Evaluating NUMA-Aware Optimizations for the Reduce Phase of the Phoenix++ MapReduce Runtime

A Thesis

submitted to the designated

by the General Assembly

of the Department of Computer Science and Engineering

Examination Committee

by

Anastasios Souris

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION

IN COMPUTER SYSTEMS

University of Ioannina

August 2020

Examining Committee:

- **Vassilios V. Dimakopoulos**, Assoc. Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Aristides Efthymiou**, Assist. Professor, Department of Computer Science and Engineering, University of Ioannina
- **Evaggelia Pitoura**, Professor, Department of Computer Science and Engineering, University of Ioannina

Table of Contents

List of Figures

Abstract

Anastasios Souris, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, August 2020.
Evaluating NUMA-Aware Optimizations for the Reduce Phase of the Phoenix++ MapReduce Runtime.
Advisor: Vassilios V. Dimakopoulos, Associate Professor.

MapReduce is a programming model used to process large volumes of data. To implement an algorithm in the MapReduce programming model we need to provide two methods called *map* and *reduce*. The *map* function transforms the input to a set of (key, value) pairs. The *reduce* function receives as input all values associated with a key, as they were produced by the *map* function, aggregates them according to a user-supplied function and produces a single value as output. Phoenix++ is an implementation of the MapReduce parallel programming model for shared memory systems.

In this thesis we evaluate NUMA-aware optimization techniques for the reduce phase of the Phoenix++ implementation of the MapReduce parallel programming model for shared memory systems. A NUMA machine is comprised of a set of NUMA nodes that are linked together with interconnect links. Each NUMA node consists of its own local memory (i.e DRAM) and a number of CPUs. In this way, a CPU can access memory in its own NUMA node faster than memory located in other NUMA nodes.

To begin with, we evaluate two sets of methods that are based on the well-known and historical tournament-based barrier algorithm, whereby we hierarchically reduce the (key,value) pairs first within NUMA nodes and then among all NUMA nodes. The second set of methods we evaluate are extensions of the current implementation of the reduce phase in the Phoenix++ runtime, whereby we implement various reduce task

distribution policies that dictate to which thread a reduce task should be executed, where a reduce task implies the reduction over a specific range of keys. We evaluate those methods against synthetic workloads and deduce that for the case where the workload exhibits a specific kind of locality we observe performance advantages of up to 30.85%.

Εκτεταμενη Περιληψη

Αναστάσιος Σουρής, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Αύγουστος 2020.
Αξιολόγηση Μεθόδων Βελτιστοποίησης για NUMA Αρχιτεκτονικές Στην Φάση Reduce του Μοντέλου Παράλληλου Προγραμματισμού MapReduce Χρησιμοποιώντας την Υλοποίηση Phoenix++.
Επιβλέπων: Βασίλειος Β. Δημακόπουλος, Αναπληρωτής Καθηγητής.

Το MapReduce είναι ένα μοντέλο προγραμματισμού που χρησιμοποιείται για την επεξεργασία μεγάλων όγκων δεδομένων. Προκειμένου να προγραμματίσουμε έναν αλγόριθμο στο μοντέλο προγραμματισμού MapReduce, πρέπει να παρέχουμε δύο μεθόδους που ονομάζονται *map* και *reduce*. Η συνάρτηση *map* μετατρέπει την είσοδο σε ένα σύνολο ζευγών (κλειδί, τιμή). Η συνάρτηση *reduce* λαμβάνει ως είσοδο όλες τις τιμές που σχετίζονται με ένα κλειδί, όπως παρήχθησαν από τη συνάρτηση *map*, τις συγκεντρώνει σύμφωνα με μια συνάρτηση παρεχόμενη απο τον χρήστη και παράγει μία μόνο τιμή ως έξοδο. Το Phoenix++ είναι μια υλοποίηση του μοντέλου παράλληλου προγραμματισμού MapReduce για κοινόχρηστα συστήματα μνήμης.

Σε αυτή τη διατριβή αξιολογούμε τις τεχνικές βελτιστοποίησης για αρχιτεκτονικές NUMA στην φάση μείωσης του Phoenix++ που είναι υλοποίηση του μοντέλου παράλληλου προγραμματισμού MapReduce για κοινόχρηστα συστήματα μνήμης. Ένα μηχάνημα NUMA αποτελείται από ένα σύνολο κόμβων NUMA που συνδέονται μαζί με συνδέσμους διασύνδεσης. Κάθε κόμβος NUMA αποτελείται από τη δική του τοπική μνήμη (δηλαδή DRAM) και έναν αριθμό CPU. Με αυτόν τον τρόπο, μια CPU μπορεί να αποκτήσει πρόσβαση στη μνήμη στον δικό της κόμβο NUMA ταχύτερα από τη μνήμη που βρίσκεται σε άλλους κόμβους NUMA.

Κατ 'αρχάς, αξιολογούμε δύο σύνολα από μεθόδους που βασίζονται στον γνωστό και ιστορικό ιεραρχικό αλγόριθμο για barriers (tournament barrier algorithm), όπου

μειώνουμε ιεραρχικά τα ζεύγη (κλειδί, τιμή) πρώτα μέσα στους κόμβους NUMA και μετά μεταξύ όλων των κόμβων NUMA. Το δεύτερο σύνολο μεθόδων που αξιολογούμε είναι οι επεκτάσεις της τρέχουσας εφαρμογής της φάσης μείωσης στο χρόνο εκτέλεσης του phoenix++, όπου εφαρμόζουμε διάφορες πολιτικές διανομής εργασιών reduce που υπαγορεύουν σε ποιο νήμα πρέπει να εκτελεστεί μια εργασία reduce, όπου μια εργασία reduce συνεπάγεται τη μείωση σε ένα συγκεκριμένο εύρος κλειδιών. Αξιολογούμε αυτές τις μεθόδους έναντι συνθετικών φορτίων εργασίας και συμπεραίνουμε ότι στην περίπτωση όπου ο φόρτος εργασίας εμφανίζει ένα συγκεκριμένο είδος locality παρατηρούμε πλεονεκτήματα απόδοσης έως και 30,85%.

