
02029
02030 section_scope:
02031     '{'                                                          /* spit sections */
02032     {
02033       int_list *p;
02034
02035       spitcode(
02036         "{\n"
02037         "    int _omp_section_job;\n"
02038         "    int _omp_sec_id = _omp_module.sections_ofs + %d;\n\n"
02039
02040         "    _omp_init_sections(_omp_sec_id, _omp_num_section[%d]);\n"
02041         "    while (1) {\n"
02042         "        _omp_section_job = _omp_get_next_section(_omp_sec_id);\n"
02043         "        if (_omp_section_job < 0) break;\n"
02044         "        switch (_omp_section_job) {\n"
02045         "            case 0:\n", num_sections, num_sections);
02046       p = (int_list *) SAFE_MALLOC(sizeof(int_list));
02047       p->count = 0;
02048       p->next = NULL;
02049       p->prev = sec_tail;
02050       sec_tail->next = p;
02051       sec_tail = p;
02052     }
02053     section_sequence '}'
```

```
02054     {
02055       spitcode("        } /* switch */ \n");
02056       spitcode("        if (_omp_section_job == %d) {\n", sec_tail->count-1);
02057       spit_lastprivate_assign();
02058       spitcode("        }\n");
02059       spitcode(
02060         "    } /* while */\n"
02061         "}\n");
02062       spit_reduction_code();
02063       spit_implicit_barrier(Directive, "}\n");
02064       strcpy($$, SP);
02065     }
02066 ;
02067
02068 section_sequence:
02069     structured_block
02070     {
02071       strcpy($$, SP);
02072       sec_tail->count++;
02073       pomp_region_exit(PR_SECTION);
02074       spitcode("                break;\n");
02075     }
02076   | section_directive structured_block
02077     {
02078       strcpy($$, SP);
02079       sec_tail->count++;
02080       pomp_region_exit(PR_SECTION);
02081       spitcode("                break;\n");
02082     }
02083   | section_sequence section_directive structured_block
02084     {
02085       strcpy($$, SP);
02086       sec_tail->count++;
02087       pomp_region_exit(PR_SECTION);
02088       spitcode("                break;\n");
02089     }
02090 ;
02091
02092 section_directive:
02093     PRAGMA_OMP OMP_SECTION '\n'
02094     {
02095       if (sec_tail->count)                    /* case 0 was treated in "section_scope" */
02096         spitcode("        case %d:          /* section */\n", sec_tail->count);
02097       pomp_region_enter(PR_SECTION, "");          /* for all cases, including 0 */
02098       strcpy($$, SP);
02099     }
02100 ;
02101
02102 single_construct:
02103     single_directive structured_block
02104     {
02105       pomp_spit_region_code(PR_SINGLE, "  POMP_Single_end(&%s);\n");
02106       if (Directive->has_cpriv)
02107         spit_copyprivate_store();
02108       if (Directive->has_cpriv) {
02109         spit_implicit_barrier(Directive, "");
02110         spit_copyprivate_assign();
02111       }
02112       else
02113         spit_implicit_barrier(Directive, "");                    /* TODO: removed }\n */
02114       spitcode("} /* end single */ \n");
02115       close_directive_scope();
02116       pomp_region_exit(PR_SINGLE);
02117       strcpy($$, SP);
02118     }
02119 ;
02120
02121 single_clause_optseq:
02122     /* empty */
02123     {
02124       strcpy($$, SP);
02125     }
02126   | single_clause_optseq single_clause
02127     {
02128       sprintf($$, "%s %s", $1, $2);
02129     }
02130   | single_clause_optseq ',' single_clause
```

```
02131     {
02132       sprintf($$, "%s, %s", $1, $3);
02133     }
02134 ;
02135
02136 single_directive:
02137     PRAGMA_OMP OMP_SINGLE
02138     {
02139       open_directive_scope(D_SINGLE);
02140       directive_line.active_directive = D_SINGLE;
02141       directive_line.expr = 0;
02142       directive_line.def_found = 0;
02143       Directive->has_cpriv = 0;
02144       num_single++;
02145     }
02146     single_clause_optseq '\n'
02147     {
02148       if (Directive->has_cpriv && Directive->has_nowait)
02149         show_error("  both copyprivate and nowait clauses are not allowed on the same directive\n");
02150       spitcode("/* #pragma omp single %s */\n", $4);
02151       pomp_region_enter(PR_SINGLE, $4);
02152       spitcode("{\n");
02153       spit_vis_initializations();
02154       spitcode("%s", init_statement);
02155       IST_RESET();
02156       directive_line.active_directive = D_NONE;
02157       if (Directive->has_cpriv)
02158         spitcode(
02159           "  /* prepare for copyprivate */\n"
02160           "    _ompi_thread_t *_omp_th, *_omp_thp;\n"
02161           "    int _omp_cpriv_size;\n"
02162           "\n"
02163           "    _omp_th = _OMP_THREAD;\n"
02164           "    _omp_thp = _omp_th->parent;\n");
02165       spitcode(
02166         "/* the first thread executes the code */\n"
02167         "    if (_omp_run_single(_omp_module.single_ofs + %d))\n", num_single);/* TODO: removed { */
02168       pomp_spit_region_code(PR_SINGLE, "  POMP_Single_begin(&%s);\n");
02169       strcpy($$, SP);
02170     }
02171 ;
02172
02173 single_clause:
02174     data_clause
02175     {
02176       strcpy($$, $1);
02177     }
02178   | OMP_NOWAIT
02179     {
02180       if (Directive->has_nowait)
02181           show_error("  multiple nowait clauses are not allowed on a single directive\n");
02182       strcpy($$, "nowait");
02183       Directive->has_nowait = 1;
02184     }
02185 ;
02186
02187 parallel_for_construct:
02188     parallel_for_directive
02189     {
02190       initialize_parallel_block();
02191       spitcode("{\n");
02192       pomp_spit_region_code(PR_PARALLEL, "  POMP_Parallel_begin(&%s);\n");
02193       pomp_region_enter(PR_FOR, "");
02194     }
02195     parallel_for_statement
02196     {
02197       pomp_region_exit(PR_FOR);
02198       pomp_spit_region_code(PR_PARALLEL, "  POMP_Parallel_end(&%s);\n");
02199       finalize_parallel_block();
02200       pomp_region_exit(PR_PARALLEL);
02201       strcpy($$, SP);
02202     }
02203 ;
02204
02205 parallel_for_clause_optseq:
02206     /* empty */
02207     {
```

```
02208       strcpy($$, SP);
02209     }
02210   | parallel_for_clause_optseq parallel_for_clause
02211     {
02212       sprintf($$, "%s %s", $1, $2);
02213     }
02214   | parallel_for_clause_optseq ',' parallel_for_clause
02215     {
02216       sprintf($$, "%s, %s", $1, $3);
02217     }
02218 ;
02219
02220 parallel_for_directive:
02221     PRAGMA_OMP OMP_PARALLEL OMP_FOR
02222     {
02223       open_directive_scope(D_FOR);
02224       Directive->is_parallel = 1;
02225       directive_line.active_directive = D_FOR;
02226       directive_line.expr = 0;
02227       directive_line.def_found = 0;
02228       for_info.has_ordered = 0;
02229       for_info.has_schedule = 0;
02230       strcpy(for_info.chunksize, "#");
02231       strcpy(for_info.schedule, "_OMP_STATIC");
02232       num_for++;
02233       num_parallel++;
02234     }
02235     parallel_for_clause_optseq '\n'
02236     {
02237       spitcode("/* #pragma omp parallel for%s */\n", $5);
02238       pomp_region_enter(PR_PARALLEL, "");
02239       strcpy($$, SP);
02240       directive_line.active_directive = D_NONE;
02241       if (Directive->copyin != -1) {
02242         num_copyin++;
02243         fprintf(copyin_fp, "  }\n}\n");
02244       }
02245     }
02246 ;
02247
02248 parallel_for_clause:
02249     unique_parallel_clause
02250     {
02251       strcpy($$, $1);
02252     }
02253   | unique_for_clause
02254     {
02255       strcpy($$, $1);
02256     }
02257   | data_clause
02258     {
02259       strcpy($$, $1);
02260     }
02261 ;
02262
02263 parallel_sections_construct:
02264     parallel_sections_directive
02265     {
02266       initialize_parallel_block();
02267       in_sections = 1;
02268       spitcode("{\n");
02269       spit_vis_initializations();
02270       spitcode("%s", init_statement); IST_RESET();
02271       pomp_spit_region_code(PR_PARALLEL, "  POMP_Parallel_begin(&%s);\n");
02272       pomp_region_enter(PR_SECTIONS, "");
02273     }
02274     section_scope
02275     {
02276       in_sections = 0;
02277       pomp_region_exit(PR_SECTIONS);
02278       pomp_spit_region_code(PR_PARALLEL, "  POMP_Parallel_end(&%s);\n");
02279       finalize_parallel_block();
02280       pomp_region_exit(PR_PARALLEL);
02281       strcpy($$, SP);
02282     }
02283 ;
02284
```

```
02285 parallel_sections_clause_optseq:
02286     /* empty */
02287     {
02288       strcpy($$, SP);
02289     }
02290   | parallel_sections_clause_optseq parallel_sections_clause
02291     {
02292       sprintf($$, "%s %s", $1, $2);
02293     }
02294   | parallel_sections_clause_optseq ',' parallel_sections_clause
02295     {
02296       sprintf($$, "%s, %s", $1, $3);
02297     }
02298 ;
02299
02300 parallel_sections_directive:
02301     PRAGMA_OMP OMP_PARALLEL OMP_SECTIONS
02302     {
02303       open_directive_scope(D_SECTIONS);
02304       Directive->is_parallel = 1;
02305       directive_line.active_directive = D_SECTIONS;
02306       directive_line.expr = 0;
02307       directive_line.def_found = 0;
02308       num_sections++;
02309       num_parallel++;
02310     }
02311     parallel_sections_clause_optseq '\n'
02312     {
02313       spitcode("/* #pragma omp parallel sections%s */\n", $5);
02314       pomp_region_enter(PR_PARALLEL, "");
02315       directive_line.active_directive = D_NONE;
02316       if (Directive->copyin != -1) {
02317         num_copyin++;
02318         fprintf(copyin_fp, "  }\n}\n");
02319       }
02320       strcpy($$, SP);
02321     }
02322 ;
02323
02324 parallel_sections_clause:
02325     unique_parallel_clause
02326     {
02327       strcpy($$, $1);
02328     }
02329   | data_clause
02330     {
02331       strcpy($$, $1);
02332     }
02333 ;
02334
02335 master_construct:
02336     master_directive
02337     {
02338       spitcode("if (_OMP_THREAD->thread_num == 0) {\n");
02339       pomp_region_enter(PR_MASTER, "");
02340     }
02341     structured_block
02342     {
02343       pomp_region_exit(PR_MASTER);
02344       spitcode("}\n");
02345       strcpy($$, SP);
02346     }
02347 ;
02348
02349 master_directive:
02350     PRAGMA_OMP OMP_MASTER '\n'
02351     {
02352       spitcode("/* #pragma omp master */\n");
02353       strcpy($$, SP);
02354     }
02355 ;
02356
02357 critical_construct:
02358     critical_directive
02359     {
02360       if (add_critical($1))                    /* $1 is the name of the critical region */
02361         fprintf(modh_fp, "extern othread_lock_t _omp_critical_lock_%s;\n", $1);
```

```
02362       pomp_region_enter(PR_CRITICAL, $1);
02363       spitcode("othread_set_lock(&_omp_critical_lock_%s);\n", $1);
02364       spitcode("/* implied flush */\n");
02365       pomp_spit_region_code(PR_CRITICAL, "  POMP_Critical_begin(&%s);\n");
02366     }
02367     structured_block
02368     {
02369       pomp_spit_region_code(PR_CRITICAL, "  POMP_Critical_end(&%s);\n");
02370       spitcode("/* implied flush */\n");
02371       spitcode("othread_unset_lock(&_omp_critical_lock_%s);\n", $1);
02372       pomp_region_exit(PR_CRITICAL);
02373       strcpy($$, $P);
02374     }
02375 ;
02376
02377 critical_directive:
02378     PRAGMA_OMP OMP_CRITICAL
02379     {
02380       directive_line.active_directive = D_CRITICAL;
02381       directive_line.expr = 0;
02382     }
02383     region_phrase_opt '\n'
02384     {
02385       spitcode("/* #pragma omp critical (%s) */\n", $4);
02386       strcpy($$, $4);
02387       directive_line.active_directive = D_NONE;
02388     }
02389 ;
02390
02391 region_phrase_opt:
02392     /* empty */
02393     {
02394       strcpy($$, $P);
02395     }
02396   | '(' IDENTIFIER ')'
02397     {
02398       strcpy($$, $2);
02399     }
02400 ;
02401
02402 barrier_directive:
02403     PRAGMA_OMP OMP_BARRIER '\n'
02404     {
02405       spitcode("/* #pragma omp barrier */\n");
02406       spit_barrier_code();
02407       strcpy($$, $P);
02408     }
02409 ;
02410
02411 atomic_construct:
02412     atomic_directive
02413     {
02414       pomp_region_enter(PR_ATOMIC, "");
02415       spitcode("othread_set_lock(&_omp_atomic_lock);\n");
02416     }
02417     expression_statement
02418     {
02419       spitcode("  othread_unset_lock(&_omp_atomic_lock);\n");
02420       pomp_region_exit(PR_ATOMIC);
02421       strcpy($$, $P);
02422     }
02423 ;
02424
02425 atomic_directive:
02426     PRAGMA_OMP OMP_ATOMIC '\n'
02427     {
02428       spitcode("/* #pragma omp atomic */\n");
02429       strcpy($$, $P);
02430     }
02431 ;
02432
02433 flush_directive:
02434     PRAGMA_OMP OMP_FLUSH
02435     {
02436       directive_line.active_directive = D_FLUSH;
02437       directive_line.expr = 0;
02438       spitcode("/* OMP FLUSH BEGIN */\n");
```

```
02439     }
02440     flush_vars_opt '\n'
02441     {
02442       directive_line.active_directive = D_NONE;
02443       spitcode("/* #pragma omp flush%s */\n", $4);
02444       spitcode("/* OMP FLUSH END */\n");
02445       strcpy($$, SP);
02446     }
02447 ;
02448
02449 flush_vars_opt:
02450     /* empty */
02451     {
02452       strcpy($$, SP);
02453       spitcode("  _omp_flush_all();\n");
02454     }
02455   | '(' variable_list ')'
02456     {
02457       sprintf($$, " (%s)", $2);
02458     }
02459 ;
02460
02461 ordered_construct:                                          /* spit ordered */
02462     ordered_directive
02463     {
02464       spitcode("    _omp_ordered_begin();\n");
02465     } structured_block
02466     {
02467       spitcode("    _omp_ordered_end();\n");
02468       strcpy($$, SP);
02469     }
02470 ;
02471
02472 ordered_directive:
02473     PRAGMA_OMP OMP_ORDERED '\n'
02474     {
02475       spitcode("/* #pragma omp ordered */\n");
02476       strcpy($$, SP);
02477     }
02478 ;
02479
02480 threadprivate_directive:
02481     PRAGMA_OMP_THREADPRIVATE '('
02482     {
02483       directive_line.data_clause_type = V_THREADPRIVATE;
02484       directive_line.active_directive = D_THREADPRIVATE;
02485       directive_line.expr = 0;
02486     }
02487     variable_list ')' '\n'
02488     {
02489       directive_line.data_clause_type = V_NONE;
02490       directive_line.active_directive = D_NONE;
02491       spitcode("/* #pragma omp threadprivate (%s) */\n", $4);
02492       strcpy($$, SP);
02493     }
02494 ;
02495
02496 data_clause:
02497     OMP_PRIVATE '('
02498     {
02499       directive_line.data_clause_type = V_PRIVATE;
02500     }
02501     variable_list
02502     {
02503       directive_line.data_clause_type = V_NONE;
02504     }
02505     ')'
02506     {
02507       sprintf($$, "private(%s)", $4);
02508     }
02509   | OMP_FIRSTPRIVATE '('
02510     {
02511       directive_line.data_clause_type = V_FIRSTPRIVATE;
02512     }
02513     variable_list
02514     {
02515       directive_line.data_clause_type = V_NONE;
```

```
02516     }
02517     ')'
02518     {
02519       sprintf($$, "firstprivate(%s)", $4);
02520     }
02521   | OMP_LASTPRIVATE '('
02522     {
02523       if (directive_line.active_directive == D_FOR || directive_line.active_directive == D_SECTIONS)
02524         directive_line.data_clause_type = V_LASTPRIVATE;
02525       else
02526         show_error("  lastprivate clause is only allowed within 'for' and 'sections' directives\n");
02527     }
02528     variable_list
02529     {
02530       directive_line.data_clause_type = V_NONE;
02531     }
02532     ')'
02533     {
02534       sprintf($$, "lastprivate(%s)", $4);
02535     }
02536   | OMP_SHARED '('
02537     {
02538       if (!(directive_line.active_directive == D_PARALLEL
02539         || (directive_line.active_directive == D_FOR && Directive->is_parallel)
02540         || (directive_line.active_directive == D_SECTIONS && Directive->is_parallel)) )
02541           show_error("  shared clause is only allowed within 'parallel' directive\n");
02542       directive_line.data_clause_type = V_SHARED;
02543     }
02544     variable_list
02545     {
02546       directive_line.data_clause_type = V_NONE;
02547     }
02548     ')'
02549     {
02550       sprintf($$, "shared(%s)", $4);
02551     }
02552   | OMP_DEFAULT '(' OMP_SHARED ')'
02553     {
02554       if (!(directive_line.active_directive == D_PARALLEL
02555         || (directive_line.active_directive == D_FOR && Directive->is_parallel)
02556         || (directive_line.active_directive == D_SECTIONS && Directive->is_parallel)) )
02557           show_error("  default clause is only allowed within 'parallel' directive\n");
02558       if (directive_line.def_found)
02559         show_error("  multiple default clauses in a directive are not allowed\n");
02560       directive_line.def_found = 1;
02561       Directive->defshared = 1;
02562       strcpy($$, "default(shared)");
02563     }
02564   | OMP_DEFAULT '(' OMP_NONE ')'
02565     {
02566       if (!(directive_line.active_directive == D_PARALLEL
02567         || (directive_line.active_directive == D_FOR && Directive->is_parallel)
02568         || (directive_line.active_directive == D_SECTIONS && Directive->is_parallel)) )
02569           show_error("  default clause is only allowed within 'parallel' directive\n");
02570       if (directive_line.def_found)
02571         show_error("  multiple default clauses in a directive are not allowed\n");
02572       directive_line.def_found = 1;
02573       Directive->defshared = 0;
02574       strcpy($$, "default(none)");
02575     }
02576   | OMP_REDUCTION '(' reduction_operator ':' variable_list ')'
02577     {
02578       if (!(directive_line.active_directive == D_PARALLEL
02579         || directive_line.active_directive == D_FOR
02580         || directive_line.active_directive == D_SECTIONS) )
02581           show_error(
02582             "  reduction clause is only allowed within 'parallel',"
02583             " 'for' and 'sections' directives\n");
02584       Directive->has_reduction = 1;
02585       directive_line.reduction_op = R_NONE;
02586       sprintf($$, "reduction(%s: %s)", $3, $5);
02587     }
02588   | OMP_COPYIN '('
02589     {
02590       if (!(directive_line.active_directive == D_PARALLEL
02591         || (directive_line.active_directive == D_FOR && Directive->is_parallel)
02592         || (directive_line.active_directive == D_SECTIONS && Directive->is_parallel)) )
```