Chapter 1

Introduction

1.1 Parallel Systems and Parallel Programming Models

Parallel systems are systems that are comprised of many computational units that work in unison in order to solve a computational task. In this thesis we are concerned with shared-memory cc-NUMA parallel architectures. These architectures are comprised of multiple processor interconnected by a network interconnect, each of which is a multi-core processor comprised of multiple cores each of which may or may not be multithreaded. The main memory in such a parallel system is shared between all processing units in a single address space and because the main memory is not accessed with equal latency and bandwidth from all processing units we call this architecture non-uniform memory access. This happens because each NUMA node is equipped with its own RAM module and a processing unit accessing memory that does not belong to its own NUMA node has to traverse the interconnection network to reach the destination memory module. Furthermore, in a cc-NUMA architecture (i.e cache-coherent NUMA architecture) each processor as a cache hierarchy typically

comprised of 1 or 2 levels of private caches followed by 1 level of a larger shared cache. A parallel programming model is a paradigm used to program algorithms on various parallel architectures. An important feature of a parallel programming model is its ability to perform optimally on a wide range of different parallel architectures. Popular parallel programming models that have shown to exhibit these virtues are the *task/dataflow* model and the *MapReduce* model. In this thesis we are concerned with the MapReduce parallel programming model.

1.2 The MapReduce Programming Model

MapReduce is a programming model used to process large volumes of data. To implement an algorithm in the *MapReduce* programming model we need to provide two methods called *map* and *reduce*.

Map The *map* function transforms the input to a set of $(key, value)$ pairs.

Reduce The *reduce* function receives as input all *values* associated with a key, as they were produced by the *map* function, aggregates them according to a user-supplied function and produces a *single value* as output.

To illustrate the *MapReduce* model we are going to use the popular *word count* application. Our goal is to process an input document and output the number of items each distinct word appears in that document. The *map* function receives a portion of the input document, splits it into its constituent words, and for each word emits a $(key, value)$ pair where the *key* is the word itself and the value is the number 1. The *reduce* function takes as input all values produced for one key, that is for one *word*. By summing those values, which are all equal to 1 according to the *map* function, we get the number of occurrences of that word in the input document.

A *MapReduce* application may optionally include a *merge* phase after the *map* and *reduce* stages have finished. In the *merge* phase, the output of the *reduce* stage is sorted by value. In our *word count* example, if we sort the output by value, that is by number of occurrences, in decreasing order, we can easily get the *top-K* occurring words in the input document.

1.3 Objectives of this Thesis

In this thesis we aim to improve the performance of the reduce phase for the Phoenix++ runtime on cc-NUMA architectures. We evaluate two sets of methods that are based on the well-known and historical tournament-based barrier algorithm, whereby we hierarchically reduce the (key,value) pairs first within NUMA nodes and then among all NUMA nodes. The second set of methods we evaluate are extensions of the current implementation of the reduce phase in the Phoenix++ runtime, whereby we implement various reduce task distribution policies that dictate to which thread a reduce task should be executed, where a reduce task implies the reduction over a specific range of keys. We evaluate those methods against synthetic workloads and deduce that for the case where the workload exhibits a specific kind of locality we observe performance advantages of up to 30.85%.

1.4 Thesis Organization

This thesis consists of the following chapters:

Introduction This chapter.

Background on cc-NUMA Architectures This chapter provides a background on the characteristics of cc-NUMA Architectures, how they affect performance and optimization techniques for dealing with them.

Background on Phoenix++ This chapter provides a background on the Map/Reduce programming model and on the structure and architecture of the phoenix++ implementation.

Improving the Reduce Phase of Phoenix++ This chapter describes two sets of algorithms that are evaluated for the reduce phase of Phoenix++. The first set of methods rely on the hierarchical algorithms for the reduce phase and the second set of methods are task distribution policies for the reduce phase.

Implementation Details This chapter provides some details on the modifications and additions made to the phoenix++ runtime for the purpose of this thesis.

Experimental Evaluation Lastly, this chapter provides the results from the experiments as well as conclusions and future work.

Conclusions and Future Work This chapter concludes this thesis.

Chapter 2

Background on cc-NUMA Architectures

2.1 Optimization Techniques on cc-NUMA Architectures

In a Non-Uniform Memory Architecture (NUMA) system, the machine is comprised of a set of NUMA nodes that are linked together with interconnect links. Each NUMA node consists of its own local memory (i.e DRAM) and a number of CPUs. In this way, a CPU can access memory in its own NUMA node faster than memory located in other NUMA nodes. Moreoever, in a cache-coherent NUMA system (*cc-NUMA*) the system maintains coherence between its caches among the NUMA nodes. An example *cc-NUMA* system is showcased in figure 2.1. This system consists of 4 NUMA nodes each of which contains 6 cores for a total of 24 cores. Each NUMA node contains approximately 4GB RAM for a total of 16 GB of RAM. All cores within a NUMA node share approximately 5MB of L3 cache. The NUMA nodes are connected via direct bidirectional interconnection links.

Figure 2.1: The architecture of a cc-NUMA machine (output given by lstopo)

2.1 Optimization Techniques on cc-NUMA Architectures

The problem of scheduling multi-programmed workloads in NUMA architectures is of particular interest due to the idiosyncrasies of the architecture. Early solutions to this problem considered *data locality* as the optimizing goal for NUMA architectures. The intuition is that since a CPU can access memory faster in its local NUMA node, then it is best to schedule a process in the NUMA node that contains its memory. However, in Blagodurov et al. [2011] they have shown experimentally that data locality may even hurt performance in NUMA architectures, and therefore a new line of research has emerged that aims to design and implement new scheduling algorithms for NUMA systems.