```
02593           show_error("  copyin clause is only allowed within 'parallel' directive\n");
02594       if (Directive->copyin == -1) {
02595         fprintf(copyin_fp_h, "void _omp_copyin_%d(int numth);\n", num_copyin);
02596         fprintf(copyin_fp, "void _omp_copyin_%d(int numth)\n"
02597                            "{\n"
02598                            "  int i;\n"
02599                            "  _omp_modules_resize_tp_vars(numth);\n"
02600                            "  if (numth == -1) numth = _omp_max_threads;\n"
02601                            "  for (i = 1; i < numth; i++) {\n", num_copyin);
02602       }
02603       directive_line.data_clause_type = V_COPYIN;
02604       Directive->copyin = num_copyin;
02605     }
02606     variable_list ')'
02607     {
02608       directive_line.data_clause_type = V_NONE;
02609       sprintf($$, "copyin(%s)", $4);
02610     }
02611   | OMP_COPYPRIVATE '('
02612     {
02613       if (directive_line.active_directive != D_SINGLE)
02614         show_error("  copyprivate clause is only allowed within 'single' directive\n");
02615       directive_line.data_clause_type = V_CPRIV;
02616       Directive->has_cpriv = 1;
02617     }
02618     variable_list ')'
02619     {
02620       directive_line.data_clause_type = V_NONE;
02621       sprintf($$, "copyprivate(%s)", $4);
02622     }
02623 ;
02624
02625 reduction_operator:
02626     '+'
02627     {
02628       strcpy($$, "+");
02629       directive_line.reduction_op = R_ADD;
02630     }
02631   | '*'
02632     {
02633       strcpy($$, "*");
02634       directive_line.reduction_op = R_MULT;
02635     }
02636   | '-'
02637     {
02638       strcpy($$, "-");
02639       directive_line.reduction_op = R_SUB;
02640     }
02641   | '&'
02642     {
02643       strcpy($$, "&");
02644       directive_line.reduction_op = R_BAND;
02645     }
02646   | '^'
02647     {
02648       strcpy($$, "^");
02649       directive_line.reduction_op = R_XOR;
02650     }
02651   | '|'
02652     {
02653       strcpy($$, "|");
02654       directive_line.reduction_op = R_BOR;
02655     }
02656   | AND_OP
02657     {
02658       strcpy($$, "&&");
02659       directive_line.reduction_op = R_LAND;
02660     }
02661   | OR_OP
02662     {
02663       strcpy($$, "||");
02664       directive_line.reduction_op = R_LOR;
02665     }
02666 ;
02667
02668 variable_list:
02669     IDENTIFIER
```


```
02670     {
02671       handle_variable_list($1);
02672       strcpy($$, $1);
02673     }
02674   | variable_list ',' IDENTIFIER
02675     {
02676       handle_variable_list($3);
02677       sprintf($$, "%s, %s", $1, $3);
02678     }
02679 ;
02680
02681
02682 labeled_statement:
02683     IDENTIFIER ':'
02684     {
02685       spitcode("%s:\n", $1);
02686     }
02687     statement
02688     {
02689       sprintf($$, SP);
02690     }
02691   | CASE constant_expr ':'
02692     {
02693       spitcode("case %s:\n", $2);
02694     }
02695     statement
02696     {
02697       strcpy($$, SP);
02698     }
02699   | DEFAULT ':'
02700     {
02701       spitcode("default:\n");
02702     }
02703     statement
02704     {
02705       strcpy($$, SP);
02706     }
02707 ;
02708
02709 open_brace:
02710     {
02711       spitcode("{\n");
02712     }
02713 ;
02714
02715 compound_statement:
02716     '{'
02717     {
02718    /* if ((!in_parallel) || (in_parallel && block_level != parallel_block_level)) */
02719       spitcode("{\n");
02720       block_level++;
02721       open_scope();
02722     }
02723     compound_statement2 '}'
02724     {
02725       block_level--;
02726       close_scope();
02727    /* if ((!in_parallel) || (in_parallel && block_level != parallel_block_level)) */
02728 /* if an implicit barrier was just spitted, add "}\n" to the after_barrier_string and remark */
02729       if (spit_in_which_file() == thread_fun_fp && last_barrier_end == ftell(thread_fun_fp) ) {
02730         strcat(after_barrier_string, "}\n");
02731         spitcode("}\n");
02732         last_barrier_end = ftell(thread_fun_fp);
02733       }
02734       else
02735         spitcode("}\n");
02736       strcpy($$, SP);
02737     }
02738 ;
02739
02740 statement_list2:
02741     /* empty */
02742     {
02743       entity *e;
02744
02745       if (block_level == 1 && strcmp(func_name, "main") == 0) {
02746         spitcode("_omp_initialize();\n");
```

```
02747         has_main = 1;
02748         pomp_spit_code("POMP_Init(); /* automatically inserted at start of main */\n");
02749 /* Automatic function instrumentation should be implemented here, but there is a problem:
02750    function entrance can be easily detected, but not function exit. One should look for every
02751    possible way: return, exit, normal function termination etc. So for the time being function
02752    instrumentation should be performed manually. WARNING: ALWAYS instrument the main function,
02753    because the EXPERT tool needs a single entry point (root).
02754         pomp_region_enter(PR_USER, func_name); */
02755       }
02756
02757     /* spitcode("%s", init_statement); IST_RESET(); */
02758
02759       if (n_threadprivate) {                    /* if there exist threadprivate vars in this scope */
02760         spitcode("{ /* initialization of static threadprivate vars -- BEGIN */\n");
02761         spitcode("int i, b;\n");
02762         spitcode("if (_OMP_THREAD != _omp_master_thread) i = _OMP_THREAD->thread_num, b = i + 1;\n"
02763                  "else i = 0, b = _omp_module.len_tp_vars;\n"
02764                  "for (; i < b; i++) {\n");
02765         for (e = Scope->head; e != NULL; e = e->next)
02766           if (e->flags.threadprivate) {
02767             char id[STR_MAX_LEN] = {0};
02768
02769             strncpy(id, &e->name[e->id_h], e->id_l);
02770             spitcode("if (!(*_omp_tp_vars)[i].%s_inited) {\n", e->new_id);
02771             if (e->dtype == DT_BASETYPE)
02772               spitcode("(*_omp_tp_vars)[i].%s = %s;\n", e->new_id, id);
02773             else
02774               spitcode("memcpy((*_omp_tp_vars)[i].%s, %s, sizeof(%s));\n", e->new_id, id, id);
02775             spitcode("    (*_omp_tp_vars)[i].%s_inited = 1;\n"
02776                      "}\n", e->new_id);
02777           }
02778         n_threadprivate = 0;
02779         spitcode("}\n} /* initialisation of static threadprivate vars -- END */\n");
02780       }
02781     }
02782     statement_list {
02783 /* This does not work as expected, because it is inserted AFTER any return/exit instructions...
02784       if (block_level == 1 && strcmp(func_name, "main") == 0) {
02785         pomp_region_exit(PR_USER);
02786       } */
02787       strcpy($$, SP);
02788     }
02789 ;
02790
02791 compound_statement2:
02792     /* empty */
02793     {
02794       strcpy($$, SP);
02795     }
02796   | statement_list2
02797     {
02798       strcpy($$, SP);
02799     }
02800   | declaration_list
02801     {
02802       strcpy($$, SP);
02803     }
02804   | declaration_list statement_list2 { strcpy($$, SP); }
02805 ;
02806
02807 declaration_list:
02808     declaration
02809     {
02810       strcpy($$, SP);
02811     }
02812   | declaration_list declaration
02813     {
02814       strcpy($$, SP);
02815     }
02816 ;
02817
02818 statement_list:
02819     statement
02820     {
02821       strcpy($$, SP);
02822     }
02823   | openmp_directive
```

```
02824     {
02825       strcpy($$, SP);
02826     }
02827   | statement_list statement
02828     {
02829       strcpy($$, SP);
02830     }
02831   | statement_list openmp_directive
02832     {
02833       strcpy($$, SP);
02834     }
02835 ;
02836
02837 expreseion_statement:
02838     ';'
02839     {
02840       spitcode(";");
02841       strcpy($$, SP);
02842     }
02843   | expr ';'
02844     {
02845       spitcode("%s;", $1);
02846       spitcode("\n");
02847       strcpy($$, SP);
02848     }
02849 ;
02850
02851 else_statement:
02852     /* empty */
02853     {
02854       strcpy($$, SP);
02855     }
02856   | ELSE
02857     {
02858       spitcode("else\n");
02859     }
02860     statement
02861 ;
02862
02863 selection_statement:
02864     IF '(' expr ')'
02865     {
02866       spitcode("if (%s)\n", $3);
02867     }
02868     statement else_statement
02869     {
02870       strcpy($$, SP);
02871     }
02872   | SWITCH '(' expr ')'
02873     {
02874       spitcode("switch (%s)\n", $3);
02875       spitcode("{\n");
02876     }
02877     statement
02878     {
02879       spitcode("}\n");
02880       strcpy($$, SP);
02881     }
02882 ;
02883
02884 init_expr:
02885     IDENTIFIER '=' expr
02886     {
02887       entity *e;
02888       int level;
02889
02890       sprintf($$, "%s = %s", $1, $3);
02891       strcpy(for_info.var, $1);
02892       strcpy(for_info.lb, $3);
02893     /* block_level++;
02894       open_scope();
02895       add_entity("int", $1);
02896       Scope->head->flags.vis[Directive->depth] = V_PRIVATE;
02897     */
02898       e = get_entity($1, &level);
02899       if (e == NULL)
02900         show_error("  %s undeclared; maybe you should run cc first...\n", $1);
```

```
02901       if (!(e->flags.vis[Directive->depth] & V_PRIVATE))
02902         e->flags.vis[Directive->depth] = V_PRIVATE;
02903     }
02904 ;
02905
02906 incr_expr:
02907     INC_OP IDENTIFIER
02908     {
02909       sprintf($$, "++%s", $2);
02910       strcpy(for_info.incr, "1");
02911     }
02912   | DEC_OP IDENTIFIER
02913     {
02914       sprintf($$, "--%s", $2);
02915       strcpy(for_info.incr, "(-1)");
02916     }
02917   | IDENTIFIER INC_OP
02918     {
02919       sprintf($$, "%s++", $1);
02920       strcpy(for_info.incr, "1");
02921     }
02922   | IDENTIFIER DEC_OP
02923     {
02924       sprintf($$, "%s--", $1);
02925       strcpy(for_info.incr, "(-1)");
02926     }
02927   | IDENTIFIER ADD_ASSIGN expr
02928     {
02929       sprintf($$, "%s += %s", $1, $3);
02930       strcpy(for_info.incr, $3);
02931     }
02932   | IDENTIFIER SUB_ASSIGN expr
02933     {
02934       sprintf($$, "%s -= %s", $1, $3);
02935       sprintf(for_info.incr, "-(%s)", $3);
02936     }
02937   | IDENTIFIER '=' expr
02938     {
02939       char *s;
02940       char *incr_err = "  for-increment expression not in canonical shape\n";
02941
02942       s = strstr($3, $1);
02943       if (s == NULL)
02944         show_error(incr_err);
02945       if (s == $3) {
02946      /* var = var +/- incr */
02947         s += strlen($1);
02948         if (*s == 0 || (s[1] != '+' && s[1] != '-'))
02949           show_error(incr_err);
02950         sprintf(for_info.incr, "(%s)", s);
02951       }
02952       else if (strcmp(s,$1) == 0) {
02953     /* var = incr + var */
02954         if (*(s-2) != '+')
02955           show_error(incr_err);
02956         for_info.incr[0] = '(';
02957         strcpy(&for_info.incr[1], $3);
02958         for_info.incr[(s-$3+1)-3] = ')';
02959         for_info.incr[(s-$3+1)-2] = 0;
02960       }
02961       else
02962         show_error(incr_err);
02963       sprintf($$, "%s = %s", $1, $3);
02964     }
02965 ;
02966
02967 logical_op:
02968     '<'
02969     {
02970       strcpy($$,"<");
02971       for_info.log_op = '<';
02972       for_info.eq_op = 0;
02973     }
02974   | LE_OP
02975     {
02976       strcpy($$,"<=");
02977       for_info.log_op = '<';
```

```
02978       for_info.eq_op = 1;
02979     }
02980   | '>'
02981     {
02982       strcpy($$,">");
02983       for_info.log_op = '>';
02984       for_info.eq_op = 2;
02985     }
02986   | GE_OP
02987     {
02988       strcpy($$,">=");
02989       for_info.log_op = '>';
02990       for_info.eq_op = 3;
02991     }
02992 ;
02993
02994 logical_expr:
02995     IDENTIFIER logical_op expr
02996     {
02997       sprintf($$, "%s %s %s", $1, $2, $3);
02998       strcpy(for_info.b, $3);
02999     }
03000 ;
03001
03002 parallel_for_statement:                                          /* spit for */
03003     FOR '(' init_expr ';' logical_expr ';' incr_expr ')'
03004     {
03005       int static_chunk;
03006       char isched;
03007       char sched[20];
03008
03009       switch (for_info.schedule[5]) {
03010         case 'S': strcpy(sched, "static"); isched = 's'; break;
03011         case 'D': strcpy(sched, "dynamic"); isched = 'd'; break;
03012         case 'G': strcpy(sched, "guided"); isched = 'g'; break;
03013         case 'R': strcpy(sched, "runtime"); isched = 'r'; break;
03014       }
03015
03016       static_chunk =  isched == 's' && for_info.chunksize[0] != '#';
03017
03018       spit_vis_initializations();
03019     /* spitcode("    int %s;\n", for_info.var); */
03020       if (for_info.eq_op & 1) {                                  /* "=" */
03021         char tmp[STR_MAX_LEN], sign;
03022
03023         if (for_info.eq_op & 2)
03024           sign = '-';
03025         else sign = '+';                                       /* '+' if '>' */
03026         sprintf(tmp, "((%s) %c 1)", for_info.b, sign);
03027         strcpy(for_info.b, tmp);
03028       }
03029
03030       /* spit common declarations */
03031       spitcode("    int _omp_start, _omp_end, _omp_incr, _omp_last_iter = 0;\n");
03032       spitcode("    int _omp_for_id = _omp_module.for_ofs + %d;\n", num_for);
03033       spitcode("    int (*_omp_sched_bounds_func)(int, int, int, int, int, int *, int *,"
03034                " int, int, int *);\n");
03035       spitcode("    /* static with chunksize or runtime */\n");
03036       spitcode("    int _omp_init_start, _omp_nchunks, _omp_c = 0, _omp_chunksize;\n");
03037       spitcode("%s", init_statement);
03038       IST_RESET();
03039       spitcode("    _omp_incr = (%s);\n"
03040                "    _omp_init_directive(_OMP_FOR, _omp_for_id, %s, _omp_incr, %d, %d);\n",
03041              for_info.incr, for_info.lb, for_info.has_ordered, isched);
03042       if (isched != 'r') {                                     /* STATIC, DYNAMIC, GUIDED */
03043         spitcode("    _omp_sched_bounds_func = _omp_%s_bounds;\n", sched);
03044         if (static_chunk) {                                      /* static with chunksize */
03045             spitcode("    _omp_chunksize = (%s);\n", for_info.chunksize);
03046             spitcode("    _omp_static_bounds_chunk(%s, %s, _omp_incr, _omp_chunksize,"
03047                      " &_omp_nchunks, &_omp_init_start);\n", for_info.b, for_info.lb);
03048             spitcode("    while ((*_omp_sched_bounds_func)(%s, %s, _omp_for_id, _omp_incr,"
03049                      " _omp_chunksize, &_omp_start, &_omp_end, _omp_nchunks, _omp_init_start,"
03050                      " &_omp_c)) {\n", for_info.b, for_info.lb);
03051         }
03052         else {
03053           if (isched == 's') {
03054             spitcode("    _omp_static_bounds_default(%s, %s, _omp_incr, &_omp_start, &_omp_end);\n",
```