2.1.1 Characteristics of Scheduling Algorithms for NUMA Systems

In order to design a scheduling algorithm for NUMA systems, one must take into consideration the following issues:

- Relative Placement of Threads and Memory
- Thread Migration
- Memory Migration
- Data Sharing
- Resource Contention
- Asymmetry
- Process Metrics Measurement

2.1.2 Factors Affecting Performance

Starting with a deep experimental analysis in Blagodurov et al. [2011], one witnesses the results of *resource contention* in the performance of schedules. The sources of resource contention addressed in Blagodurov et al. [2011] are (1) contention for the shared last-level cache, (2) contention for the memory controller, (3) contention for the inter-domain interconnect and, finally, (4) remote access latency. Contrary to schedules optimized for data locality and, hence, minimizing remote access latency,

the authors in Blagodurov et al. [2011] show that the order of importance for the sources of contention as far as performance degradation is concerned, is (1) contention for the memory controller, (2) contention for the inter-domain interconnect, (3) remote access latency and, lastly, (4) contention for the shared last-level cache. In Lepers et al. [2015], it is shown that the asymmetry in the architecture of a NUMA system also influences the performance of a scheduler, and, hence, must be taken into consideration when deciding the placement of threads. In particular, in a NUMA system (1) links can have different bandwidths, (2) links can send data transfer in one direction than in the other, (3) links can be used only by a subset of the nodes, and, finally, (4) links can be either bidirectional or unidirectional. Last but not least, data sharing also affects the choice of thread placement. As pointed out in Srikanthan et al. [2015] threads that share data and are not scheduled in the same CPU cause increased contention in the inter-domain interconnect and the local cache controllers due to increased cache-coherence traffic.

2.1.3 Collecting Metrics

In order to quantify the degree of resource contention and data sharing, a scheduler needs to collect metrics on the behaviour of an application. Approaches here differ on the how fine-grained these measurements are. In Blagodurov et al. [2011] they use a rather simple inference prediction heuristic for resource contention, which is the *cache-miss rate heuristic*, stating that if two threads in one CPU both have high cache-miss rates then they probably interfere for shared resources and, hence, should be scheduled in different domains. Other schedulers take into consideration which CPUs are communicating with each other. In Lepers et al. [2015], they collect CPU-to-CPU communication using hardware counters that measures both cache-coherence and memory traffic from one CPU to another. In Srikanthan et al. [2015] they use a more fine-grained approach, whereby they are able to separate traffic due to cache-coherence protocols and traffic due to memory accesses. Finally, for memory migration one needs to identify which memory pages one thread accesses. To that end, in Blagodurov et al. [2011], Dashti et al. [2013] they use instruction-based sampling, a technique that samples instructions and filters those that are remote memory accesses.

2.1.4 Memory Migration

The schedulers in Blagodurov et al. [2011], Dashti et al. [2013], Lepers et al. [2015] always migrate the memory together with migrating a thread. This happens by using instruction-based sampling and capturing remote memory references. When a remote memory reference is encountered then the corresponding memory page, as well as some close-by memory pages, are migrated to the issuing NUMA node. In Lepers et al. [2015], they also perform a *full memory migration* when the above approach doesn't reduce the amount of remote memory references, which means that all memory pages of the application are moved.

2.1.5 Thread and Memory Placement

Algorithms for scheduling on NUMA systems, broadly speaking can be classified in three major categories: (1) those that migrate memory towards the thread that accesses that memory, (2) those that migrate a thread towards the memory it accesses, and (3) those that employ a hybrid approach utilizing both solutions.

The aim of these schedulers is to reduce resource contention and co-locate data-sharing threads as long as it doesn't increase resource contention. DINO recognizes threads that are competing for resources inside a CPU and then migrates threads so as to co-schedule threads with a high resource consumption with threads that have low resource consumption. When migrating a thread, DINO also employs a mechanism for memory migration in order to avoid the disadvantages of NUMA-agnostic migrations. To account for data sharing, DINO uses a heuristic to co-locate threads of the same application that may share data. Carrefour only deals with memory placement and assumes a thread placement mechanism that co-locates data-sharing threads as long as they do not put pressure on the shared resources and puts bandwidth hungry threads on separate sockets. Then, Carrefour attempts to place memory in an optimal position by utilizing three mechanisms: (1) page co-location that places the memory page together with the threads that access this page so as to avoid contention on the inter-domain interconnect and remote memory references, (2) page interleaving that attempts to equalize the pressure on the local memory controllers across the NUMA nodes, and (3) page replication that replicates a memory page to the NUMA nodes from which it is accessed in order to eliminate memory controller and inter-domain interconnect contention. AsymSched attempts to find a subset of the NUMA nodes that

provides the highest bandwidth to the cluster of threads that require high bandwidth. SAM attempts to co-locate data sharing threads by monitoring inter-socket coherence traffic and migrates threads that put pressure on the local bandwidth consumption to another socket in order to reduce the pressure.

In Srikanthan et al. [2016], the authors extend the *SAM* algorithm described in Srikanthan et al. [2015] in order to, first, take into consideration *latency tolerance* for tasks with high inter-socket coherence activity, and, secondly, to make better mapping decisions on hyperthreads. The *latency tolerance* of a pair of tasks is their ability to hide their communication latency and is quantified by the per-task IPC of that pair of tasks, because a high per-task IPC results in more potential of instruction interleaving in-between successive communication and cache coherence events. Consequently, *SAM* is extended to prefer co-locating pairs of tasks with lower latency tolerance. To measure the latency tolerance, *SAM* monitors for each task its IPC and *SPC* which stands for *stalls per inter-socket coherence event*. Through extended experimentation, the authors concluded that when the *SPC* value is high enough it can be used by itself as an indicator of latency tolerance. For low to moderate value sof *SPC*, however, we need to examine the IPC of each task to determine the latency tolerance of the pair. Specifically, higher IPC indicates better latency tolerance for reasons described previously.