```
03055                   for_info.b, for_info.lb);
03056           spitcode("    while ((*_omp_sched_bounds_func)(%s, %s, _omp_for_id, _omp_incr, -1,"
03057                   " &_omp_start, &_omp_end, 1, 0, &_omp_c)) {\n", for_info.b, for_info.lb);
03058         }
03059         else {
03060           spitcode("    while ((*_omp_sched_bounds_func)(%s, %s, _omp_for_id, _omp_incr, (%s),"
03061                   " &_omp_start, &_omp_end, 1, 0, &_omp_c)) {\n", for_info.b, for_info.lb,
03062                    for_info.chunksize[0] == '#'? "1": for_info.chunksize);
03063         }
03064       }
03065     }
03066     else {                                                              /* RUNTIME */
03067       spitcode("  { _omp_sched_info_t _omp_sched_info ="
03068             " _OMP_THREAD->parent->con_run[_OMP_FOR][_omp_for_id]->sched_info;\n");
03069       spitcode(
03070         "    switch (_omp_sched_info.sched) {\n"
03071         "      case _OMP_STATIC:\n"
03072         "        _omp_sched_bounds_func = _omp_static_bounds;\n"
03073         "        if (_omp_sched_info.chunksize == -1) {\n"
03074         "          _omp_static_bounds_default(%1$s, %2$s, _omp_incr, &_omp_start, &_omp_end);\n"
03075         "          _omp_chunksize = -1;\n"
03076         "        } else {\n"
03077         "           _omp_chunksize = _omp_sched_info.chunksize;\n"
03078         "           _omp_static_bounds_chunk(%1$s, %2$s, _omp_incr, _omp_chunksize,"
03079         " &_omp_nchunks, &_omp_init_start);\n"
03080         "        }\n"
03081         "        break;\n"
03082         "      case _OMP_DYNAMIC:\n"
03083         "        _omp_sched_bounds_func = _omp_dynamic_bounds;\n"
03084         "        _omp_chunksize = _omp_sched_info.chunksize;\n"
03085         "        if (_omp_chunksize == -1) _omp_chunksize = 1;\n"
03086         "        break;\n"
03087         "      case _OMP_GUIDED:\n"
03088         "        _omp_sched_bounds_func = _omp_guided_bounds;\n"
03089         "        _omp_chunksize = _omp_sched_info.chunksize;\n"
03090         "        if (_omp_chunksize == -1) _omp_chunksize = 1;\n"
03091         "        break;\n"
03092         "    }\n"
03093         "  }\n", for_info.b, for_info.lb);
03094       spitcode("    while ((*_omp_sched_bounds_func)(%s, %s, _omp_for_id, _omp_incr,"
03095         " _omp_chunksize, &_omp_start, &_omp_end, _omp_nchunks, _omp_init_start, &_omp_c)) {\n",
03096         for_info.b, for_info.lb);
03097       }
03098       if (for_info.has_ordered)
03099         spitcode("        _omp_push_for_data(_omp_for_id, _omp_start);\n");
03100       spitcode("    if (_omp_start %c (%s) && _omp_end == (%s)) _omp_last_iter = 1;\n",
03101         for_info.log_op, for_info.b, for_info.b);
03102       spitcode("    for (%1$s = _omp_start; %1$s %2$c _omp_end; %3$s) {\n",
03103         for_info.var, for_info.log_op, $7);
03104     }
03105     statement
03106     {
03107       strcpy($$, SP);
03108       spitcode("        } /* for */\n");
03109       if (for_info.has_ordered)
03110         spitcode("            _omp_pop_for_data();\n");
03111       spitcode("    }\n");
03112       spitcode("    if (_omp_last_iter) { /* lastprivate assignments */\n",
03113               for_info.log_op, for_info.b, for_info.b, for_info.log_op);
03114       spit_lastprivate_assign();
03115       spitcode("    }\n");
03116       spit_reduction_code();
03117       spit_implicit_barrier(Directive, "}\n");
03118     /* close_scope(); block_level--; */
03119     }
03120 ;
03121
03122 iteration_statement:
03123     WHILE '(' expr ')'
03124     {
03125       spitcode("while (%s)\n", $3);
03126     }
03127     statement
03128     {
03129       strcpy($$,SP);
03130     }
03131   | DO
```

```
03132     {
03133       spitcode("do\n");
03134     }
03135     statement WHILE '(' expr ')' ';'
03136     {
03137       spitcode(" while (%s);\n", $6);
03138     }
03139   | FOR '(' ';' ';' ')'
03140     {
03141       spitcode("for (;;)\n");
03142     }
03143     statement
03144     {
03145       strcpy($$, SP);
03146     }
03147   | FOR '(' ';' ';' expr ')'
03148     {
03149       spitcode("for (;;%s)\n", $5);
03150     }
03151     statement
03152     {
03153       strcpy($$, SP);
03154     }
03155   | FOR '(' ';' expr ';' ')'
03156     {
03157       spitcode("for (; %s;)\n", $4);
03158     }
03159     statement
03160     {
03161       strcpy($$, SP);
03162     }
03163   | FOR '(' ';' expr ';' expr ')'
03164     {
03165       spitcode("for (; %s; %s)\n", $4, $6);
03166     }
03167     statement
03168     {
03169       strcpy($$, SP);
03170     }
03171   | FOR '(' expr ';' ';' ')'
03172     {
03173       spitcode("for (%s;;)\n", $3);
03174     }
03175     statement
03176     {
03177       strcpy($$, SP);
03178     }
03179   | FOR '(' expr ';' ';' expr ')'
03180     {
03181       spitcode("for (%s;; %s)\n", $3, $6);
03182     }
03183     statement
03184     {
03185       strcpy($$, SP);
03186     }
03187   | FOR '(' expr ';' expr ';' ')'
03188     {
03189       spitcode("for (%s; %s;)\n", $3, $5);
03190     }
03191     statement
03192     {
03193       strcpy($$, SP);
03194     }
03195   | FOR '(' expr ';' expr ';' expr ')'
03196     {
03197       spitcode("for (%s; %s; %s)\n", $3, $5, $7);
03198     }
03199     statement
03200     {
03201       strcpy($$, SP);
03202     }
03203 ;
03204
03205 jump_statement:
03206     GOTO IDENTIFIER ';'
03207     {
03208       spitcode("goto %s;\n", $2);
```

```
03209       strcpy($$, SP);
03210     }
03211   | CONTINUE ';'
03212     {
03213       spitcode("continue;\n");
03214       strcpy($$, SP);
03215     }
03216   | BREAK ';'
03217     {
03218       spitcode("break;\n");
03219       strcpy($$, SP);
03220     }
03221   | RETURN ';'
03222     {
03223       spitcode("return;");
03224       strcpy($$, SP);
03225     }
03226   | RETURN expr ';'
03227     {
03228       spitcode("return %s;", $2);
03229       strcpy($$, SP);
03230     }
03231 ;
03232
03233 file:
03234     external_definition
03235   | file external_definition
03236 ;
03237
03238 external_definition:
03239     function_definition
03240     {
03241       strcpy($$, SP);
03242     }
03243   | declaration
03244     {
03245       strcpy($$, SP);
03246     }
03247 ;
03248
03249 function_definition:
03250     declarator
03251     {
03252       spitcode("%s\n", $1);
03253       strcpy(dec_spec[dec_depth], "int");
03254       handle_function_definition($1);
03255       strcpy($$, SP);
03256     }
03257     function_body
03258     {
03259       strcpy($$, SP);
03260     }
03261   | declaration_specifiers declarator
03262     {
03263       spitcode("%s %s\n", $1, $2);
03264       handle_function_definition($2);
03265       strcpy($$, SP);
03266     }
03267     function_body
03268     {
03269       strcpy($$, SP);
03270     }
03271 ;
03272
03273 function_body:
03274     compound_statement
03275     {
03276       strcpy($$, SP);
03277     }
03278   | declaration_list
03279     {
03280       spitcode("%s\n", $1);
03281     }
03282     compound_statement
03283     {
03284       strcpy($$, SP);
03285     }
```

```
03286 ;
03287
03288 identifier:
03289     IDENTIFIER
03290     {
03291       handle_identifier($$, $1);
03292     }
03293 ;
03294 %%
03295
03296 #ifdef HAVE_CONFIG_H
03297 #include "config.h"
03298 #endif
03299
03300 extern char yytext[];
03301 extern int column;
03302 extern FILE *yyin;
03303
03304 /*
03305 void strlist_init(strlist_t *l)
03306 {
03307   l->head = l->tail = NULL;
03308 }
03309
03310 void strlist_add(strlist_t *l, char *item)
03311 {
03312   strlistelem_t *p;
03313
03314   for (p = l->head; p != NULL; p = p->next)
03315     if (strcmp(p->data, item) == 0)
03316       return;
03317
03318   p = SAFE_MALLOC(sizeof(strlistelem_t));
03319
03320   strcpy(p->data, item);
03321   p->next = NULL;
03322
03323   if (l->head == NULL) {
03324     l->head = l->tail = p;
03325   }
03326   else {
03327     l->tail->next = p;
03328     l->tail = p;
03329   }
03330 }
03331
03332 void strlist_free(strlist_t *l)
03333 {
03334   strlistelem_t *p, *q;
03335
03336   p = l->head;
03337
03338   while (p != NULL) {
03339     q = p->next;
03340     free(p);
03341     p = q;
03342   }
03343   l->head = l->tail = NULL;
03344 }
03345 */
03346
03347
03348 /**
03349   Returns 0 if critical id for s already exists; otherwise adds it in the list.
03350  */
03351 int add_critical(char *s)
03352 {
03353   critical_id_t *p;
03354
03355   for (p = Critical; p != NULL; p = p->next)
03356     if (strcmp(p->phrase, s) == 0)
03357       return 0;
03358   p = (critical_id_t *) SAFE_MALLOC(sizeof(critical_id_t));
03359   strcpy(p->phrase, s);
03360   p->next = Critical;
03361   Critical = p;
03362
```

```
03363   return 1;
03364 }
03365
03366 /**
03367   Returns 1 if var is const-qualified.
03368  */
03369 int is_const(entity *e)
03370 {
03371   char *c = strstr(e->type, "const");
03372
03373   while (c != NULL) {
03374     if (c[5] == 0 || c[5] == ' ')
03375       return 1;
03376     c = strstr(c + 5, "const");
03377   }
03378   return 0;
03379 }
03380
03381 /**
03382   Returns 1 if var is static.
03383  */
03384 int is_static(entity *e)
03385 {
03386   char *c = strstr(e->type, "static");
03387
03388   while (c != NULL) {
03389     if (c[6] == 0 || c[6] == ' ')
03390       return 1;
03391     c = strstr(c + 5, "static");
03392   }
03393
03394   return 0;
03395 }
03396
03397 /**
03398   Deletes a word from src by overwriting it with spaces.
03399  */
03400 void del_word(char *dest, char *word, int len, char *src)
03401 {
03402   char *c;
03403
03404   strcpy(dest, src);
03405   c = strstr(dest, word);
03406   while (c != NULL) {
03407     if (c[len] == 0 || c[len] == ' ') {
03408       memset(c, ' ', len);
03409       return;
03410     }
03411     c = strstr(c + len, word);
03412   }
03413 }
03414
03415 /**
03416   Generate a local type, that is a type without storage class specifiers.
03417  */
03418 void gen_local_type(char *dest, char *src)
03419 {
03420   char tmp[STR_MAX_LEN];
03421   char *s;
03422
03423   del_word(tmp, "extern", 6, src);
03424   strcpy(dest, tmp);
03425   del_word(tmp, "static", 6, dest);
03426   strcpy(dest, tmp);
03427   del_word(tmp, "auto", 4, dest);
03428   strcpy(dest, tmp);
03429   del_word(tmp, "register", 8, dest);
03430   strcpy(dest, tmp);
03431   del_word(tmp, "const", 5, dest);
03432   strcpy(dest, tmp);
03433   del_word(tmp, "volatile", 8, dest);
03434   strcpy(dest, tmp);
03435   s = dest;
03436   while (*s != 0 && *s == ' ')
03437     s++;
03438   if (*s == 0)
03439     strcpy(dest, "int");
```

```
03440 }
03441
03442 /**
03443   Returns 1 if a variable with this level has been (or will be) moved in the struct;
03444   This actually happens when:
03445     var is not a function
03446     var is not global
03447     var was visible from the outermost parallel directive
03448     and it was shared or firstprivate or firstlastprivate
03449  */
03450 int var_in_struct(entity *e, int level)
03451 {
03452   directive_t *p;
03453
03454   for (p = Directive->next; p != NULL && p->depth >= level; p = p->next)
03455     if (e->flags.vis[p->depth] & V_PRIVATE)
03456       return 0;
03457   return e->flags.var_reffed ||
03458         e->dtype != DT_FUNCTION && in_parallel && (level <= parallel_block_level) && (level > 0);
03459 }
03460
03461 void initialize_parallel_block()
03462 {
03463   if (in_parallel) {
03464     if (Directive->if_expr[0] != '#')
03465       spitcode("  if (%s)\n", Directive->if_expr);
03466     /* copyin variable intialization */
03467     if (Directive->copyin != -1)
03468       spitcode("_omp_copyin_%d(%s), /* initialize copyin variables */\n",
03469                Directive->copyin, Directive->numth_expr);
03470     spitcode("    _omp_create_team((%s), _OMP_THREAD, 0, _OMP_THREAD->sdn->shared_data);\n",
03471           Directive->numth_expr);
03472     if (Directive->if_expr[0] != '#') {
03473       spitcode("  else\n");
03474       if (Directive->copyin != -1)
03475         spitcode("    _omp_copyin_%d(%s), /* initialize copyin variables */\n", Directive->copyin);
03476       spitcode("    _omp_create_team(1, _OMP_THREAD, 0, _OMP_THREAD->sdn->shared_data);\n");
03477     }
03478     spitcode("{\n");
03479   }
03480   if (!nest_parallel) {
03481     in_parallel = nest_parallel = 1;
03482     parallel_block_level = block_level;
03483     parallel_dir_depth = Directive->depth;
03484     sprintf(ompc_func, "%s_parallel_%d", func_name, n_parblock);
03485     n_parblock++;
03486     fprintf(thread_fun_fp_h, "void *%s(void *);\n",ompc_func);
03487     spitcode("void *%s(void *_omp_thread_data)\n{\n"
03488              "int _omp_dummy = _omp_assign_key(_omp_thread_data);\n"
03489              "#OMP_VARS\n", ompc_func);
03490     fprintf(par_var_fp, "typedef struct {\n");
03491 #ifdef NO_EMPTY_STRUCT
03492     fprintf(par_var_fp, "int _omp_dummy_field; /* empty structs are not allowed */\n");
03493 #endif
03494     fprintf(out_fp, "{\n");
03495     fprintf(out_fp, "_OMP_PARALLEL_DECL_VARSTRUCT(%s); /* declare a struct for shared vars */\n",
03496       ompc_func);
03497   }
03498   else
03499     nest_parallel++;
03500 }
03501
03502 void finalize_parallel_block()
03503 {
03504   remove_redudant_barrier();
03505   if (nest_parallel <= 1) {
03506     fprintf(shared_var_fp, "#\n");
03507     spit_reduction_code();
03508     spitcode("return 0;\n}\n");
03509     fprintf(par_var_fp, "} %s_vars;\n\n", ompc_func);
03510     in_parallel = 0;
03511     nest_parallel = 0;
03512     parallel_block_level = -1;
03513     parallel_dir_depth = 0;
03514     if (Directive->if_expr[0] != '#')
03515       spitcode("  if (%s) {\n", Directive->if_expr);
03516     /* copyin variable intialization */
```

```
03517     if (Directive->copyin != -1)
03518       spitcode("_omp_copyin_%d(%s), /* initialize copyin variables */\n",
03519                Directive->copyin, Directive->numth_expr);
03520     spitcode("    _omp_create_team((%s), _OMP_THREAD, %s, (void *) &%s_var);"
03521       "  /* create team of threads */\n", Directive->numth_expr, ompc_func, ompc_func);
03522     spitcode("    _omp_destroy_team(_OMP_THREAD->parent);\n");
03523     if (Directive->if_expr[0] != '#') {
03524       spitcode("  } else {\n"
03525                "      void *tmp = _OMP_THREAD->sdn;\n"
03526                "      _OMP_THREAD->shared_data = (void *) &%s_var;\n"
03527                "      _OMP_THREAD->sdn = _OMP_THREAD;\n"
03528                "      _omp_serialize++;\n"
03529                "      %s((void *) _OMP_THREAD);\n"
03530                "      _omp_serialize--;\n"
03531                "      _OMP_THREAD->sdn = tmp;\n"
03532                "  }\n", ompc_func, ompc_func);
03533     }
03534     spitcode("}\n");
03535   }
03536   else {
03537     spit_reduction_code();
03538     spitcode("}\n");
03539  /* if (Directive->if_expr[0] != '#')
03540       spitcode("  if (%s)\n", Directive->if_expr);
03541   */
03542     spitcode("    _omp_destroy_team(_OMP_THREAD->parent);\n");
03543     nest_parallel--;
03544   }
03545   close_directive_scope();
03546 }
03547
03548 void handle_function_definition(char *declarator)
03549 {
03550   if (dec_spec[dec_depth][0]) {
03551     strcpy(last_dec_spec[dec_depth], dec_spec[dec_depth]);
03552     dec_spec[dec_depth][0] = 0;
03553   }
03554   if (!is_typedef) {
03555     entity *e;
03556     int level;
03557
03558     e = get_entity(func_name, &level);
03559     if (e == NULL)
03560       add_entity(&Scope->head, 0, last_dec_spec[dec_depth], declarator,
03561         declared_var[dec_depth].id_h, declared_var[dec_depth].id_l, declared_var[dec_depth].type);
03562   }
03563   last_dec_spec[dec_depth][0] = 0;
03564   in_func_def = 1;
03565 }
03566
03567 void handle_identifier(char *result, char *identifier)
03568 {
03569   int level;
03570   entity *e;
03571   char *name_tail, name_head[STR_MAX_LEN] = {0};
03572   char id[STR_MAX_LEN] = {0};
03573   char ltype[STR_MAX_LEN] = {0};
03574
03575   strcpy(result, identifier);
03576
03577   e = get_entity(identifier, &level);
03578   /* check for declaration */
03579   if (e == NULL)
03580     show_error("  %s undeclared; maybe you should run cc first...\n", identifier);
03581
03582   /* threadprivate variable reference transformation */
03583   if (e->flags.threadprivate /*&& directive_line.active_directive == D_NONE*/) {
03584     /* if threadprivate variable has not been implicitly specified as private, in a for-loop */
03585     if (!(e->flags.vis[Directive->depth] & V_PRIVATE))
03586       sprintf(result, "(*_omp_tp_vars)[_OMP_THREAD->real_thread_num].%s", e->new_id);
03587     return;
03588   }
03589
03590   if (Directive->type == D_NONE)                    /* do not transform if not in a construct */
03591     return;
03592
03593   if (e->dtype == DT_FUNCTION)                                          /* ignore functions */
```

```
03594     return;
03595
03596   /* if we are in the directive line then don't transform
03597       except on 'expr' evaluation (i.e. chunksize) */
03598   if (directive_line.active_directive != D_NONE && directive_line.expr == 0)
03599     return;
03600
03601   /* variable transformation */
03602
03603   /* default(shared): all vars shared if not explicitly declared... */
03604   if (Directive->defshared && e->flags.vis[Directive->depth] == V_NONE) {
03605     e->flags.vis[Directive->depth] = V_SHARED;
03606     if (e->flags.arg && e->dtype == DT_ARRAY)
03607       show_warning("  array %s as function argument may cause invalid declarations\n", e->name);
03608   }
03609
03610   /* error: default(none)
03611             unspecified variable visibility
03612             not declared in this directive
03613             not const-qualified
03614     */
03615   if (!Directive->defshared && e->flags.vis[Directive->depth] == V_NONE
03616       && level <= Directive->block_level) {
03617     /* except const */
03618     if (is_const(e))
03619       e->flags.vis[Directive->depth] = V_SHARED;
03620     else
03621       show_error("  %s should be explicitly declared as shared/private\n", identifier);
03622     if (e->flags.arg && e->dtype == DT_ARRAY)
03623       show_warning("  array %s as function argument may cause invalid declarations\n", e->name);
03624   }
03625
03626   strncpy(id, &e->name[e->id_h], e->id_l);
03627   name_tail = &e->name[e->id_h + e->id_l];
03628   strncpy(name_head, e->name, e->id_h);
03629
03630   /* shared variable reference transformation */
03631   if (e->flags.vis[Directive->depth] == V_SHARED && var_in_struct(e, level))
03632     sprintf(result, "(*(%s))", id);
03633
03634   /* if it is the first time we encounter a (shared/firstprivate) variable:
03635       * declare a pointer to the variable in the structure
03636       * initialize the pointer so that it points to the variable */
03637   if (!e->flags.var_reffed &&
03638       (e->flags.vis[Directive->depth] == V_SHARED ||
03639       (e->flags.vis[Directive->depth] & (V_FIRST | V_LAST)) ||
03640       e->flags.reduction_op[Directive->depth] != R_NONE) && var_in_struct(e, level)) {
03641     e->flags.var_reffed = 1;
03642     gen_local_type(ltype, e->type);
03643     fprintf(par_var_fp, "    %s %s(*%s)%s;\n", ltype, name_head, id, name_tail);
03644     fprintf(out_fp, "_OMP_PARALLEL_INIT_VAR(%s, %s);\n", ompc_func, id);
03645
03646     /* write to a tmp file a shortcut to the shared variable in the struct,
03647        in order to avoid extensive use of _OMP_VARREF (slower) */
03648     if (!((e->flags.vis[parallel_dir_depth] & (V_FIRST | V_LAST))
03649         || e->flags.reduction_op[parallel_dir_depth] != R_NONE))
03650       fprintf(shared_var_fp, "%s %s(*%s)%s = &_OMP_VARREF(%s, %s);\n", ltype, name_head,
03651               id, name_tail, ompc_func, id);
03652   }
03653
03654   e->flags.used[Directive->depth] = 1;
03655 }
03656
03657 void handle_variable_list(char *identifier)
03658 {
03659   int level;
03660   entity *e = get_entity(identifier, &level);
03661
03662   if (e == NULL)
03663     show_error("  %s undeclared; maybe you should run cc first...\n", identifier);
03664
03665   /* flush */
03666   if (directive_line.active_directive == D_FLUSH) {
03667     char id[STR_MAX_LEN] = {0};
03668
03669     strncpy(id, &e->name[e->id_h], e->id_l);
03670     spitcode("    _omp_flush((void *) &%s);\n", id);
```

```
03671     return;
03672   }
03673
03674   /* copyin */
03675   if (directive_line.data_clause_type == V_COPYIN) {
03676     if (!e->flags.threadprivate)
03677       show_error("  copyin variable %s should be threadprivate\n", identifier);
03678     if (e->dtype == DT_BASETYPE)
03679       fprintf(copyin_fp,
03680         "    (*_omp_tp_vars)[i].%s = (*_omp_tp_vars)[0].%s;\n", e->new_id, e->new_id);
03681     else
03682       fprintf(copyin_fp,
03683         "    memcpy((*_omp_tp_vars)[i].%s, (*_omp_tp_vars)[0].%s, sizeof((*_omp_tp_vars)[0].%s));\n",
03684        e->new_id, e->new_id, e->new_id);
03685     return;
03686   }
03687
03688   /* threadprivate vars cannot be redeclared */
03689   if (e->flags.threadprivate && directive_line.data_clause_type != V_CPRIV)
03690     show_error("  visibility redeclaration of threadprivate variable %s not allowed\n", identifier);
03691
03692   if (directive_line.data_clause_type == V_THREADPRIVATE) {
03693     char ltype[STR_MAX_LEN] = {0}, name_head[STR_MAX_LEN] = {0};
03694     char tmp[STR_MAX_LEN];
03695
03696     if (level < block_level)
03697       show_error("  threadprivate declaration of %s must be in the same scope\n", identifier);
03698
03699     if (level > 0 && !is_static(e))
03700       show_error("  threadprivate declaration of non-static variable %s\n", identifier);
03701
03702     e->flags.threadprivate = 1;
03703     e->new_id = (char *) SAFE_MALLOC(STR_MAX_LEN);
03704     strncpy(name_head, e->name, e->id_h);
03705     strncpy(ltype, &e->name[e->id_h], e->id_l);
03706     gen_unique_name(e->new_id, ltype);
03707     gen_local_type(ltype, e->type);
03708     fprintf(threadp_fp, "%s %s%s%s;\n",
03709       ltype, name_head, e->new_id, &e->name[e->id_h + e->id_l]);
03710     if (level > 0)
03711       fprintf(threadp_fp, "unsigned int %s_inited: 1;\n", e->new_id);
03712     else
03713       g_threadprivate++;
03714     n_threadprivate++;
03715     return;
03716   }
03717
03718   if (directive_line.data_clause_type == V_CPRIV) {
03719     e->flags.cpriv[Directive->depth] = 1;
03720   }
03721
03722   /* if var was private, then it must be shared (and thus not declared) in all enclosed directives
03723       unless it is specified again as private (OpenMP 2.0) */
03724   if (e->flags.vis[Directive->depth-1] == V_PRIVATE && directive_line.data_clause_type == V_PRIVATE)
03725     e->flags.reprivate = 1;
03726   else if ((e->flags.vis[Directive->depth-1] & V_PRIVATE) &&
03727            (e->flags.cpriv[Directive->depth] == 0) && (Directive->type != D_PARALLEL))
03728     show_error("  visibility redeclaration of private variable %s not allowed\n", identifier);
03729
03730   if (directive_line.reduction_op != R_NONE) {
03731     /* variables that appear in the reduction clause must be shared in the enclosing context */
03732     if (!(Directive->next->defshared && e->flags.vis[Directive->depth-1] == V_NONE ||
03733         e->flags.vis[Directive->depth-1] == V_SHARED))
03734       show_error("  reduction variable %s must be shared in the enclosing context\n", identifier);
03735     /* reduction variable must not be const-qualified */
03736     if (is_const(e))
03737       show_error("  reduction variable %s must not be const-qualified\n", identifier);
03738     /* the type of the variables in the reduction clause must be valid for the reduction operator */
03739     if (e->dtype != DT_BASETYPE || strchr(e->type, '*') != NULL)
03740       show_error("  specified reduction operator not applicable to %s\n", identifier);
03741   }
03742
03743   /* if var is specified in more than one clause */
03744   if ((e->flags.vis[Directive->depth] & V_PRIVATE)!=(e->flags.vis[Directive->depth-1] & V_PRIVATE))
03745     if (e->flags.vis[Directive->depth] == V_FIRSTPRIVATE
03746         && directive_line.data_clause_type != V_LASTPRIVATE
03747         || e->flags.vis[Directive->depth] == V_LASTPRIVATE
```

```
03748         && directive_line.data_clause_type != V_FIRSTPRIVATE
03749         || e->flags.vis[Directive->depth] == V_PRIVATE
03750         || e->flags.reduction_op[Directive->depth] != V_NONE)
03751       show_error("  %s should not be specified in more than one clause\n", identifier);
03752
03753   /* if var was firstprivate and is also declared as lastprivate
03754      or var was lastprivate and is also declared as firstprivate
03755      then make it firstlastprivate */
03756   if ((e->flags.vis[Directive->depth] ^ directive_line.data_clause_type) == (V_FIRST | V_LAST))
03757     e->flags.vis[Directive->depth] = V_FIRSTLASTPRIVATE;
03758   else
03759     if (directive_line.reduction_op != R_NONE) {
03760       e->flags.vis[Directive->depth] = V_PRIVATE;
03761       e->flags.reduction_op[Directive->depth] = directive_line.reduction_op;
03762     }
03763     else
03764       e->flags.vis[Directive->depth] = directive_line.data_clause_type;
03765 }
03766
03767 /**
03768  Spits barrier code for directives that need implied barriers.
03769  Also keeps track of the last spitted barrier, see remove_redudant_barrier() for details.
03770  */
03771 void spit_implicit_barrier(directive_t *d, char* abs)
03772 {
03773   if (d->has_nowait == 0 && d->is_parallel == 0) {
03774     if (spit_in_which_file() == thread_fun_fp) {   /* if not, then this is an orphaned directive, */
03775       last_barrier_out_fp_pos = ftell(out_fp);          /* and the barrier should not be marked. */
03776       last_barrier_start = ftell(thread_fun_fp);
03777     }
03778     spitcode("/* implicit barrier */\n");
03779     if (pomp_rd_stack)
03780       pomp_spit_region_code(pomp_rd_stack->pr, "  POMP_Barrier_enter(&%s);\n");
03781     spitcode("  _omp_barrier_wait(&_OMP_THREAD->parent->barrier); /* implied flush */\n");
03782     if (pomp_rd_stack)
03783       pomp_spit_region_code(pomp_rd_stack->pr, "  POMP_Barrier_exit(&%s);\n");
03784     spitcode("%s", abs);
03785     if (spit_in_which_file() == thread_fun_fp) {
03786       last_barrier_end = ftell(thread_fun_fp);
03787       strcpy(after_barrier_string, abs);
03788     }
03789   }
03790   else
03791     spitcode("%s", abs);
03792 }
03793
03794 void spit_barrier_code()
03795 {
03796   pomp_region_enter(PR_BARRIER, "");
03797   spitcode(
03798     "  ompi_barrier_wait((ompi_barrier_t *)&_OMP_THREAD->parent->barrier); /* implied flush */\n");
03799   pomp_region_exit(PR_BARRIER);
03800 }
03801
03802 /**
03803   Optimizes output by trying to remove an already inserted implicit barrier.
03804   It is called at the end of a parallel region, which also implements a barrier, so if another
03805   barrier was already spitted it should be removed.
03806     +- Example: ------------------------------------------------------+
03807     | #omp parallel {                                                 |
03808     |   ...code...                                                    |
03809     |   #omp for {                                                    |
03810     |     ...code...                                                  |
03811     |   } => the implicit barrier inserted here is redudant           |
03812     | }                                                               |
03813     +-----------------------------------------------------------------+
03814   All implicit barriers are placed in thread_fun_fp, except for the ones belonging to orphaned
03815   directives which are placed in out_fp and are not recorded (they cannot be removed).
03816   On each call to spit_implicit_barrier we record the starting and ending file position of the
03817   inserted barrier, including the after_barrier_string.
03818   When a parallel region end is detected, remove_redudant_barrier is called. If the file position is
03819   unchanged, then it is safe to remove the barrier.
03820  */
03821 int remove_redudant_barrier()
03822 {
03823   int i;
03824
```

```
03825 #ifdef DEBUGGING
03826   fprintf(stderr, "/* Remove_redudant_barrier was called: */\n");
03827   fprintf(stderr, "/* last_barrier_end = %d, ftell(thread_fun_fp) = %d,\n"
03828     "last_barrier_out_fp_pos = %d, ftell(out_fp) = %d.*/\n",
03829     last_barrier_end, ftell(thread_fun_fp), last_barrier_out_fp_pos, ftell(out_fp));
03830 #endif
03831   if (last_barrier_end == ftell(thread_fun_fp) && last_barrier_out_fp_pos == ftell(out_fp)) {
03832 #ifdef DEBUGGING
03833     fprintf(stderr, "/* it is safe to remove the barrier */\n");
03834 #endif
03835 #ifndef NO_OPTIMIZATIONS
03836     fseek(thread_fun_fp, last_barrier_start, SEEK_SET);
03837     fprintf(thread_fun_fp, "/* implicit barrier removed by optimizer */\n%s", after_barrier_string);
03838     for (i = ftell(thread_fun_fp) + 1; i < last_barrier_end; i++)
03839       fputc(' ', thread_fun_fp);
03840     fputc('\n', thread_fun_fp);
03841 #endif
03842     last_barrier_start = -1;
03843     last_barrier_end = -1;
03844     last_barrier_out_fp_pos = -1;
03845     strcpy(after_barrier_string, "\0");
03846   }
03847 }
03848
03849 void spit_vis_initializations()
03850 {
03851   scope *s;
03852   entity *e;
03853
03854   for (s = Scope; s != NULL; s = s->next)
03855     for (e = s->head; e != NULL; e = e->next) {
03856       if (e->flags.vis[Directive->depth]!=e->flags.vis[Directive->depth-1] || e->flags.reprivate) {
03857         char *name_tail, name_head[STR_MAX_LEN] = {0};
03858         char id[STR_MAX_LEN] = {0};
03859         char ltype[STR_MAX_LEN];
03860
03861         name_tail = &e->name[e->id_h + e->id_l];
03862         strncpy(id, &e->name[e->id_h], e->id_l);
03863         strncpy(name_head, e->name, e->id_h);
03864         gen_local_type(ltype, e->type);
03865
03866         e->flags.reprivate = 0;
03867         /* firstprivate, lastprivate or both */
03868         if (e->flags.vis[Directive->depth] & (V_FIRST | V_LAST)) {
03869           /* nested or orphaned directives */
03870           if (!var_in_struct(e, s->level))
03871             spitcode("%s %s(*_omp_firstlastprivate_%s)%s = &%s;\n",
03872               ltype, name_head, id, name_tail, id);
03873         }
03874           /* private or lastprivate only */
03875           if (e->flags.vis[Directive->depth] == V_PRIVATE
03876               && e->flags.reduction_op[Directive->depth] == V_NONE
03877               || e->flags.vis[Directive->depth] == V_LASTPRIVATE) {
03878             spitcode("%s %s;\n", ltype, e->name);
03879             if (e->flags.arg && e->dtype == DT_ARRAY)
03880               show_warning("  %s should be declared as pointer\n", e->name);
03881           }
03882           else if (e->flags.reduction_op[Directive->depth] != V_NONE) {     /* reduction variable */
03883             if (!var_in_struct(e, s->level))
03884               spitcode("%s *_omp_reduction_%s = &%s;\n", ltype, e->name, e->name);
03885               spitcode("%s %s = %s;\n",
03886                 ltype, e->name, reduction_init_val[e->flags.reduction_op[Directive->depth]]);
03887           }
03888           /* firstprivate initialization */
03889           if (e->flags.vis[Directive->depth] & V_FIRST) {
03890             if (e->dtype == DT_BASETYPE) {
03891               if (var_in_struct(e, s->level)) {
03892                 char dummy[STR_MAX_LEN];
03893
03894                 spitcode("%s %s = _OMP_VARREF(%s, %s);\n", ltype, e->name, ompc_func, e->name);
03895                 handle_identifier(dummy, id);
03896               }
03897               else
03898                 spitcode("%s %s = *_omp_firstlastprivate_%s;\n", ltype, e->name, id);
03899             }
03900             else {                                                          /* array */
03901               spitcode("%s %s;\n", ltype, e->name);
```

```
03902               if (var_in_struct(e, s->lsval)) {
03903                 char dummy[STR_MAX_LEN];
03904
03905                 sprintf(ist, "memcpy(%s, &_OMP_VARREF(%s, %s), sizeof(%s));\n",
03906                   id, ompc_func, id, id);
03907                 IST_SEEK();
03908                 handle_identifier(dummy, id);
03909               }
03910               else {
03911                 sprintf(ist, "memcpy(%s, _omp_firstlastprivate_%s, sizeof(%s));\n", id, id, id);
03912                 IST_SEEK();
03913               }
03914               if (e->flags.arg)
03915                 show_warning("  firstprivate %s might cause invalid declaration\n", e->name);
03916             }
03917           }
03918         }
03919       }
03920 }
03921
03922 void spit_reduction_code()
03923 {
03924   scope *s;
03925   entity *e;
03926   char id[STR_MAX_LEN] = {0};
03927   char dummy[STR_MAX_LEN];
03928   reduction_op_t rop;
03929
03930   if (Directive->has_reduction != 1)
03931     return;
03932   spitcode("othread_set_lock(&_omp_module.reduction_lock[%d]);\n", num_reduction);
03933   for (s = Scope; s != NULL; s = s->next)
03934     for (e = s->head; e != NULL; e = e->next) {
03935       /* reduction var assignment */
03936       rop = e->flags.reduction_op[Directive->depth];
03937       if (rop != R_NONE) {
03938         strncpy(id, &e->name[e->id_h], e->id_l); id[e->id_l] = 0;
03939         if (var_in_struct(e, s->level)) {
03940           if (rop == R_LAND || rop == R_LOR)
03941             spitcode("_OMP_VARREF(%s, %s) = _OMP_VARREF(%s, %s) %s %s;\n",
03942               ompc_func, id, ompc_func, id, reduction_op_str[rop], id);
03943           else
03944             if (rop == R_SUB)
03945               spitcode("_OMP_VARREF(%s, %s) += %s;\n", ompc_func, id, id);
03946             else
03947               spitcode("_OMP_VARREF(%s, %s) %s= %s;\n", ompc_func, id, reduction_op_str[rop], id);
03948           handle_identifier(dummy, id);
03949         }
03950         else {
03951           if (rop == R_LAND || rop == R_LOR)
03952             spitcode("*_omp_reduction_%s = *_omp_reduction_%s %s %s;\n",
03953               id, id, reduction_op_str[rop], id);
03954           else
03955             if (rop == R_SUB)
03956               spitcode("*_omp_reduction_%s += %s;\n", id, id);
03957             else
03958               spitcode("*_omp_reduction_%s %s= %s;\n", id, reduction_op_str[rop], id);
03959         }
03960       }
03961     }
03962   spitcode("othread_unset_lock(&_omp_module.reduction_lock[%d]);\n", num_reduction);
03963   Directive->has_reduction = 2;
03964 }
03965
03966 void spit_lastprivate_assign()
03967 {
03968   scope *s;
03969   entity *e;
03970   char id[STR_MAX_LEN] = {0};
03971   char dummy[STR_MAX_LEN];
03972
03973   for (s = Scope; s != NULL; s = s->next)
03974     for (e = s->head; e != NULL; e = e->next) {
03975       /* lastprivate assignment */
03976       if (e->flags.vis[Directive->depth] & V_LAST) {
03977         strncpy(id, &e->name[e->id_h], e->id_l); id[e->id_l] = 0;
03978         if (e->dtype == DT_BASETYPE) {
```