In addition to that, after *SAM* has identified a pair of tasks to co-locate by taking into consideration inter-socket coherence traffic and latency tolerance, then it needs to decide whether to schedule those tasks on the same core as hyperthreads, on distinct cores on the same socket or even not to co-locate them at all. The benefit of hyper-threads is that they share the private caches and their disadvantage is that they share functional units. We consider two cases. The first case is about non latency tolerant tasks. Those tasks where categorized as such because they do not have sufficient ILP to mask the latency of communication/coherence events. Therefore, these tasks do not require exclusive use of the core's functional units and would benefit from the fast communication through the core's private caches. The second case is about latency tolerant tasks. Those tasks have sufficient ILP which means that they make heavy usage of a core's functional resources. Consequently, we need to place those tasks either on distinct cores on the same socket or even to distribute them on different sockets in case of a high latency tolerance value.

In Popov et al. [2019] stress the importance of co-optimizing the following aspects of

executing a parallel application on a parallel architecture:

Thread Mapping How are threads mapped to the target architecture. The threads could be *scattered* across the available sockets, *compacted* or mapping according to a *contiguous* policy. The difference between the compact and contiguous policies is that compact first fills one numa node before proceeding to the next, whereas a compact policy evenly spreads the threads across the numa domains.

Page Mapping How are pages mapped, i.e according to a *first-touch* policy , according to a *locality* policy whereby pages are mapped to the NUMA domain from which most accesses to that page occur, according to a *balance* policy such that pages are scattered across the NUMA domains so that each NUMA domain receives an almost equal share of the total memory traffic, or according to a *mixed* policy that tries to optimize for both locality and balance.

Numa Node Degree How many of the NUMA nodes should be used.

Degree of Parallelism This involves choosing how many threads to use for the application.

However, optimizing simultaneously for all of those 4 criteria results in a extremely large search space. To reduce the search space, the authors employ the following two strategies. To begin with, instead of executing the whole application, the framework extracts and executes specific *codelets* that are representative of the application (the authors observe that almost all applications consist of phases of executions and each phase basically consists of a single kernel – called codelet, which is repeatedly executed). One specific codelet will have to be executed under various configurations from the search space so that an optimal solution is found. Henceforth, for one codelet, the framework extracts its input working set as well as the NUMA first-touch based page mapping of its input working set so that the codelet can be replayed. This is achieved using the codelet extractor and replayer (CERE) framework that uses *ptrace* to capture memory accesses.

The next step is to executing each codelet under various configuration of the 4 criteria above. For each codelet, its input working set is laoded and the captured page mapping is applied. The codelet is executed once to warm the cache, and then multiple times to derive a median execution time. Even though the framework executes individual codelets and not the whole application and, therefore, we already have

a substantial decrease in search time, the possible configurations of those 4 criteria above still result in a very large search space which the framework must prune. The search space per each of the aforementioned criteria is:

Thread Mapping This involves choosing number of NUMA nodes (called thread NUMA degree *TND*) and how threads are mapped to the available units (called thread placement policy *TPP*) for which the *scatter* and *contiguous* policies are considered.

Page Mapping This involves choosing the page numa degree *PDN* which specifies how many of the numa nodes to use for page mapping, as well as the page placement policy *PPP* for which the *first-touch*, *locality*, *balance* and *mix* policies are considered.

Degree of Parallelism This involves choosing how many threads to use for the application.

To reduce the search space, the framework performs the following simple search space pruning phase:

- Only one thread placement policy is evaluated whenever the number of numa nodes used is either 1 or larger or equal to the number of threads used. That is because in that case, both scatter and contiguous policies are equivalent.
- Locality and first-touch page placement policies are evaluated only when the thread numa degree is the same as the page numa degree.
- For all configurations, the number of threads considered is always larger or equal than the number of numa nodes and less than or equal to the total available cores for those numa nodes.

A large overhead incurred by the framework when applying page placement policies like locality or balance, is the monitoring of how threads access the pages. To reduce this overhead, the framework captures this information only once when the codelet is captured before being replayed. The last step in their framework consists of application-wide optimization. By optimizing each codelet separately, the framework optimizes execution per region. However, one region may affect another region, for instance, if the page mapping for one region as produced by the optimal solution

of it's codelet search space exploration phase is different from that used by the next region that would result in costly page migrations for the next region. The framework proposes this solution: If for two consecutive regions A and B we have found two optimal configurations Ca and Cb, respectively, and regions A and B share pages or threads (which is identified by the memory accesses collected by the codelet's execution), then we choose whichever of Ca or Cb results in the best execution for both regions.

In Muddukrishna et al. [2016] the authors place emphasis on the fact that for maximal efficiency on NUMA systems besides load balancing we need to focus on minimizing memory access costs albeit in a portable manner that is agnostic to the details of the underlying NUMA architecture. Such an approach can be utitlized by both experienced and non-experienced programmers and is integrated by the authors in the user-friendly OpenMP programming model. To handle data distribution the authors propose that the programmer uses a memory allocation function that receives a policy as an extra argument. Three policies are suggested: The first is called *standard* and just uses the OS memory allocation policy. The second is called *fine* and splits the memory being allocated into units and distributes these units in a round-robin fashion across all NUMA domains. The last policy is called *coarse* and places the memory being allocated whole in a single NUMA domain, but each time memory is allocated using the *coarse* policy a different NUMA domain is selected in a round-robin fashion for that memory to be allocated at. The *fine* policy is suggested to be used in applications where we allocate the data once and then we instantiate multiple tasks that operate on that data. If we were allocating the data on a single NUMA donain, then we would suffer from traffic congestion from all those tasks trying to access that data. Instead, we use the *fine* policy and distribute the data across the NUMA domains which means that memory traffic produced by the tasks accessing that data is also distributed among the memory controllers. In case the data is allocated by many tasks then the authors propose using the *coarse* policy regardless of the number of tasks accessing each data allocated. In case of one task accessing each data, a *coarse* policy assures that each task accesses its data in a separate NUMA domain from other tasks and hence we utilize the available bandwidth better. The same is true in the case where multiple tasks access the same data with the hypothesis that each task accesses only one allocation, cause otherwise one task would require bandwidth from