```
03979           if (var_in_struct(e, s->level)) {
03980             spitcode("_OMP_VARREF(%s, %s) = %s;\n", ompc_func, id, id);
03981             handle_identifier(dummy, id);
03982           }
03983           else
03984             spitcode("*_omp_firstlastprivate_%s = %s;\n", id, id);
03985         }
03986         else                                                          /* array */
03987           if (var_in_struct(e, s->level)) {
03988             spitcode("memcpy(&_OMP_VARREF(%s, %s), %s, sizeof(%s));\n", ompc_func, id, id, id);
03989             handle_identifier(dummy, id);
03990           }
03991           else
03992             spitcode("memcpy(_omp_firstlastprivate_%s, %s, sizeof(%s));\n", id, id, id);
03993       }
03994     }
03995 }
03996
03997 void spit_copyprivate_store()
03998 {
03999   scope *s;
04000   entity *e;
04001   char id[STR_MAX_LEN] = {0};
04002
04003   spitcode("_omp_cpriv_size = 0;\n");
04004   for (s = Scope; s != NULL; s = s->next)
04005     for (e = s->head; e != NULL; e = e->next) {
04006       if (e->flags.cpriv[Directive->depth]) {
04007         strncpy(id, &e->name[e->id_h], e->id_l); id[e->id_l] = 0;
04008         spitcode("_omp_cpriv_size += sizeof(%s);\n", id);
04009       }
04010     }
04011   spitcode(
04012     "_omp_thp->copyprivate.data = (char *) _omp_alloc(_omp_cpriv_size);\n"
04013     "_omp_thp->copyprivate.owner = othread_self();\n\n"
04014     "_omp_thp->copyprivate.del_counter = _omp_thp->num_children;\n"
04015     "_omp_cpriv_size = 0;\n");
04016
04017   for (s = Scope; s != NULL; s = s->next)
04018     for (e = s->head; e != NULL; e = e->next) {
04019       if (e->flags.cpriv[Directive->depth]) {
04020         strncpy(id, &e->name[e->id_h], e->id_l); id[e->id_l] = 0;
04021         if (e->flags.threadprivate) {
04022           if (e->dtype == DT_BASETYPE)
04023             spitcode(
04024               "memcpy(_omp_thp->copyprivate.data+_omp_cpriv_size, "
04025               "&((*_omp_tp_vars)[_OMP_THREAD->real_thread_num].%s), sizeof(%s));\n", e->new_id, id);
04026           else
04027             spitcode(
04028               "memcpy(_omp_thp->copyprivate.data+_omp_cpriv_size, "
04029               "(*_omp_tp_vars)[_OMP_THREAD->real_thread_num].%s, sizeof(%s));\n", e->new_id, id);
04030           spitcode("_omp_cpriv_size += sizeof(%s);\n", id);
04031         }
04032         else {
04033           if (e->dtype == DT_BASETYPE)
04034             spitcode("memcpy(_omp_thp->copyprivate.data+_omp_cpriv_size, &(%s), sizeof(%s));\n",
04035               id, id);
04036           else
04037             spitcode("memcpy(_omp_thp->copyprivate.data+_omp_cpriv_size, %s, sizeof(%s));\n",
04038               id, id);
04039           spitcode("_omp_cpriv_size += sizeof(%s);\n", id);
04040         }
04041       }
04042     }
04043 }
04044
04045 void spit_copyprivate_assign()
04046 {
04047   scope *s;
04048   entity *e;
04049   char id[STR_MAX_LEN] = {0};
04050
04051   spitcode("if (_omp_thp->copyprivate.owner != othread_self()) {\n");
04052   spitcode("_omp_cpriv_size = 0;\n");
04053   for (s = Scope; s != NULL; s = s->next)
04054     for (e = s->head; e != NULL; e = e->next) {
04055       if (e->flags.cpriv[Directive->depth]) {
```

```
04056         strncpy(id, &e->name[e->id_h], e->id_l); id[e->id_l] = 0;
04057         if (e->flags.threadprivate) {
04058           if (e->dtype == DT_BASETYPE)
04059             spitcode("memcpy(&((*_omp_tp_vars)[_OMP_THREAD->real_thread_num].%s), "
04060               "_omp_thp->copyprivate.data+_omp_cpriv_size, sizeof(%s));\n", e->new_id, id);
04061           else
04062             spitcode("memcpy((*_omp_tp_vars)[_OMP_THREAD->real_thread_num].%s, "
04063               "_omp_thp->copyprivate.data+_omp_cpriv_size, sizeof(%s));\n", e->new_id, id);
04064           spitcode("_omp_cpriv_size += sizeof(%s);\n", id);
04065         }
04066         else {
04067           if (e->dtype == DT_BASETYPE)
04068             spitcode("memcpy(&(%s), _omp_thp->copyprivate.data+_omp_cpriv_size, sizeof(%s));\n",
04069               id, id);
04070           else
04071             spitcode("memcpy(%s, _omp_thp->copyprivate.data+_omp_cpriv_size, sizeof(%s));\n",
04072               id, id);
04073           spitcode("_omp_cpriv_size += sizeof(%s);\n", id);
04074         }
04075       }
04076     }
04077   spitcode("}\n");
04078
04079   spitcode("othread_set_lock(&_omp_thp->copyprivate.lock);\n");
04080   spitcode("_omp_thp->copyprivate.del_counter--;\n");
04081   spitcode("if (_omp_thp->copyprivate.del_counter == 0)  /* the last thread must free memory */\n"
04082            "free(_omp_thp->copyprivate.data);\n");
04083   spitcode("othread_unset_lock(&_omp_thp->copyprivate.lock);\n");
04084 }
04085
04086 /**
04087   malloc() with error handling. Should not be called directly, use SAFE_MALLOC macro instead.
04088  */
04089 void *safe_malloc(size_t size, char *file_name, int line_number)
04090 {
04091   void *p = malloc(size);
04092
04093   if (p == NULL) {
04094     fprintf(stderr, "ERROR: out of memory at file %s, line %d.\n", file_name, line_number);
04095     _exit(10);
04096   }
04097   return p;
04098 }
04099
04100 /**
04101   strdup() with error handling. Should not be called directly, use SAFE_STRDUP macro instead.
04102  */
04103 char *safe_strdup(char *s, char *file_name, int line_number)
04104 {
04105   int len = strlen(s) + 1;                          /* including the terminating NULL character */
04106   void *p = malloc(len);
04107
04108   if (p == NULL) {
04109     fprintf(stderr, "ERROR: out of memory at file %s, line %d.\n", file_name, line_number);
04110     _exit(10);
04111   }
04112   memcpy(p, s, len);
04113
04114   return p;
04115 }
04116
04117 void show_error(char *format, ...)
04118 {
04119   va_list ap;
04120
04121   va_start(ap, format);
04122   fprintf(stderr, "error in file %s at line %d:\n", orig_fname, line_no - new_line + orig_line);
04123   vfprintf(stderr, format, ap);
04124   va_end(ap);
04125
04126   exit(1);
04127 }
04128
04129 void show_warning(char *format, ...)
04130 {
04131   va_list ap;
04132
```

```
04133   va_start(ap, format);
04134   fprintf(stderr, "warning: in file %s at line %d:\n", orig_fname, line_no - new_line + orig_line);
04135   vfprintf(stderr, format, ap);
04136   va_end(ap);
04137 }
04138
04139 FILE *spit_in_which_file()
04140 {
04141   FILE *f;
04142
04143   f = in_parallel ? thread_fun_fp : out_fp;
04144   if (is_typedef || (is_seu && block_level == 0))
04145     f = typedef_fp;
04146
04147   return f;
04148 }
04149
04150 void spitcode(char *format, ...)
04151 {
04152   va_list ap;
04153
04154   va_start(ap, format);
04155 /*  fprintf(stderr, "==> %d %d\n", is_seu, block_level);*/
04156   vfprintf(spit_in_which_file(), format, ap);
04157   va_end(ap);
04158 }
04159
04160 /**
04161   Spits code in "spit_in_which_file()" but only if instrumentation is enabled.
04162  */
04163 void pomp_spit_code(char *format, ...)
04164 {
04165   if (pomp_instrument) {
04166     va_list ap;
04167
04168     va_start(ap, format);
04169     vfprintf(spit_in_which_file(), format, ap);
04170     va_end(ap);
04171   }
04172 }
04173
04174 /**
04175   Spits code in "spit_in_which_file()" but only if instrumentation for this region is enabled.
04176  */
04177 void pomp_spit_region_code(pomp_region pr, char *format)
04178 {
04179   static char rd_name[20];
04180
04181   if (!pomp_rd_stack)  /* if no instrumented pomp region was entered, there is nothing to monitor */
04182     return;
04183   if (pomp_depth != pomp_rd_stack->depth)
04184     return;        /* this happens if instrumentation was turned off somewhere "after" pomp_depth */
04185 /* Don't check pomp_region_inst_on, it was checked at pomp_region_enter / exit */
04186   sprintf(rd_name, "omp_rd_%d", pomp_rd_stack->index);
04187   spitcode(format, rd_name);
04188 }
04189
04190 /**
04191   Returns TRUE if instrumentation for this region type is enabled.
04192  */
04193 int pomp_region_instr_on(pomp_region pr)
04194 {
04195   return pomp_instrument                                    /* Instrumentation is on */
04196     && (pomp_disable & (1 << pr)) == 0;          /* and the specific region type is not disabled. */
04197 }
04198
04199 /*
04200  It is not possible to declare the descriptors as static and spit the code at once, because:
04201    a) The "ending source code line" is not known at declaration time
04202    b) For parallel constructs, the code is moved into a function and it cannot access the
04203       original descriptor unless it is declared globally.
04204  So we have to
04205    a) Keep all the descriptors in memory, to update the "ending source code lines"
04206    b) Keep a STACK of the encountered directives, so that when a "FOR" closes we know that
04207       the implicit barrier NEXT spitted refers to #pragma omp parallel...
04208  But using just a stack has the disadvantage that each descriptor is "spitted" when it is "leaving"
04209  the stack, so the are declared in "first closing" order. Furthermore, a lot of variable names have
```

```
04210  to be generated, making the produced code more difficult to read.
04211  The solution is to create an array of descriptors, omp_rd_[]. Descriptors are placed into the
04212  array in order of the respective #pragma appearance.
04213  So all descriptors should be in memory while parsing, and interconnected with a stack AND a list.
04214  To preserve some memory, only "sub_name" should be malloc'ed, the other "char *" members should
04215  point in constant strings (or, in case of file_name, the same string).
04216  NOTE: following the OPARI convensions, a collection of variables are spitted, named omp_rd_XXX.
04217 */
04218
04219 #ifdef DEBUGGING
04220 void report_stack(char *msg)
04221 {
04222   fprintf(stderr, "%s\n", msg);
04223   fprintf(stderr, "pomp_depth = %d, ", pomp_depth);
04224   if (!pomp_rd_stack)
04225     fprintf(stderr, "stack is empty\n");
04226   else {
04227     fprintf(stderr, "name = %s, depth = %d\n",
04228       pomp_region_name[pomp_rd_stack->pr], pomp_rd_stack->depth);
04229   }
04230 }
04231 #else
04232 #define report_stack(msg)
04233 #endif
04234
04235 /**
04236   Called upon pomp region entrance. Creates a new region descriptor, initializes it, links it into
04237   rd list and stack, and spits the respective function call.
04238  */
04239 void pomp_region_enter(pomp_region pr, char *sub_name)
04240 {
04241   struct ompregdescr_parsing *rd;
04242   static char function_call[1024];
04243
04244   pomp_depth++;                          /* count depth for ALL pomp regions, even if not monitoring */
04245   if (pr == PR_USER && strcmp(sub_name, "main") == 0) {        /* Always instrument main, because */
04246     if (pomp_sync)                                  /* the EXPERT tools needs a single root, except */
04247       return;                                        /* if instrumentation is completely turned off */
04248   }
04249   else if (!pomp_region_instr_on(pr))          /* If instrumentation for this region type is off, */
04250     return;                                                                  /* then do nothing */
04251
04252   rd = (struct ompregdescr_parsing *) SAFE_MALLOC(sizeof(struct ompregdescr_parsing));
04253   memset(rd, 0, sizeof(struct ompregdescr_parsing));
04254   rd->index = ++pomp_regions;
04255   rd->pr = pr;
04256   rd->depth = pomp_depth;
04257   rd->begin_line = line_no - new_line + orig_line - 1;                  /* source code line number */
04258   if (sub_name)                                                 /* if a sub_name is defined, copy it */
04259     if (sub_name[0] != '\0')
04260       rd->sub_name = SAFE_STRDUP(sub_name);
04261
04262   if (pomp_rd_list_last)                                /* if this is not the first region encountered */
04263     pomp_rd_list_last->list_next = rd;
04264   else
04265     pomp_rd_list_root = rd;
04266   pomp_rd_list_last = rd;
04267   if (pomp_rd_stack)                                                /* if there is another "open" region */
04268     rd->stack_previous = pomp_rd_stack;
04269   pomp_rd_stack = rd;
04270   snprintf(function_call, sizeof(function_call), "  { POMP_%s(&%%s); {\n",
04271     pomp_region_enter_function[pr]);  /* &%%s becomes &%s for the following pomp_spit_region_code */
04272   pomp_spit_region_code(pr, function_call);
04273
04274   report_stack("pomp_region_enter:");
04275 }
04276
04277 /**
04278   Called upon pomp region exit. Checks if the last open region was indeed of pr type, fills the
04279   end_line field, updates rd stack and spits the respective function call.
04280  */
04281 void pomp_region_exit(pomp_region pr)
04282 {
04283   int do_instrument = pomp_region_instr_on(pr);
04284   static char function_call[1024];
04285
04286   report_stack("pomp_region_exit:");
```

```
04287 /* A region ends. Special cases:
04288   *) depth == pomp_rd_stack->depth and pr doesn't match pomp_rd_stack->pr => ERROR
04289   1) not instrumenting but region #start# was instrumented => spit warning & instrument
04290   2) instrumenting, no matching region #start# => spit warning & don't instrument
04291  */
04292   if (pomp_rd_stack)
04293     if (pomp_depth == pomp_rd_stack->depth && pr != pomp_rd_stack->pr)
04294       show_error(
04295         "POMP REGION STACK ERROR: A region of type [%s] ended while expecting\n"
04296         "the end of the region of type [%s] which started at line %d.\n",
04297         pomp_region_name[pr], pomp_region_name[pomp_rd_stack->pr], pomp_rd_stack->begin_line);
04298   if (do_instrument) {
04299     if (!pomp_rd_stack || pomp_depth != pomp_rd_stack->depth) {
04300       show_warning(
04301         "POMP: A region of type [%s] ended while instrumentation was ON, but \n"
04302         "instrumentation was OFF at region entrance. Instrumentation code will NOT be written.\n",
04303         pomp_region_name[pr]);
04304       do_instrument = 0;
04305     }
04306   }
04307   else {
04308     if (pomp_rd_stack)
04309       if (pomp_depth == pomp_rd_stack->depth) {
04310         show_warning(
04311           "POMP: A region of type [%s] was started at line %d while instrumentation was ON,\n"
04312           "but instrumentation was OFF at region exit. Instrumentation code will be written.\n",
04313           pomp_region_name[pomp_rd_stack->pr], pomp_rd_stack->begin_line);
04314         do_instrument = 1;
04315       }
04316   }
04317
04318   if (do_instrument) {
04319     pomp_rd_stack->end_line = line_no - new_line + orig_line;          /* source code line number */
04320     snprintf(function_call, sizeof(function_call), "  } POMP_%s(&%%s); }\n",
04321       pomp_region_exit_function[pr]); /* &%%s becomes &%s for the following pomp_spit_region_code */
04322     pomp_spit_region_code(pr, function_call);
04323     pomp_rd_stack = pomp_rd_stack->stack_previous;                    /* remove region from stack */
04324   }
04325
04326   pomp_depth--;
04327 }
04328
04329 /**
04330   Called at the end of parsing to spit all the rd variables.
04331  */
04332 void pomp_spit_regdescr(FILE *f)
04333 {
04334   struct ompregdescr_parsing *rd = pomp_rd_list_root;
04335   char *sn;
04336   int i;
04337
04338   if (pomp_sync)            /* if "--pomp-disable=sync" parameter was passed, don't spit anything */
04339     return;
04340   fprintf(f,
04341     "\n"
04342     "/*****************************************/\n"
04343     "/* POMP region descriptor declarations */\n"
04344     "/*****************************************/\n"
04345     "struct ompregdescr {\n"
04346     "  char* name;                 /* name of construct                 */\n"
04347     "  char* sub_name;             /* optional: region name             */\n"
04348     "  int   num_sections;         /* sections only: number of sections */\n"
04349     "  char* file_name;            /* source code location              */\n"
04350     "  int   begin_first_line;     /* line number first line opening pragma */\n"
04351     "  int   begin_last_line;      /* line number last  line opening pragma */\n"
04352     "  int   end_first_line;       /* line number first line closing pragma */\n"
04353     "  int   end_last_line;        /* line number last  line closing pragma */\n"
04354     "  void* data;                 /* space for performance data        */\n"
04355     "  struct ompregdescr* next;   /* for linking                       */\n"
04356     "};\n\n");
04357
04358   while (rd) {
04359     sn = rd->sub_name ? rd->sub_name : "";                    /* if sub_name is NULL, print "" */
04360     fprintf(f,
04361       "struct ompregdescr omp_rd_%d = {\n"
04362       "  \"%s\", \"%s\", %d, \"%s\", %d, %d, %d, %d, 0, 0};\n", rd->index,
04363       pomp_region_name[rd->pr], sn, rd->num_sections, orig_fname,
```

```
04364       rd->begin_line, rd->begin_line, rd->end_line, rd->end_line);
04365     rd = rd->list_next;
04366   }
04367
04368   fprintf(f,
04369     "int POMP_MAX_ID = %d;\n"
04370     "struct ompregdescr* pomp_rd_table[%d] = {",
04371     pomp_regions, pomp_regions);
04372
04373   for (i = 1; i < pomp_regions; i++) {
04374     if (i % 5 == 1)
04375       fprintf(f, "\n  ");
04376     fprintf(f, "&omp_rd_%d, ", i);
04377   }
04378   fprintf(f, "&omp_rd_%d};\n", pomp_regions);
04379 }
04380
04381 void free_entity_list(entity **e)
04382 {
04383   if (*e != NULL) {
04384     free_entity_list(&(*e)->next);
04385     free(*e);
04386     *e = NULL;
04387   }
04388 }
04389
04390 void open_directive_scope(directive_e directive_type)
04391 {
04392   scope *s;
04393   entity *e;
04394   directive_t *new_directive;
04395   int d;
04396
04397   new_directive = (directive_t *) SAFE_MALLOC(sizeof(directive_t));
04398
04399   new_directive->type = directive_type;
04400   new_directive->defshared = 1;
04401   new_directive->has_reduction = 0;
04402   new_directive->has_nowait = 0;
04403   new_directive->is_parallel = 0;
04404   new_directive->copyin = -1;
04405   new_directive->depth = d = Directive->depth + 1;
04406   new_directive->block_level = block_level;
04407   strcpy(new_directive->if_expr, "#");
04408   strcpy(new_directive->numth_expr, "-1");
04409   new_directive->next = Directive;
04410   Directive = new_directive;
04411
04412   for (s = Scope; s != NULL; s = s->next)
04413     for (e = s->head; e != NULL; e = e->next) {
04414       e->flags.used[d] = 0;
04415       if (directive_type != D_PARALLEL)
04416         e->flags.vis[d] = e->flags.vis[d-1];
04417       else
04418         e->flags.vis[d] = V_NONE;
04419       e->flags.reduction_op[d] = R_NONE;
04420     }
04421 }
04422
04423 void close_directive_scope()
04424 {
04425   directive_t *t;
04426   scope *s;
04427   entity *e;
04428
04429   num_reduction += Directive->has_reduction != 0;
04430   t = Directive;
04431   Directive = Directive->next;
04432   free(t);
04433   if (Directive->depth == 0)
04434     for (s = Scope; s != NULL; s = s->next)
04435       for (e = s->head; e != NULL; e = e->next)
04436         e->flags.var_reffed = 0;
04437 }
04438
04439 void open_scope()
04440 {
```

```
04441   scope *new_scope;
04442   new_scope = (scope *) SAFE_MALLOC(sizeof(scope));
04443
04444   if (func_args != NULL) {
04445  /* entity *e;
04446
04447     fprintf(stderr, "%s----------\n", func_name);
04448     for (e = func_args; e != NULL; e = e->next)
04449       fprintf(stderr, "->{%s|||%s}",e->type,e->name);
04450     fprintf(stderr, "\n");
04451  */
04452     new_scope->head = func_args;
04453     func_args = NULL;
04454   }
04455   else
04456     new_scope->head = NULL;
04457   in_func_def = 0;
04458   new_scope->level = block_level;
04459   new_scope->next = Scope;
04460   Scope = new_scope;
04461   open_type_scope();
04462 }
04463
04464 void close_scope()
04465 {
04466   entity *p, *q;
04467   scope *t;
04468
04469   p = Scope->head;
04470   while (p != NULL) {
04471     q = p;
04472     p = p->next;
04473     if (q->new_id != NULL)
04474       free(q->new_id);
04475     free(q);
04476   }
04477
04478   t = Scope;
04479   Scope = Scope->next;
04480   free(t);
04481   close_type_scope();
04482 }
04483
04484 void add_entity(entity **dest, int arg, char *type, char *name, int head, int len, int dtype)
04485 {
04486   entity *p;
04487   p = (entity *) SAFE_MALLOC(sizeof(entity));
04488
04489   strcpy(p->name, name);
04490   strcpy(p->type, type);
04491   p->new_id = NULL;
04492   p->flags.used[Directive->depth] = 0;
04493   p->flags.vis[Directive->depth]  = V_NONE;
04494   p->flags.cpriv[Directive->depth] = 0;
04495   p->flags.var_reffed = 0;
04496   p->flags.arg = arg;
04497   p->flags.reprivate = 0;
04498   p->flags.threadprivate = 0;
04499   p->flags.reduction_op[dec_depth] = R_NONE;
04500   p->id_h = head;
04501   p->id_l = len;
04502   p->dtype = dtype;
04503
04504   p->next = *dest;
04505   *dest = p;
04506 /*
04507   p->id_h = declared_var[dec_depth].id_h;
04508   p->id_l = declared_var[dec_depth].id_l;
04509   p->dtype = declared_var[dec_depth].type;
04510 */
04511
04512 /*
04513   p->next = Scope->head;
04514   Scope->head = p;
04515 */
04516
04517 /*
```

```
04518   {
04519     int i;
04520     for (i=0; i<=dec_depth+1; i++)
04521       fprintf(stderr, "===> %s\n", declared_var[i].name);
04522     fprintf(stderr, "====================>>>\n");
04523
04524   }
04525 */
04526
04527 /*
04528   fprintf(stderr, "%s ----> %s\n", p->name, type);
04529 */
04530
04531 /*
04532   fprintf(stderr, "name : %s\n"
04533                   "     : %*s\n"
04534                   "     : %*s\n",p->name,p->id_h+1, "^", p->id_h+p->id_l, "^");
04535   fprintf(stderr, "DEBUG: %d, %d\n\n", p->id_h, p->id_l);
04536 */
04537 }
04538
04539 entity *get_entity(char *name, int *level)
04540 {
04541   scope *s;
04542   entity *e;
04543
04544   for (s = Scope; s != NULL; s = s->next)
04545     for (e = s->head; e != NULL; e = e->next) {
04546       if (strncmp(name, &e->name[e->id_h], e->id_l) == 0 && !name[e->id_l]) {
04547         *level = s->level;
04548      /* {
04549           type_entity *t;
04550           fprintf(stderr, "entity=>%s<=\n", e->name);
04551           fprintf(stderr, "e-type=>%s<=\n", e->type);
04552           t = get_type_entity(e->type, level);
04553           if (t == NULL)
04554             fprintf(stderr, "e-tent=>NULL<=\n");
04555           else
04556             fprintf(stderr, "e-tent=>%s<=\n", t->name);
04557         }
04558       */
04559         return e;
04560       }
04561     }
04562
04563   return NULL;
04564 }
04565
04566 /**
04567   Open a scope for type names (typedefs) so that they can be moved to global scope with correct
04568   names.
04569  */
04570 void open_type_scope()
04571 {
04572   type_scope *new_scope;
04573   int_list *b;
04574
04575   new_scope = (type_scope *) SAFE_MALLOC(sizeof(type_scope));
04576   new_scope->head = NULL;
04577   new_scope->level = block_level;
04578   new_scope->next = Type_Scope;
04579   Type_Scope = new_scope;
04580
04581   b = (int_list *) SAFE_MALLOC(sizeof(int_list));
04582   b->count = 0;
04583   b->next = NULL;
04584   b->prev = bt_tail;
04585   bt_tail->next = b;
04586   bt_tail = b;
04587 }
04588
04589 void close_type_scope()
04590 {
04591   type_entity *p, *q;
04592   type_scope *t;
04593
04594   p = Type_Scope->head;
```

```
04595   while (p != NULL) {
04596     q = p;
04597     p = p->next;
04598     free(q);
04599   }
04600
04601   t = Type_Scope;
04602   Type_Scope = Type_Scope->next;
04603   free(t);
04604
04605   bt_tail = bt_tail->prev;
04606   free(bt_tail->next);
04607   bt_tail->next = NULL;
04608   bt_tail->count++;
04609 }
04610
04611 void gen_unique_name(char *dest, char *src)
04612 {
04613   char num[16], *s, *t;
04614   int_list *b;
04615
04616   if (bt_head == bt_tail)
04617     strcpy(dest, src);
04618   else {
04619     for (s = dest, t = src; *t; *s++ = *t++)
04620       ;
04621     *s++ = '_';
04622     for (t = func_name; *t; *s++ = *t++);
04623     for (b = bt_head->next; b != bt_tail; b = b->next) {
04624       *s++ = '_';
04625       sprintf(num, "%d", b->count);
04626       for (t = num; *t; *s++ = *t++);
04627     }
04628     *s = 0;
04629   }
04630 }
04631
04632 void add_type_entity(char *name)
04633 {
04634   type_entity *p;
04635
04636   p = (type_entity *) SAFE_MALLOC(sizeof(type_entity));
04637   strcpy(p->name, name);
04638   gen_unique_name(p->new_name, name);
04639 /* fprintf(stderr, "==> %s\n", p->new_name); */
04640   p->next = Type_Scope->head;
04641   Type_Scope->head = p;
04642 }
04643
04644 type_entity *get_type_entity(char *name, int *level)
04645 {
04646   type_scope *s;
04647   type_entity *e;
04648
04649   for (s = Type_Scope; s != NULL; s = s->next)
04650     for (e = s->head; e != NULL; e = e->next) {
04651       if (strcmp(name, e->name) == 0) {
04652         *level = s->level;
04653         return e;
04654       }
04655     }
04656
04657   return NULL;
04658 }
04659
04660 void print_Scope()
04661 {
04662   scope *s;
04663   entity *e;
04664
04665   fprintf(stderr, "\n\n\n");
04666   for (s = Scope; s != NULL; s = s->next) {
04667     fprintf(stderr, "%d:",s->level);
04668     for (e = s->head; e != NULL; e = e->next)
04669       fprintf(stderr, "->{%s|||%s}", e->type, e->name);
04670     fprintf(stderr, "\n");
04671   }
```

```
04672 }
04673
04674 int yyerror(char *s)
04675 {
04676   fprintf(stderr, "error in file %s at line %d: %s\n",
04677     orig_fname, line_no - new_line + orig_line, s);
04678
04679   return 1;
04680 }
04681
04682 void svars_pass()
04683 {
04684   char buffer[8192];
04685
04686   rewind(thread_fun_fp);
04687   rewind(shared_var_fp);
04688
04689   while (fgets(buffer, sizeof(buffer), thread_fun_fp) != NULL) {
04690     if (strncmp(buffer, "#OMP_VARS", 9) == 0) {
04691       /* shared variable declaration */
04692       while (fgets(buffer, sizeof(buffer), shared_var_fp) != NULL) {
04693         if (buffer[0] == '#')
04694           break;
04695         fputs(buffer, thread_fun_fp2);
04696       }
04697     }
04698     else
04699       fputs(buffer, thread_fun_fp2);
04700   }
04701 }
04702
04703 void second_pass()
04704 {
04705   char buffer[8192];
04706   char inc_file[32];
04707   FILE *f;
04708
04709   rewind(out_fp);
04710   while (fgets(buffer, sizeof(buffer), out_fp) != NULL)
04711     if (sscanf(buffer, "#OMP_INCLUDE %s", inc_file) == 1) {
04712       f = fopen(inc_file,"r");
04713       while (fgets(buffer, sizeof(buffer), f) != NULL)
04714         fputs(buffer, stdout);
04715       fclose(f);
04716     }
04717     else
04718       fputs(buffer, stdout);
04719 }
04720
04721 int main(int argc, char **argv)
04722 {
04723   int yyparse();
04724   int r;
04725   char basename[STR_MAX_LEN] = {0};
04726   char fname[STR_MAX_LEN];
04727   struct timeval ts;
04728 #if YYDEBUG == 1
04729   extern int yydebug;
04730
04731   yydebug = 0;
04732 #endif
04733
04734   if (argc != 4 || (argc == 4 && strcmp(argv[2], "__ompi__") != 0))  {
04735     fprintf(stderr, "OMPI: this file should not be run directly, use `ompicc' instead\n");
04736     _exit(20);
04737   }
04738   pomp_disable = atoi(argv[3]);                    /* get pomp_disable_mask argument from ompicc */
04739   if (pomp_disable == -1) {                      /* if "--pomp-disable=sync" was passed as argument */
04740     pomp_instrument = 0;                                          /* then disable optimization */
04741     pomp_sync = 1;              /* and mark pomp_sync so that pomp_instrument never gets ON */
04742   }
04743   yyin = fopen(argv[1], "r");
04744   if (yyin == NULL) {
04745     fprintf(stderr, "OMPI: could not open %s\n", argv[1]);
04746     _exit(30);
04747   }
04748
```

```
04749   gettimeofday(&ts, NULL);
04750   sprintf(module_id, "%X%X", ts.tv_sec, ts.tv_usec);
04751
04752   strncpy(basename, argv[1], strlen(argv[1])-2);
04753
04754   sprintf(fname, "%s.c0", basename);
04755   out_fp = fopen(fname, "w+");
04756   thread_fun_fp2 = fopen("funcs.c", "w");
04757   par_var_fp = fopen("vars.h", "w");
04758   modh_fp = fopen("mod.h", "w");
04759   thread_fun_fp_h = fopen("funcs.h", "w");
04760   shared_var_fp = fopen("svars.dat", "w+");
04761   typedef_fp = fopen("types.h", "w+");
04762   threadp_fp = fopen("tpvars.h", "w+");
04763   copyin_fp = fopen("copyin.c", "w+");
04764   copyin_fp_h = fopen("copyin.h", "w+");
04765   thread_fun_fp = fopen("threadfuncs.c0", "w+");
04766
04767   Directive = (directive_t *) SAFE_MALLOC(sizeof(directive_t));
04768   Directive->type = D_NONE;
04769   Directive->defshared = 1;
04770   Directive->depth = 0;
04771   Directive->next = NULL;
04772
04773   Scope = (scope *) SAFE_MALLOC(sizeof(scope));
04774   Scope->level = 0;
04775   Scope->head = NULL;
04776   Scope->next = NULL;
04777
04778   Type_Scope = (type_scope *) SAFE_MALLOC(sizeof(type_scope));
04779   Type_Scope->level = 0;
04780   Type_Scope->head = NULL;
04781   Type_Scope->next = NULL;
04782
04783 /* strlist_init(&idlist); */
04784
04785   bt_head = bt_tail = (int_list *) SAFE_MALLOC(sizeof(int_list));
04786   bt_head->count = 0;
04787   bt_head->next = NULL;
04788   bt_head->prev = NULL;
04789
04790   sec_head = sec_tail = (int_list *) SAFE_MALLOC(sizeof(int_list));
04791   sec_head->count = -1;
04792   sec_head->next = NULL;
04793   sec_head->prev = NULL;
04794
04795   spitcode("\n\n/* user-defined types */\n\n");
04796   spitcode("#OMP_INCLUDE types.h\n");
04797   spitcode("\n\n/**********************/\n\n");
04798
04799   spitcode("\n\n/* runtime library header file */\n\n");
04800   spitcode("#include <ompi.h>\n");
04801   spitcode("\n\n/**********************/\n\n");
04802
04803   if (!pomp_sync)                              /* if instrumentation is not completely disabled */
04804     spitcode(
04805       "\n"
04806       "/* Definitions and macros to replace omp_*_lock() function calls with POMP_*_lock(). */\n"
04807       "extern void POMP_Init_lock(omp_lock_t *s);\n"
04808       "extern void POMP_Destroy_lock(omp_lock_t *s);\n"
04809       "extern void POMP_Set_lock(omp_lock_t *s);\n"
04810       "extern void POMP_Unset_lock(omp_lock_t *s);\n"
04811       "extern int  POMP_Test_lock(omp_lock_t *s);\n"
04812       "extern void POMP_Init_nest_lock(omp_nest_lock_t *s);\n"
04813       "extern void POMP_Destroy_nest_lock(omp_nest_lock_t *s);\n"
04814       "extern void POMP_Set_nest_lock(omp_nest_lock_t *s);\n"
04815       "extern void POMP_Unset_nest_lock(omp_nest_lock_t *s);\n"
04816       "extern int  POMP_Test_nest_lock(omp_nest_lock_t *s);\n"
04817
04818       "#define omp_init_lock POMP_Init_lock\n"
04819       "#define omp_destroy_lock POMP_Destroy_lock\n"
04820       "#define omp_set_lock POMP_Set_lock\n"
04821       "#define omp_unset_lock POMP_Unset_lock\n"
04822       "#define omp_test_lock POMP_Test_lock\n"
04823       "#define omp_init_nest_lock POMP_Init_nest_lock\n"
04824       "#define omp_destroy_nest_lock POMP_Destroy_nest_lock\n"
04825       "#define omp_set_nest_lock POMP_Set_nest_lock\n"
```

```
04826       "#define omp_unset_nest_lock POMP_Unset_nest_lock\n"
04827       "#define omp_test_nest_lock POMP_Test_nest_lock\n"
04828       "\n");
04829
04830   spitcode("\n\n/* OMPi-generated function prototypes */\n\n");
04831   spitcode("#OMP_INCLUDE funcs.h\n");
04832   spitcode("\n\n/***********************/\n\n");
04833
04834   spitcode("\n\n/* parallel construct shared variables */\n\n");
04835   spitcode("#OMP_INCLUDE vars.h\n");
04836   spitcode("\n\n/***********************/\n\n");
04837
04838   spitcode("\n\n/* threadprivate variables  */\n\n");
04839   spitcode("#OMP_INCLUDE tpvars.h\n");
04840   spitcode("\n\n/***********************/\n\n");
04841
04842   spitcode("\n\n/* module-specific macros */\n\n");
04843   spitcode("#OMP_INCLUDE mod.h\n");
04844   spitcode("\n\n/***********************/\n\n");
04845
04846   spitcode("\n\n/* copyin variable initialization functions (prototypes) */\n\n");
04847   spitcode("#OMP_INCLUDE copyin.h\n");
04848   spitcode("\n\n/***********************/\n\n");
04849
04850   fprintf(threadp_fp, "struct _omp_threadprivate_s%s {\n", module_id);
04851 #ifdef NO_EMPTY_STRUCT
04852   fprintf(threadp_fp, "int _omp_dummy_field; /* empty structs are not allowed */\n");
04853 #endif
04854
04855 /* for some gcc */
04856   add_type_entity("__builtin_va_list");
04857
04858 /*****************************************************************************/
04859
04860   r = yyparse();
04861
04862 /*****************************************************************************/
04863
04864   if (__has_omp == 0 && has_main == 0)
04865     return 33;
04866
04867   spitcode("\n\n/* OMPi-generated thread functions (moved code) */\n\n");
04868   spitcode("#OMP_INCLUDE funcs.c\n");
04869   spitcode("\n\n/***********************/\n\n");
04870
04871   spitcode("\n\n/* copyin variable initialization functions */\n\n");
04872   spitcode("#OMP_INCLUDE copyin.c\n");
04873   spitcode("\n\n/***********************/\n\n");
04874
04875   fprintf(threadp_fp, "};\n");
04876
04877   /* global threadprivate initialization */
04878   if (g_threadprivate > 0) {
04879     entity *e;
04880     char id[STR_MAX_LEN] = {0};
04881
04882     spitcode("\n/* global threadprivate initialization */\n");
04883     spitcode(
04884       "void _omp_init_gtp%s()\n"
04885       "{\n"
04886       "    int _omp_i;\n\n"
04887
04888       "    _omp_module.tp_vars_structsize = sizeof(struct _omp_threadprivate_s%s);\n"
04889       "    if (_omp_module.tp_vars_structsize > 0)\n"
04890       "        _omp_module.tp_vars = "
04891       "_omp_calloc(_omp_module.len_tp_vars * _omp_module.tp_vars_structsize);\n"
04892       "    for (_omp_i = 0; _omp_i < _omp_module.len_tp_vars; _omp_i++) {\n",
04893       module_id, module_id);
04894     for (e = Scope->head; e != NULL; e = e->next)
04895       if (e->flags.threadprivate) {
04896         strncpy(id, &e->name[e->id_h], e->id_l); id[e->id_l] = 0;
04897         if (e->dtype == DT_BASETYPE)
04898           spitcode("        (*_omp_tp_vars)[_omp_i].%s = %s;\n", e->new_id, id);
04899         else
04900           spitcode("        memcpy((*_omp_tp_vars)[_omp_i].%s, %s, sizeof(%s));\n",
04901             e->new_id, id, id);
04902       }
```