multiple NUMA domains. Assuming that the programmer has chosen the memory policies per allocation optimally, we can enhance load-balancing strategies in order to better schedule tasks by minimize memory access latencies. In order to do that, the authors introduce a task queue exists per NUMA domain. It is also assumed that the programmer provides the *task footprint* of the task which consists of a description of the data the task accesses. Load-balancing is treated in two phases: In the fist phase, we have *work-dealing* where the initial placement of a task is decided during its creation. Utilizing the memory footprint of the task, we calculate the number of bytes accessed by the task from each NUMA domain. Then the task is enqueued in the queue of the NUMA domain from which it will incur the least access cost to its memory. To avoid having the scheduling costs outweight the benefit of placing the task in that queue, two heuristics are used whereby the task is immediately placed in the queue of the thread making the decision. The first heuristic tests for the memory footprint of the task being less that a threshold in which case making memory optimizations is pointless. The second heuristic tests that the memory footprint of the task is not perfectly distributed among the NUMA domains because then regardless of the NUMA node in which we place it, the task would incur the same memory access costs. In the second phase, we have *work-stealing* that occurs whenever threads have no more work in the local queue. In that case, a thread attempts to steal from NUMA nodes in increasing distance order with the exception that if the remote target queue is nearly empty that NUMA node is skipped and the stealing thread continues its stealing attempts with the next NUMA node in order.

In Drebes et al. [2016] the authors present the *enhanced work-pushing* and *deferred allocation* techniques, in order to distribute data across memory controllers to avoid contention, and keep memory accesses local in order to reduce remote memory references. Consider a task t that has input dependencies on n input buffers $I_0, \ldots, I_{n-1}$ and output dependencies on m output buffers $O_0, \ldots, O_{m-1}$. *Enhanced work pushing* attempts to optimize the data locality of the task by placing the task on the NUMA node from where its access cost in terms of remote versus local memory references to its input and output buffers is minimized. To that end, we first determine whether we want to consider only the input buffers, only the output buffers or both (*input-only*, *output-only* or *input-output* policy). Then, for each buffer we are interested in, we add its size in bytes multiplied by a weight to the total number of bytes accessed

by the task for the NUMA node where the buffer is allocated. The weight is defined to distinguish between reads and writes since their cost typically differs. At the end, for each NUMA node N_i we estimate the access cost of the task if we place it in N_i by adding up the access cost from N_i to each other NUMA node N_j (i.e multiply the average access cost between N_i and N_j by the weighted number of bytes in N_j accessed by the task) and we choose the NUMA node with the minimum such cost. Observe that this technique tries to optimize only for data locality and not for memory contention. That is, it is assumed that there already exists a good enough data distribution. However, the authors suggest that data distribution should be done automatically by the runtime system because manual data distribution in a parallel control program is error prone and not portable performance-wise between different NUMA architectures. For this reason, they suggest the *deferred allocation* technique. Since by the work-stealing principle we know that tasks are fairly distributed among the NUMA nodes, if we use the policy that each task allocates its output buffers locally in the NUMA node where it executes, then we also have a fairly good distribution of the buffers among the NUMA nodes. Note that we also gain in data locality since all writes of the task are to local output buffers. Last but not least, since deferred allocation guarantees local writes for the task, we use the enhanced work pushing technique with the *input-only* policy.

In Virouleau et al. [2016], the authors aim at improving performance of OpenMP applications written with task dependencies on NUMA systems by exploiting (1) how data distribution occurs, (2) the assignment of ready tasks to the processors and (3) how load balancing is performed with respect to the topology of the NUMA system. They rely their work on the XKAAPI runtime system which is a work-stealing based task execution runtime. To abstract the NUMA system, XKAAPI uses the notion of places. A place is a list of tasks associated with a subset of the system's processing units. The following scheduler's actions are configurable: selecting a victim which is abstracted by the method *Wsselect* which is called whenever an idle worker needs to steal a ready task from some place, selecting a place to push a ready task which happens whenever the execution of one task enables its children which now become ready for execution and need to be put in some place and is abstracted by the method *WSpush*, and pushing a set of initial ready tasks using the method *Wspush_init*. To begin with, we need to distribute the sources of the dependency graph. To mitigate

congestion on the bandwidth of NUMA nodes, the authors propose the *cyclicnuma* strategy whereby the initial tasks are distributed in a round-robin fashion around the NUMA nodes and the *randnuma* strategy whereby the initial tasks are distributed randomly around the numa nodes. The *Wsselect* strategy can be chosen from one of the following ones: *sProcNuma*, *sNumaProc*, *sProc*, *sNuma*. In more detail, *sProcNuma* dictates that a thief first tries to steal from the places of the processors of its local NUMA node, then from the place of its local NUMA node and if that fails it continues in the same manner with a randomly selected remote node. The *sNumaProc* strategy is similar to the sProcNuma strategy with the difference that the thief will first try to steal from the place of the NUMA node and then from the places of its processors. The *sProc* strategy ditcates that the thief will only look at the places of the processors of its local NUMA node and then at the place of its local NUMA node. With the *sNuma* strategy the thief will visit only the places of NUMA nodes and not the places of the processors contained within each NUMA node. When a task becomes ready the *Wspush* strategy needs to decide to which place to push that task. Two of the strategies ignore data dependencies completely and just push the task either to the place of the processor (*pLoc* strategy) or to the place of the NUMA node (*pLocNum* strategy) where the processor that executes the *Wspush* method resides. The other methods attempt at exploiting the data dependencies of the task. The first one, *pNumaW*, pushes the task at the place of the NUMA node where most of its output data is allocated. The last strategy, *pNumaWLoc*, is similar to the *pNumaW* with the difference that if the chosen NUMA node where the task is pushed is the one where the processor that executes the *Wspush* method resides, we push the task to the place of that processor instead. In their evaluation they concluded the following facts: (1) regardless of the *Wsselect* and *Wspush* strategies selected the *cyclicnuma* strategy works best. (2) Restricting a task to only the node where its output data resides and not allowing it to be stolen hurts performance. For example, if most of the tasks write the same set of data then all those tasks will occupy a subset of the processors whereas the rest of the processor will remain idle. (3) The *sNumaProc* + *pNumaWloc* combination seems to work best followed closely by the *sProcNuma* + *pNumaWloc* combination.