```
04903     spitcode("    }\n}\n");
04904   }
04905   else {
04906     spitcode("\n/* global threadprivate initialization */\n");
04907     spitcode("void _omp_init_gtp%s() { }\n", module_id);
04908   }
04909   fprintf(threadp_fp, "\nvoid _omp_init_gtp%s();\n", module_id);
04910
04911   num_sections++;
04912   num_for++;
04913   num_single++;
04914   num_parallel++;
04915
04916   if (num_sections > 0) {
04917     int i = 1;
04918
04919     sec_head = sec_head->next;
04920     fprintf(par_var_fp, "int _omp_num_section[%d] = { %d", num_sections, sec_head->count);
04921     for (sec_head = sec_head->next; sec_head != NULL; sec_head = sec_head->next) {
04922       fprintf(par_var_fp, ", %d", sec_head->count);
04923       i++;
04924       if (i % 32 == 0)
04925         fprintf(par_var_fp, "\n");
04926     }
04927     fprintf(par_var_fp, "};\n");
04928   }
04929
04930   fprintf(par_var_fp, "#define _OMP_NUM_SECTIONS %d\n", num_sections);
04931   fprintf(par_var_fp, "#define _OMP_NUM_FOR %d\n", num_for);
04932   fprintf(par_var_fp, "#define _OMP_NUM_SINGLE %d\n", num_single);
04933   fprintf(par_var_fp, "#define _OMP_NUM_REDUCTION %d\n", num_reduction);
04934   fprintf(par_var_fp, "#define _OMP_G_THREADPRIVATE %d\n", g_threadprivate);
04935
04936   fprintf(par_var_fp,
04937     "\n/* module initialization parameters */\n"
04938     "char _omp_mi_id_%s_;\n"
04939     "char _omp_mi_for_%d_;\n"
04940     "char _omp_mi_sin_%d_;\n"
04941     "char _omp_mi_sec_%d_;\n"
04942     "char _omp_mi_red_%d_;\n"
04943     "char _omp_mi_gtp_%d_;\n",
04944     module_id, num_for, num_single, num_sections, num_reduction, g_threadprivate);
04945
04946   pomp_spit_regdescr(par_var_fp);
04947
04948   fprintf(modh_fp, "#define _omp_module _omp_module%s\n", module_id);
04949   fprintf(modh_fp, "#define _omp_tp_vars _omp_tp_vars%s\n", module_id);
04950   fprintf(modh_fp, "_omp_mod_t _omp_module;\n\n");
04951   fprintf(modh_fp, "struct _omp_threadprivate_s%s **_omp_tp_vars = "
04952     "(struct _omp_threadprivate_s%s **) &_omp_module.tp_vars;\n", module_id, module_id);
04953
04954   fclose(modh_fp);
04955   fclose(par_var_fp);
04956   fclose(thread_fun_fp_h);
04957   fclose(typedef_fp);
04958   fclose(threadp_fp);
04959   fclose(copyin_fp_h);
04960   fclose(copyin_fp);
04961   fclose(yyin);
04962
04963   svars_pass();
04964
04965   fclose(thread_fun_fp);
04966   fclose(thread_fun_fp2);
04967   fclose(shared_var_fp);
04968
04969   second_pass();
04970   fclose(out_fp);
04971
04972   return r;
04973 }
```

ompi-0.8.2/ompi/scanner.l

```
00001 /*
```

```
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021 */
00022
00023 /*!
00024  \file scanner.l
00025  \brief Flex input file.
00026  */
00027
00028 D                       [0-9]
00029 L                       [a-zA-Z_]
00030 H                       [a-fA-F0-9]
00031 E                       [Ee][+-]?{D}+
00032 FS                      (f|F|l|L)
00033 IS                      (u|U|l|L)*
00034
00035 %{
00036 #include <stdio.h>
00037 #include <stdlib.h>
00038 #include "parser.h"
00039
00040 int __inomp = 0;
00041 extern int __has_omp;
00042
00043 void count();
00044 int pragma_omp();
00045 /*char *strcpy2(char *, char *);*/
00046
00047 extern int orig_line;
00048 extern char orig_fname[1024];
00049 extern int new_line;
00050 int line_no = 0;
00051
00052 extern void *get_type_entity(char *name, int *level);
00053
00054 %}
00055
00056 %%
00057 [ \t]*"#"[ \t]*"pragma"[ \t]+"omp"[ \t]+ { count(); return pragma_omp(); }
00058 [ \t]*"#"[ \t]*"pragma"[ \t]+"omp"[ \t]+"threadprivate"[ \t]* { count(); __inomp = 1; __has_omp = 1;
return PRAGMA_OMP_THREADPRIVATE; }
00059 [ \t]*"#"               { sharp(); }
00060 "parallel"              { count();
00061                         if (__inomp) return(OMP_PARALLEL);
00062                         else return check_type(); }
00063 "sections"              { count();
00064                         if (__inomp) return(OMP_SECTIONS);
00065                         else return check_type(); }
00066 "nowait"                { count();
00067                         if (__inomp) return(OMP_NOWAIT);
00068                         else return check_type(); }
00069 "ordered"               { count();
00070                         if (__inomp) return(OMP_ORDERED);
00071                         else return check_type(); }
00072 "schedule"              { count();
00073                         if (__inomp) return(OMP_SCHEDULE);
00074                         else return check_type(); }
00075 "dynamic"               { count();
00076                         if (__inomp) return(OMP_DYNAMIC);
00077                         else return check_type(); }
```

```
00078 "guided"              { count();
00079                         if (__inomp) return(OMP_GUIDED);
00080                         else return check_type(); }
00081 "runtime"             { count();
00082                         if (__inomp) return(OMP_RUNTIME);
00083                         else return check_type(); }
00084 "section"             { count();
00085                         if (__inomp) return(OMP_SECTION);
00086                         else return check_type(); }
00087 "single"              { count();
00088                         if (__inomp) return(OMP_SINGLE);
00089                         else return check_type(); }
00090 "master"              { count();
00091                         if (__inomp) return(OMP_MASTER);
00092                         else return check_type(); }
00093 "critical"            { count();
00094                         if (__inomp) return(OMP_CRITICAL);
00095                         else return check_type(); }
00096 "barrier"             { count();
00097                         if (__inomp) return(OMP_BARRIER);
00098                         else return check_type(); }
00099 "atomic"              { count();
00100                         if (__inomp) return(OMP_ATOMIC);
00101                         else return check_type(); }
00102 "flush"               { count();
00103                         if (__inomp) return(OMP_FLUSH);
00104                         else return check_type(); }
00105 "private"             { count();
00106                         if (__inomp) return(OMP_PRIVATE);
00107                         else return check_type(); }
00108 "firstprivate"            { count();
00109                         if (__inomp) return(OMP_FIRSTPRIVATE);
00110                         else return check_type(); }
00111 "lastprivate"             { count();
00112                         if (__inomp) return(OMP_LASTPRIVATE);
00113                         else return check_type(); }
00114 "shared"              { count();
00115                         if (__inomp) return(OMP_SHARED);
00116                         else return check_type(); }
00117 "none"                    { count();
00118                         if (__inomp) return(OMP_NONE);
00119                         else return check_type(); }
00120 "reduction"             { count();
00121                         if (__inomp) return(OMP_REDUCTION);
00122                         else return check_type(); }
00123 "copyin"              { count();
00124                         if (__inomp) return(OMP_COPYIN);
00125                         else return check_type(); }
00126 "num_threads"         { count();
00127                         if (__inomp) return(OMP_NUMTHREADS);
00128                         else return check_type(); }
00129 "copyprivate"         { count();
00130                         if (__inomp) return(OMP_COPYPRIVATE);
00131                         else return check_type(); }
00132 "inst"                { count();
00133                         if (__inomp) return (POMP_INST);
00134                         else return check_type(); }
00135 "init"                { count();
00136                         if (__inomp) return (POMP_INIT);
00137                         else return check_type(); }
00138 "finalize"            { count();
00139                         if (__inomp) return (POMP_FINALIZE);
00140                         else return check_type(); }
00141 "on"                    { count();
00142                         if (__inomp) return (POMP_ON);
00143                         else return check_type(); }
00144 "off"                     { count();
00145                         if (__inomp) return (POMP_OFF);
00146                         else return check_type(); }
00147 "begin"                   { count();
00148                         if (__inomp) return (POMP_BEGIN);
00149                         else return check_type(); }
00150 "end"                     { count();
00151                         if (__inomp) return (POMP_END);
00152                         else return check_type(); }
00153 "noinstrument"              { count();
00154                         if (__inomp) return (POMP_NOINSTRUMENT);
```

```
00155                           else return check_type(); }
00156 "instrument"                { count();
00157                           if (__inomp) return (POMP_INSTRUMENT);
00158                           else return check_type(); }
00159
00160 "auto"                        { count(); return(AUTO); }
00161 "break"                       { count(); return(BREAK); }
00162 "case"                        { count(); return(CASE); }
00163 "char"                        { count(); return(CHAR); }
00164 "const"                       { count(); return(CONST); }
00165 "continue"                  { count(); return(CONTINUE); }
00166 "default"                   { count();
00167                           if (__inomp) return(OMP_DEFAULT);
00168                           else return (DEFAULT); }
00169 "do"                          { count(); return(DO); }
00170 "double"                  { count(); return(DOUBLE); }
00171 "else"                        { count(); return(ELSE); }
00172 "enum"                        { count(); return(ENUM); }
00173 "extern"                  { count(); return(EXTERN); }
00174 "float"                       { count(); return(FLOAT); }
00175 "for"                       { count();
00176                           if (__inomp) return (OMP_FOR);
00177                           else return(FOR); }
00178 "goto"                        { count(); return(GOTO); }
00179 "if"                        { count();
00180                           if (__inomp) return(OMP_IF);
00181                           else return(IF); }
00182 "int"                       { count(); return(INT); }
00183 "long"                        { count(); return(LONG); }
00184 "register"                  { count(); return(REGISTER); }
00185 "return"                  { count(); return(RETURN); }
00186 "short"                       { count(); return(SHORT); }
00187 "signed"                  { count(); return(SIGNED); }
00188 "sizeof"                  { count(); return(SIZEOF); }
00189 "static"                  { count();
00190                           if (__inomp) return(OMP_STATIC);
00191                           else return (STATIC); }
00192 "struct"                  { count(); return(STRUCT); }
00193 "switch"                  { count(); return(SWITCH); }
00194 "typedef"                   { count(); return(TYPEDEF); }
00195 "union"                       { count(); return(UNION); }
00196 "unsigned"                  { count(); return(UNSIGNED); }
00197 "void"                      { count(); return(VOID); }
00198 "volatile"                  { count(); return(VOLATILE); }
00199 "while"                       { count(); return(WHILE); }
00200
00201 {L}({L}|{D})*                 { count(); return(check_type()); }
00202
00203 0[xX]{H}+{IS}?                  { count(); return(CONSTANT); }
00204 0{D}+{IS}?                  { count(); return(CONSTANT); }
00205 {D}+{IS}?                   { count(); return(CONSTANT); }
00206 '(\\.|[^\\'])+'                 { count(); return(CONSTANT); }
00207
00208 {D}+{E}{FS}?                  { count(); return(CONSTANT); }
00209 {D}*"."{D}+({E})?{FS}?          { count(); return(CONSTANT); }
00210 {D}+"."{D}*({E})?{FS}?          { count(); return(CONSTANT); }
00211
00212 \"(\\.|[^\\"])*\"           { count(); return(STRING_LITERAL); }
00213
00214 ">>="                         { count(); return(RIGHT_ASSIGN); }
00215 "<<="                         { count(); return(LEFT_ASSIGN); }
00216 "+="                        { count(); return(ADD_ASSIGN); }
00217 "-="                        { count(); return(SUB_ASSIGN); }
00218 "*="                        { count(); return(MUL_ASSIGN); }
00219 "/="                        { count(); return(DIV_ASSIGN); }
00220 "%="                        { count(); return(MOD_ASSIGN); }
00221 "&="                        { count(); return(AND_ASSIGN); }
00222 "^="                        { count(); return(XOR_ASSIGN); }
00223 "|="                        { count(); return(OR_ASSIGN); }
00224 ">>"                        { count(); return(RIGHT_OP); }
00225 "<<"                        { count(); return(LEFT_OP); }
00226 "++"                        { count(); return(INC_OP); }
00227 "--"                        { count(); return(DEC_OP); }
00228 "->"                        { count(); return(PTR_OP); }
00229 "&&"                        { count(); return(AND_OP); }
00230 "||"                        { count(); return(OR_OP); }
00231 "<="                        { count(); return(LE_OP); }
```

```
00232 ">="                      { count(); return(GE_OP); }
00233 "=="                      { count(); return(EQ_OP); }
00234 "!="                      { count(); return(NE_OP); }
00235 ";"                     { count(); return(';'); }
00236 "{"                     { count(); return('{'); }
00237 "}"                     { count(); return('}'); }
00238 ","                     { count(); return(','); }
00239 ":"                     { count(); return(':'); }
00240 "="                     { count(); return('='); }
00241 "("                     { count(); return('('); }
00242 ")"                     { count(); return(')'); }
00243 "["                     { count(); return('['); }
00244 "]"                     { count(); return(']'); }
00245 "."                     { count(); return('.'); }
00246 "&"                     { count(); return('&'); }
00247 "!"                     { count(); return('!'); }
00248 "~"                     { count(); return('~'); }
00249 "-"                     { count(); return('-'); }
00250 "+"                     { count(); return('+'); }
00251 "*"                     { count(); return('*'); }
00252 "/"                     { count(); return('/'); }
00253 "%"                     { count(); return('%'); }
00254 "<"                     { count(); return('<'); }
00255 ">"                     { count(); return('>'); }
00256 "^"                     { count(); return('^'); }
00257 "|"                     { count(); return('|'); }
00258 "?"                     { count(); return('?'); }
00259 "..."                     { count(); return(ELIPSIS); }
00260
00261 [ \t\v\f]                { count(); }
00262 \n                       { count();
00263                          if (__inomp) {
00264                              __inomp = 0;
00265                              return('\n');
00266                          }
00267                        }
00268 .                        { /* ignore bad characters */ }
00269
00270 %%
00271
00272 yywrap()
00273 {
00274         return(1);
00275 }
00276
00277 int column = 0;
00278
00279 sharp()
00280 {
00281         char c;
00282         char line[1024] = {0}, *s, *t;
00283         int l_line = 0;
00284         char l_fname[1024];
00285
00286         s = &line[0];
00287 /*        ECHO;*/
00288         while ((c = input()) != '\n' && c != 0) {
00289                 *s++ = c;
00290 /*                putchar(c);*/
00291         }
00292         line_no += c == '\n';
00293         if (c != 0) {
00294             *s = c;
00295 /*            putchar(c);*/
00296             sscanf(line, "%d", &l_line);
00297             for (s = line; *s != '\"' && *s != 0; s++);
00298             if (*s == '\"') {
00299                 t = l_fname;
00300                 for (s++; *s != '\"' && *s != 0; *t++ = *s++);
00301                 *t = 0;
00302             }
00303             if (l_line != 0 && t != l_fname) {
00304                 orig_line = l_line;
00305                 new_line = line_no;
00306                 strcpy(orig_fname, l_fname);
00307             } else printf("#%s", line);
00308             column = 0;
```

```
00309         } else printf("#%s", line);
00310 }
00311
00312 void count()
00313 {
00314   int i;
00315
00316   for (i = 0; yytext[i] != '\0'; i++)
00317     if (yytext[i] == '\n')
00318       column = 0, line_no++;
00319     else if (yytext[i] == '\t')
00320       column += 8 - (column % 8);
00321     else
00322       column++;
00323
00324 /* ECHO; */
00325   if (!isspace(yytext[0])) {
00326     strcpy(yylval.name, yytext);
00327   }
00328 }
00329
00330 int pragma_omp()
00331 {
00332 /* fprintf(stderr, "----------> PRAGMA_OMP\n"); */
00333   __inomp = 1;
00334   __has_omp = 1;
00335
00336   return (PRAGMA_OMP);
00337 }
00338
00339 /*
00340 char *strcpy2(char *t, char *s)
00341 {
00342   while (!(*s == '_' || *s >= 'a' && *s <='z' || *s >= 'A' && *s <='Z'))
00343     *s++;
00344   while (*s && (*s == '_' || *s >= 'a' && *s <='z' || *s >= 'A' && *s <='Z'
00345     || *s >= '0' && *s <= '9'))
00346     *t++ = *s++;
00347   *t = 0;
00348
00349   return s;
00350 }
00351 */
00352
00353 int check_type()
00354 {
00355     int level;
00356     return (get_type_entity(yytext, &level) != NULL) ? TYPE_NAME: IDENTIFIER;
00357 }
```

ompi-0.8.2/pomp/Makefile.am

```
00001 #
00002 #    OMPi OpenMP Compiler
00003 #    Copyright 2001-2003 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas
00004 #
00005 #    This file is part of OMPi.
00006 #
00007 #    OMPi is free software; you can redistribute it and/or modify
00008 #    it under the terms of the GNU General Public License as published by
00009 #    the Free Software Foundation; either version 2 of the License, or
00010 #    (at your option) any later version.
00011 #
00012 #    OMPi is distributed in the hope that it will be useful,
00013 #    but WITHOUT ANY WARRANTY; without even the implied warranty of
00014 #    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00015 #    GNU General Public License for more details.
00016 #
00017 #    You should have received a copy of the GNU General Public License
00018 #    along with OMPi; if not, write to the Free Software
00019 #    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00020 #
00021 ##   This file must be processed by automake
00022
00023 lib_LIBRARIES = libpomp.a
```

```
00024
00025 #ensure that pomp_lib.c "sees" omp.h
00026 AM_CFLAGS = @DEBUGFLAG@ -I../lib
00027
00028 libpomp_a_SOURCES = pomp_lib.c
00029
00030 # Headers to be installed in system places
00031 include_HEADERS = pomp_lib.h
```

ompi-0.8.2/pomp/pomp_lib.c

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021
00022   Portions of this code were take from OPARI Version 1.1, (C) 2001,
00023   Forschungszentrum Juelich, Zentralinstitut fuer Angewandte Mathematik
00024 */
00025
00026 /*! \file pomp_lib.c
00027     \brief Sample implementation of performance monitoring library.
00028     This implementation just fprintf's callee information, for normal usage libelg_pomp should
00029     be used (overwrite installed libpomp.a with libelg_pomp.a, so that ompicc links with it).
00030  */
00031
00032 #include <stdio.h>
00033 #include <stdlib.h>
00034 #include "../lib/omp.h"
00035 #include "pomp_lib.h"
00036
00037 /*
00038   [Alkis] Added print_regdescr and code to call it when needed.
00039   Prints the contents of the struct to stderr.
00040  */
00041 void print_regdescr(struct ompregdescr* r)
00042 {
00043   fprintf(stderr, " +>: r = %p", r);
00044   if (r != NULL)
00045     fprintf(stderr, " = {name=\"%s\", sub_name=\"%s\", num_sections=%d, begin=%d, end=%d}\n",
00046       r->name, r->sub_name, r->num_sections, r->begin_first_line, r->end_last_line);
00047   else
00048     fprintf(stderr, "\n");
00049 }
00050
00051 /*
00052  * Global variables
00053  */
00054
00055 int pomp_tracing = 0;
00056
00057 /*
00058  * C pomp function library
00059  */
00060
00061 void POMP_Finalize() {
00062   static int pomp_finalize_called = 0;
00063
00064   if ( ! pomp_finalize_called ) {
```

```
00065     pomp_finalize_called = 1;
00066
00067     fprintf(stderr, "  0: finalize\n");
00068   }
00069 }
00070
00071 void POMP_Init() {
00072   int i;
00073   static int pomp_init_called = 0;
00074
00075   if ( ! pomp_init_called ) {
00076     pomp_init_called = 1;
00077
00078     atexit(POMP_Finalize);
00079     fprintf(stderr, "  0: init\n");
00080
00081     for(i=0; i<POMP_MAX_ID; ++i) {
00082       if ( pomp_rd_table[i] ) {
00083         pomp_rd_table[i]->data = 0;   /* <-- allocate space for
00084                                          performance data here */
00085       }
00086     }
00087     pomp_tracing = 1;
00088   }
00089 }
00090
00091 void POMP_Off() {
00092   pomp_tracing = 0;
00093 }
00094
00095 void POMP_On() {
00096   pomp_tracing = 1;
00097 }
00098
00099 void POMP_Atomic_enter(struct ompregdescr* r) {
00100   if ( pomp_tracing ) {
00101     fprintf(stderr, "%3d: enter atomic\n", omp_get_thread_num());
00102     print_regdescr(r);
00103   }
00104 }
00105
00106 void POMP_Atomic_exit(struct ompregdescr* r) {
00107   if ( pomp_tracing ) {
00108     fprintf(stderr, "%3d: exit  atomic\n", omp_get_thread_num());
00109     print_regdescr(r);
00110   }
00111 }
00112
00113 void POMP_Barrier_enter(struct ompregdescr* r) {
00114   if ( pomp_tracing && r ) {
00115     if ( r->name[0] == 'b' )
00116       fprintf(stderr, "%3d: enter barrier\n", omp_get_thread_num());
00117     else
00118       fprintf(stderr, "%3d: enter implicit barrier of %s\n",
00119               omp_get_thread_num(), r->name);
00120     print_regdescr(r);
00121   }
00122 }
00123
00124 void POMP_Barrier_exit(struct ompregdescr* r) {
00125   if ( pomp_tracing && r ) {
00126     if ( r->name[0] == 'b' )
00127       fprintf(stderr, "%3d: exit  barrier\n", omp_get_thread_num());
00128     else
00129       fprintf(stderr, "%3d: exit  implicit barrier of %s\n",
00130               omp_get_thread_num(), r->name);
00131     print_regdescr(r);
00132   }
00133 }
00134
00135 void POMP_Flush_enter(struct ompregdescr* r) {
00136   if ( pomp_tracing ) {
00137     fprintf(stderr, "%3d: enter flush\n", omp_get_thread_num());
00138     print_regdescr(r);
00139   }
00140 }
00141
```

```
00142 void POMP_Flush_exit(struct ompregdescr* r) {
00143   if ( pomp_tracing ) {
00144     fprintf(stderr, "%3d: exit  flush\n", omp_get_thread_num());
00145     print_regdescr(r);
00146   }
00147 }
00148
00149 void POMP_Critical_begin(struct ompregdescr* r) {
00150   if ( pomp_tracing && r ) {
00151     fprintf(stderr, "%3d: begin critical %s\n",
00152             omp_get_thread_num(), r->sub_name);
00153     print_regdescr(r);
00154   }
00155 }
00156
00157 void POMP_Critical_end(struct ompregdescr* r) {
00158   if ( pomp_tracing && r ) {
00159     fprintf(stderr, "%3d: end   critical %s\n",
00160             omp_get_thread_num(), r->sub_name);
00161     print_regdescr(r);
00162   }
00163 }
00164
00165 void POMP_Critical_enter(struct ompregdescr* r) {
00166   if ( pomp_tracing && r ) {
00167     fprintf(stderr, "%3d: enter critical %s\n",
00168             omp_get_thread_num(), r->sub_name);
00169     print_regdescr(r);
00170   }
00171 }
00172
00173 void POMP_Critical_exit(struct ompregdescr* r) {
00174   if ( pomp_tracing && r ) {
00175     fprintf(stderr, "%3d: exit  critical %s\n",
00176             omp_get_thread_num(), r->sub_name);
00177     print_regdescr(r);
00178   }
00179 }
00180
00181 void POMP_For_enter(struct ompregdescr* r) {
00182   if ( pomp_tracing ) {
00183     fprintf(stderr, "%3d: enter for\n", omp_get_thread_num());
00184     print_regdescr(r);
00185   }
00186 }
00187
00188 void POMP_For_exit(struct ompregdescr* r) {
00189   if ( pomp_tracing ) {
00190     fprintf(stderr, "%3d: exit  for\n", omp_get_thread_num());
00191     print_regdescr(r);
00192   }
00193 }
00194
00195 void POMP_Master_begin(struct ompregdescr* r) {
00196   if ( pomp_tracing ) {
00197     fprintf(stderr, "%3d: begin master\n", omp_get_thread_num());
00198     print_regdescr(r);
00199   }
00200 }
00201
00202 void POMP_Master_end(struct ompregdescr* r) {
00203   if ( pomp_tracing ) {
00204     fprintf(stderr, "%3d: end   master\n", omp_get_thread_num());
00205     print_regdescr(r);
00206   }
00207 }
00208
00209 void POMP_Parallel_begin(struct ompregdescr* r) {
00210   if ( pomp_tracing ) {
00211     fprintf(stderr, "%3d: begin parallel\n", omp_get_thread_num());
00212     print_regdescr(r);
00213   }
00214 }
00215
00216 void POMP_Parallel_end(struct ompregdescr* r) {
00217   if ( pomp_tracing ) {
00218     fprintf(stderr, "%3d: end   parallel\n", omp_get_thread_num());
```

```
00219     print_regdescr(r);
00220   }
00221 }
00222
00223 void POMP_Parallel_fork(struct ompregdescr* r) {
00224   if ( pomp_tracing ) {
00225     fprintf(stderr, "%3d: fork  parallel\n", omp_get_thread_num());
00226     print_regdescr(r);
00227   }
00228 }
00229
00230 void POMP_Parallel_join(struct ompregdescr* r) {
00231   if ( pomp_tracing ) {
00232     fprintf(stderr, "%3d: join  parallel\n", omp_get_thread_num());
00233     print_regdescr(r);
00234   }
00235 }
00236
00237 void POMP_Section_begin(struct ompregdescr* r) {
00238   if ( pomp_tracing ) {
00239     fprintf(stderr, "%3d: begin section\n", omp_get_thread_num());
00240     print_regdescr(r);
00241   }
00242 }
00243
00244 void POMP_Section_end(struct ompregdescr* r) {
00245   if ( pomp_tracing ) {
00246     fprintf(stderr, "%3d: end   section\n", omp_get_thread_num());
00247     print_regdescr(r);
00248   }
00249 }
00250
00251 void POMP_Sections_enter(struct ompregdescr* r) {
00252   if ( pomp_tracing ) {
00253     fprintf(stderr, "%3d: enter sections\n", omp_get_thread_num());
00254     print_regdescr(r);
00255   }
00256 }
00257
00258 void POMP_Sections_exit(struct ompregdescr* r) {
00259   if ( pomp_tracing ) {
00260     fprintf(stderr, "%3d: exit  sections\n", omp_get_thread_num());
00261     print_regdescr(r);
00262   }
00263 }
00264
00265 void POMP_Single_begin(struct ompregdescr* r) {
00266   if ( pomp_tracing ) {
00267     fprintf(stderr, "%3d: begin single\n", omp_get_thread_num());
00268     print_regdescr(r);
00269   }
00270 }
00271
00272 void POMP_Single_end(struct ompregdescr* r) {
00273   if ( pomp_tracing ) {
00274     fprintf(stderr, "%3d: end   single\n", omp_get_thread_num());
00275     print_regdescr(r);
00276   }
00277 }
00278
00279 void POMP_Single_enter(struct ompregdescr* r) {
00280   if ( pomp_tracing ) {
00281     fprintf(stderr, "%3d: enter single\n", omp_get_thread_num());
00282     print_regdescr(r);
00283   }
00284 }
00285
00286 void POMP_Single_exit(struct ompregdescr* r) {
00287   if ( pomp_tracing ) {
00288     fprintf(stderr, "%3d: exit  single\n", omp_get_thread_num());
00289     print_regdescr(r);
00290   }
00291 }
00292
00293 void POMP_Workshare_enter(struct ompregdescr* r) {
00294   if ( pomp_tracing ) {
00295     fprintf(stderr, "%3d: enter workshare\n", omp_get_thread_num());
```

```
00296     print_regdescr(r);
00297   }
00298 }
00299
00300 void POMP_Workshare_exit(struct ompregdescr* r) {
00301   if ( pomp_tracing ) {
00302     fprintf(stderr, "%3d: exit  workshare\n", omp_get_thread_num());
00303     print_regdescr(r);
00304   }
00305 }
00306
00307 void POMP_Begin(struct ompregdescr* r) {
00308   if ( pomp_tracing && r ) {
00309     fprintf(stderr, "%3d: begin region %s\n",
00310             omp_get_thread_num(), r->sub_name);
00311     print_regdescr(r);
00312   }
00313 }
00314
00315 void POMP_End(struct ompregdescr* r) {
00316   if ( pomp_tracing && r ) {
00317     fprintf(stderr, "%3d: end   region %s\n",
00318             omp_get_thread_num(), r->sub_name);
00319     print_regdescr(r);
00320   }
00321 }
00322
00323 /*
00324  *---------------------------------------------------------------
00325  * C Wrapper for OpenMP API
00326  *---------------------------------------------------------------
00327  */
00328
00329 void POMP_Init_lock(omp_lock_t *s) {
00330   if ( pomp_tracing ) {
00331     fprintf(stderr, "%3d: init lock\n", omp_get_thread_num());
00332   }
00333   omp_init_lock(s);
00334 }
00335
00336 void POMP_Destroy_lock(omp_lock_t *s) {
00337   if ( pomp_tracing ) {
00338     fprintf(stderr, "%3d: destroy lock\n", omp_get_thread_num());
00339   }
00340   omp_destroy_lock(s);
00341 }
00342
00343 void POMP_Set_lock(omp_lock_t *s) {
00344   if ( pomp_tracing ) {
00345     fprintf(stderr, "%3d: set lock\n", omp_get_thread_num());
00346   }
00347   omp_set_lock(s);
00348 }
00349
00350 void POMP_Unset_lock(omp_lock_t *s) {
00351   if ( pomp_tracing ) {
00352     fprintf(stderr, "%3d: unset lock\n", omp_get_thread_num());
00353   }
00354   omp_unset_lock(s);
00355 }
00356
00357 int  POMP_Test_lock(omp_lock_t *s) {
00358   if ( pomp_tracing ) {
00359     fprintf(stderr, "%3d: test lock\n", omp_get_thread_num());
00360   }
00361   return omp_test_lock(s);
00362 }
00363
00364 void POMP_Init_nest_lock(omp_nest_lock_t *s) {
00365   if ( pomp_tracing ) {
00366     fprintf(stderr, "%3d: init nestlock\n", omp_get_thread_num());
00367   }
00368   omp_init_nest_lock(s);
00369 }
00370
00371 void POMP_Destroy_nest_lock(omp_nest_lock_t *s) {
00372   if ( pomp_tracing ) {
```

```
00370
00371 void POMP_Destroy_nest_lock(omp_nest_lock_t *s) {
00372   if ( pomp_tracing ) {
00373     fprintf(stderr, "%3d: destroy nestlock\n", omp_get_thread_num());
00374   }
00375   omp_destroy_nest_lock(s);
00376 }
00377
00378 void POMP_Set_nest_lock(omp_nest_lock_t *s) {
00379   if ( pomp_tracing ) {
00380     fprintf(stderr, "%3d: set nestlock\n", omp_get_thread_num());
00381   }
00382   omp_set_nest_lock(s);
00383 }
00384
00385 void POMP_Unset_nest_lock(omp_nest_lock_t *s) {
00386   if ( pomp_tracing ) {
00387     fprintf(stderr, "%3d: unset nestlock\n", omp_get_thread_num());
00388   }
00389   omp_unset_nest_lock(s);
00390 }
00391
00392 int  POMP_Test_nest_lock(omp_nest_lock_t *s) {
00393   if ( pomp_tracing ) {
00394     fprintf(stderr, "%3d: test nestlock\n", omp_get_thread_num());
00395   }
00396   return omp_test_nest_lock(s);
00397 }
```

ompi-0.8.2/pomp/pomp_lib.h

```
00001 /*
00002   OMPi OpenMP Compiler
00003   Copyright 2001-2004 Vassilios V. Dimakopoulos, Elias Leontiadis, George Tzoumas,
00004   Alkis Georgopoulos.
00005
00006   This file is part of OMPi.
00007
00008   OMPi is free software; you can redistribute it and/or modify
00009   it under the terms of the GNU General Public License as published by
00010   the Free Software Foundation; either version 2 of the License, or
00011   (at your option) any later version.
00012
00013   OMPi is distributed in the hope that it will be useful,
00014   but WITHOUT ANY WARRANTY; without even the implied warranty of
00015   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00016   GNU General Public License for more details.
00017
00018   You should have received a copy of the GNU General Public License
00019   along with OMPi; if not, write to the Free Software
00020   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
00021
00022   Portions of this code were take from OPARI Version 1.1, (C) 2001,
00023   Forschungszentrum Juelich, Zentralinstitut fuer Angewandte Mathematik
00024 */
00025
00026 #ifndef POMP_LIB_H
00027 #define POMP_LIB_H
00028
00029 #include <omp.h>
00030
00031 #ifdef __cplusplus
00032 extern "C" {
00033 #endif
00034
00035 struct ompregdescr {
00036   char* name;                 /* name of construct                    */
00037   char* sub_name;             /* optional: region name                */
00038   int   num_sections;         /* sections only: number of sections    */
00039   char* file_name;            /* source code location                 */
00040   int   begin_first_line;     /* line number first line opening pragma */
00041   int   begin_last_line;      /* line number last  line opening pragma */
00042   int   end_first_line;       /* line number first line closing pragma */
00043   int   end_last_line;        /* line number last  line closing pragma */
00044   void* data;                 /* space for performance data           */
```

```
00045   struct ompregdescr* next;    /* for linking                    */
00046 };
00047
00048 extern POMP_MAX_ID;
00049
00050 extern struct ompregdescr* pomp_rd_table[];
00051
00052 extern void POMP_Finalize();
00053 extern void POMP_Init();
00054 extern void POMP_Off();
00055 extern void POMP_On();
00056 extern void POMP_Begin(struct ompregdescr* r);
00057 extern void POMP_End(struct ompregdescr* r);
00058
00059 #ifdef _OPENMP
00060 extern void POMP_Atomic_enter(struct ompregdescr* r);
00061 extern void POMP_Atomic_exit(struct ompregdescr* r);
00062 extern void POMP_Barrier_enter(struct ompregdescr* r);
00063 extern void POMP_Barrier_exit(struct ompregdescr* r);
00064 extern void POMP_Flush_enter(struct ompregdescr* r);
00065 extern void POMP_Flush_exit(struct ompregdescr* r);
00066 extern void POMP_Critical_begin(struct ompregdescr* r);
00067 extern void POMP_Critical_end(struct ompregdescr* r);
00068 extern void POMP_Critical_enter(struct ompregdescr* r);
00069 extern void POMP_Critical_exit(struct ompregdescr* r);
00070 extern void POMP_For_enter(struct ompregdescr* r);
00071 extern void POMP_For_exit(struct ompregdescr* r);
00072 extern void POMP_Master_begin(struct ompregdescr* r);
00073 extern void POMP_Master_end(struct ompregdescr* r);
00074 extern void POMP_Parallel_begin(struct ompregdescr* r);
00075 extern void POMP_Parallel_end(struct ompregdescr* r);
00076 extern void POMP_Parallel_fork(struct ompregdescr* r);
00077 extern void POMP_Parallel_join(struct ompregdescr* r);
00078 extern void POMP_Section_begin(struct ompregdescr* r);
00079 extern void POMP_Section_end(struct ompregdescr* r);
00080 extern void POMP_Sections_enter(struct ompregdescr* r);
00081 extern void POMP_Sections_exit(struct ompregdescr* r);
00082 extern void POMP_Single_begin(struct ompregdescr* r);
00083 extern void POMP_Single_end(struct ompregdescr* r);
00084 extern void POMP_Single_enter(struct ompregdescr* r);
00085 extern void POMP_Single_exit(struct ompregdescr* r);
00086 extern void POMP_Workshare_enter(struct ompregdescr* r);
00087 extern void POMP_Workshare_exit(struct ompregdescr* r);
00088
00089 extern void POMP_Init_lock(omp_lock_t *s);
00090 extern void POMP_Destroy_lock(omp_lock_t *s);
00091 extern void POMP_Set_lock(omp_lock_t *s);
00092 extern void POMP_Unset_lock(omp_lock_t *s);
00093 extern int  POMP_Test_lock(omp_lock_t *s);
00094 extern void POMP_Init_nest_lock(omp_nest_lock_t *s);
00095 extern void POMP_Destroy_nest_lock(omp_nest_lock_t *s);
00096 extern void POMP_Set_nest_lock(omp_nest_lock_t *s);
00097 extern void POMP_Unset_nest_lock(omp_nest_lock_t *s);
00098 extern int  POMP_Test_nest_lock(omp_nest_lock_t *s);
00099 #endif
00100
00101 extern int pomp_tracing;
00102
00103 #ifdef __cplusplus
00104 }
00105 #endif
00106
00107 #endif
```