In Chen and Guo [2015] the authors target iterate divide-and-conquer applications that have tree-shaped execution DAGs. Their aim is to solve two problems with randomized work stealing: (1) Tasks perform remote memory accesses because their

assigned to random sockets, and (2) the shared caches are not utilized efficiently. *LAWS* is an algorithm that comprises of three subsystems: The *load-balanced task allocator*, the *adaptive dag packer* and the *triple-level work-stealing scheduler*. As far as the load-balanced task allocator is concerned, its purpose is to distribute the data set and the tasks of the applications evenly among the available sockets. The assumption made is that a task divides its data set evenly among its children according to its branching degree. Assuming that the data region accessed by all the tasks is $[0, D)$ then, based on our assumption, we can determine for each task the data region it will access based on the data region of its parent and its parent's branching degree. Moreover, due to the assumption that the amount of data coincides with processing requirements, if we assume that we have M sockets then we want to evenly distribute the data region $[0, D)$ to all sockets. Therefore, each socket I, $1 <= I <= M$, will receive a portion of the data region $[0, D)$ described as $[(i-1)D/M, i(D/M))$. When we spawn a task we know its data region and hence can determine to which socket we should schedule that task. This mapping computation is deterministic across iterations and, consequently, a task will be assigned to the same socket each time. This results in the task accessing from its local memory node and not from a remote node. This is true because at the first iteration when a task is assigned to a socket it will be executed there (i.e work-stealing is disabled) and due to the first-touch policy its data region will be allocated on that socket. Regarding the adaptive dag packer, after we have chosen which tasks to execute in each socket we want to optimize shared cache usage for each socket. The rationale is to group the tasks assigned to each socket into CF subtrees where a CF subtree denotes a group of tasks whose combined data fits in the shared cache of the socket and, therefore, should be executed in isolation in that socket. So, the CF subtrees are executed sequentially in each socket and each CF subtree within the socket is executed in parallel. The problem is how to find the CF subtrees. To do that, LAWS performs the first iteration to estimate the shared cache size required by each task (note that we consider the subtree rooted at that task). Then, a task becomes a CF subtree root if its shared cache size fits in the shared cache of the socket and its parent's doesn't. The authors explain two drawbacks in their method of estimating the shared cache size required by a task which results in suboptimal CF subtrees. For that reason, in later iterations LAWS attempts to alter the CF subtrees root assignment in order to find the optimal one. The authors make the following observation: Both too large and too small CF subtree sizes result in

worse execution times. The optimal solution is somewhere in between. Therefore, LAWS starts by evaluating smaller CF subtree sizes. If execution times get better then the original CF subtree sizes were too large and, as a result, LAWS need to keep decreasing the subtree sizes until an optional one is found. Otherwise, the original CF subtree sizes were too small and LAWS, instead, increases the CF subtree sizes. Last but not least, the triple-level work-stealing scheduler has a pool of CF subtree roots for each socket that they execute sequentially. After a CF subtree has been taken from the pool, the cores in the socket use classic work-stealing to execute the assigned CF subtree. When execution of that subtree has finished an assigned head core for the socket fetches the next available CF subtree from the socket's pool or if none exists it attempts to steal a CF subtree from another socket's pool.

In Majo and Gross [2017] the authors describe how to expose to the application programmers NUMA-aware optimizations that are portable among different architectures and composable, meaning that the application is composed of multiple independent parallel software libraries. The authors implement their ideas as extensions to the popular TBB platform named TBB-NUMA. To begin with, the platform is made aware of different concurrently executing runtimes by providing them with threads. TBB-NUMA provides a *Resource Management Layer* that distributes available threads between co-running threads schedulers so that each thread scheduler gets approximately an equal number of threads from each available processor. The reason behind this decision is to make available all of the system's resources like bandwidth to each registered task scheduler. Furthermore, to become NUMA-aware TBB-NUMA assigns a *mailbox* to each NUMA domain. Then, the application programmer can specify affinities for a task to a specific NUMA domain by associating the task with the mailbox of that NUMA domain. Since it suffices for any thread assigned to the NUMA domain of the mailbox to execute that task for it to avoid remote memory references, in TBB-NUMA a thread looks for work in the mailbox assigned to its NUMA domain. With respect to mailboxes, TBB-NUMA provides some more optimizations. A task is present both in the local deque of the thread that spawned it and in the mailbox to which it has affinity to. Therefore, if the task gets stolen from the private dequeue of the thread before it gets retrieved from the mailbox it may not execute in the desired NUMA domain and hence experience remote memory references. A similar scenario occurs if the thread that spawned the task retrieves it from its local dequeue. TBB-

NUMA deals with these issues by having threads assess whether a given task is likely to be picked up soon from its mailbox before trying to steal it. Moreover, to avoid having the thread that spawned the task to retrieve it from its local deque before it gets retrieved from its mailbox, TBB-NUMA employs two heuristics. The first heuristic dictates that when a task submits children tasks for execution it should do so in an order such that tasks with affinity to the current NUMA domain are submitted last and those that have affinities to a different NUMA domain should be submitted first. In that way, when the thread retrieves from its local deque it will first take tasks with affinity from the local NUMA domain and later tasks that have affinity to a distant mailbox are more likely to be picked up at their destination. This heuristic however does not work in cases where a task submits only one task for execution that has affinity to a distant mailbox. In that cases, TBB-NUMA suggests that the programmer detaches the spawned task from its parent which basically means that the spawned task is not inserted in the thread's local dequeue but only in the processor it has affinity to. This does not mean, however, that the task is inserted only to the mailbox of the NUMA domain of the destination processor. That is, because due to the fact that the task scheduler gets assigned different number of threads each time it may be the case that the NUMA domain of the destination mailbox has no thread assigned to and therefore the task is never picked up from the mailbox. To resolve this issue, TBB-NUMA assigns a shared queue for each NUMA domain. The detached task is enqueued to the shared queue of the destination processor.

In Anbar et al. [2016] the authors propose *PHLAME* which is an execution model that is able to take advantage of the hierarchical locality in architectures with deep memory hierarchies. *PHLAME* relies on a locality-aware programming model meaning that the programmer is responsible for annotating in some way the data that each task/thread or activity touches. The experiments performed in their paper rely on PGAS and MPI programming models. Then, *PHLAME* is responsible for co-locating those tasks with the data they touch and, moreover, to distance every pair of activities accordingly to their degree of interaction. To decide on a best mapping strategy, *PHLAME* uses offline profiling. To characterize the architecture, *PHLAME* discovers its levels (i.e cores, dies, sockets, nodes, blades, chassis, cabinet etc) and the cost of transferring a message of size L between entities of the same level, for each level of the machine and for various message sizes L. This profiling stage needs to be

performed only once for each machine architecture. Then, for a given application and input profile, *PHLAME* performs an application profiling stage whereby for each pair of activities X_i and X_j the average message size and number of messages exchanged between those two activities are computed (this occurs for different bin sizes, i.e if bin 0 holds message sizes between 1 and 63 bytes then a message of size $1 <= k <= 63$ is counted for in bin 0, and similarly for bin 1 with message sizes of 64 bytes to 255 bytes etc). The first step to deciding a mapping for the threads, is to find a metric that characterizes the benefit of placing two tasks at a specific distance from each other. *PHLAME* calls this metric *FIT*. First, we compute for each pair of tasks X_i and X_j what communication cost would they incur if placed at some level for each level (i.e if placed in the same core, in the same socket, in the same numa node etc). This cost can be computed because from the application profiling phase PHLAME knows the amount of communication between every pair of activities. The fitness metric FIT for the pair of activities X_i and X_j and a particular level L, is computed as the sum for each other level $L^{'} != L$ of the difference of the cost of placing X_i and X_j at level $L^{'}$ and the cost of placing X_i and X_j at level L. Intuitively, the larger this metric the better level L is suited for activities X_i and X_j because the individual differences would be larger which means that placing X_i and X_j at a different level $L^{'}$ would incur larger cost. To produce hierarchical mappings, *PHLAME* follows 2 steps. In the first step a set of mappings is generated based on either bottom-up or top-down clustering approaches. Then, the *PHAST* algorithm chooses from among those mappings the one with the minimum total communication cost, where the total communication cost is the sum of the communication cost for each pair of activities X_i and X_j if placed at the level imposed by that mapping. Both clustering based methods model the execution as a graph where each vertex is a task and an edge between two tasks denotes their communication cost, which is known from the application profiling stage. A bottom-up based clustering algorithm works as follows: At the first iteration each task is placed in its own cluster. Then, groupings are made from those clusters based on the *FIT* metric. This means that tasks whose FIT metric is higher should be placed in the same cluster because they would benefit from being placed in the same level. For the new groups that have been computed, we re-compute their communication costs and their FIT metrics (as aggregates of their members) and then continue until we are left with one large group. Last but not least, a top-down clustering algorithm works as follows: We start with all tasks as a single group and

then split them into separate groups. We should split tasks that will not benefit from being placed together at that level, and hence the splitting phase puts priority to the pairs with the lowest fitness values. This process continues until we reach partitions of size 1.

The purpose of *Tumbler* described in Pusukuri et al. [2015] is to evenly balance load of threads across CPU sockets not based on the common heuristics employed by OS scheduler they compared against that rely on the number of threads. If two sockets have assigned the same number of threads this doesn't mean that the load imposed by their assigned threads will be the same, which almost certainly leads to performance degradation. Number of threads as an indicator to load balance across CPU sockets is poor because first, threads may perform different functionality and thus require different amounts of CPU resources (i.e consider the different stages of a pipeline application), secondly even if we are considering identical threads they may have been assigned different amounts of input to process, and, lastly, threads that communicate via locking may result in different loads to their CPUs due to lock contention that results in increased lock waiting times. Thus, Tumbler by evenly balancing load across CPU sockets attempts to minimize CPU idle times and improve performance. *Tumbler* periodically, after a grouping interval, examines the CPU load imposed by each thread, which is approximated by the sum of user percentage and kernel percentage time for that thread divided by the grouping interval time, sorts the threads in increasing load and divides them in as many groups as there are CPU sockets in a round-robin fashion. This results in an almost even distribution of cumulative load across the CPU sockets. Then, *Tumbler* performs the necessary thread migrations so that each group of threads is scheduled to its CPU socket leaving the OS scheduler the flexibility to load balance the group's threads within the CPU socket. To accommodate high lock contention scenarios where the load of threads changes frequently, *Tumbler* must adapt the grouping interval so that it reacts faster to application's phase changes. To do that, *Tumbler* keeps track of the variation of CPU variation and accordingly selects a grouping interval, i.e high variation in CPU utilization may imply high lock contention and, consequently, *Tumbler* needs to perform grouping and thread migrations more frequently.

In Jeannot et al. [2014] the authors present the *TreeMatch* algorithm that is used

to map processes to processing units in such a way that the communication profile of the application and the topology of the machine is taken into consideration. To begin with, *TreeMatch* profiles the application in order to collect its communication profile that consists of the following metrics for each pair of processes: (1) the number of messages exchanged, (2) the total amount of communication and (3) the average size of the messages exchanged between them. The drawback of this approach is that it isn't suitable for applications whose communication profile varies during their execution and also the profiling phase needs to be executed each time parameters such as number of processes or input data size change. Next, the topology of the machine is abstracted using the *hwloc* library in a tree. Last but not least, *TreeMatch* computes the placement of the processes, that is to which processing unit each will bind to. In this last phase, *TreeMatch* iteratively calculates how to group the processes for each level of the topology. The high level idea is the following: At each level of the topology we group the processes taking into consideration their communication profile into groups of size that is determined by the arity of the nodes in the next level of the topology. For example, in the level of cores the next level could be the level of shared caches and if the arity of that level was 4, meaning that each cache is shared by 4 cores then we would make groups of size 4. The calculated groups for the current level become virtual processes that are used for the next level. So in the next level we would have virtual processes each of which is a group of processes that was computed in the previous level and whose communication profile is the sum of the communication profiles of its constituent processes. The iterations continue until we compute a single group for the first level of the topology. The important thing is how groups are formed. To from the groups of size k for a particular level, we form a graph where each vertex is one set of k processes (so we have as many vertices as there are subsets of size k of the virtual processes) and we add an edge between each pair of vertices whose respective subsets share a vertex. The reason for this is that we want to select subsets of size k for the current level and we want each virtual process to belong to exactly one subset. So by adding those edges then we can rely on finding an independent set on the constructed graph which means that we select vertices with no edge between them and this translates into subsets of virtual processes with no virtual process in common. However, this simple variant of the independent set application to the constructed graph does not take into consideration the communication profile of the virtual processes. Intuitively, we would like

to favor a subset of virtual processes that result in lower total communication. To that end, we add weights to each vertex of the graph where the weight of a vertex is equal to the sum of the communication profiles of each virtual process in the vertex's group minus the amount of communication saved by grouping the virtual processes represented by that vertex together. Therefore, if the virtual processes communicate a lot then their weight will be small. Consequently, the minimum weight variant of the independent set problem will favor vertices of small weight which means that it will group virtual processes that will benefit the most from being grouped together. But since this variant is a NP-Hard problem and inapproximable at a constant ratio the authors propose some heuristics: (1) The *smallest-values-first* heuristic uses a greedy algorithm to finding an independent set that picks vertices in increasing order of their weight, (2) the *largest-values-last* heuristic also sorts the vertices in increasing order of their weight and chooses an independent set where the largest index of the vertices chosen is minimized, and lastly (3) the *largest-weighted-degrees-first* heuristic sorts the vertices according to the average weight of their neighbours in decreasing order and then finds an independent set greedily picking vertices in that order (the rationale is that when a vertex is picked in such an order then its neighbours cannot be chosen later and this means that vertices with large weight will not be chosen).

In Cruz et al. [2019] the authors describe the *EagerMap* algorithm to solve the mapping problem, i.e deciding on which processing unit each task should execute based on information about (1) the communication profile of the application which is given by the communication matrix that captures the amount of communication between each pair of tasks, (2) the topology of the architecture and in particular which levels are shared, like shared L3 caches, and the links between elements of the architecture, and, last but not least, (3) the load profile of the application consisting of the load of each task measured in number of CPU instructions executed. EagerMap attempts to take advantage of structured communication whereby tasks are partitioned into groups and most communication occurs within a group. The first step of the *EagerMap* algorithm is to form groups for each shared level of the architecture in a hierarchical fashion. Then, those groups are mapped to the topology of the architecture. To partition K tasks (or subgroups of tasks) into groups for a particular shared level of the architecture that consists of L elements, *EagerMap* takes advantage of the communication matrix and the load profile for those K tasks using a greedy algo-

rithm. In more detail, the algorithm begins by accumulating the load for all K tasks and deciding for the load that the next group will be responsible for by dividing the remaining load by the number of remaining groups to form. Once the load for the next group has been decided, the algorithm proceeds on to forming that next group by choosing among the free tasks (i.e those that haven't already been assigned to previous groups) such that their cumulative load doesn't exceed the load assigned to that group and the communication between tasks of that group is maximized. That is achieved using a greedy approach as follows: among the remaining free tasks, the next task to be assigned to the group is that one with the maximum amount of communication with the tasks already assigned to that group. Once the groups for the current level of the architecture is formed before proceeding on to the next level, *EagerMap* recreates the communication matrix and load profile for the newly formed groups using the communication and load profiles of the current level. One important detail is that *EagerMap* never creates more groups of tasks than there are elements of the current level of the architecture. The last stage of *EagerMap* is to map the groups to the topology of the architecture in a top-down fashion. Assuming that we are at some level consisting of L elements then because we know that for that level we have at most L groups we assign each group to a separate element of the current level and proceed recursively in a top-down fashion.

Chapter 3

Background on Phoenix++

3.1 Architecture of The Phoenix++ Runtime System

This section contains a description of each of the components of the *Phoenix*++ runtime system.

Synchronization This component provides an OS agnostic API over common synchronization primitives used by the runtime system. These primitives are implemented in header file `sync.h` and are:

Lock Two implementations of a lock type are provided with an API offering the `acquire` and `release` methods. The first implementation is based on the `pthread_mutex_t` type and the second one is an implementation of the *MCS* lock.

Semaphore The semaphore implementation is based on the `sem_t` type provided by *POSIX* and provides the `wait` and post methods.

Locality This component is implemented in header file `locality.h` and provides the notion of a *locality group* that is equivalent to a *NUMA node*. The implementation is based on the *libnuma* library for linux but makes certain machine-specific assumptions (i.e like that CPU ids are contiguous within the same NUMA node). The method `loc_mem_to_lgrp(const void *addr)` can be used to obtain the identifier of the locality group in which the address specified by the parameter is located at. Also, method `loc_get_lgrp ()` can be used to obtain the identifier of the locality group to which the calling thread belongs.

Processor The `processor.h` provides an API to bind threads to cores. Method `proc_bind_thread (int cpu_id)` binds the calling thread to the CPU core whose OS index is specified by the parameter `cpu_id`. Method `proc_unbind_thread()` can be used to unbind a thread from its CPU core and, lastly, the method `proc_get_cpuid()` returns the OS index for the CPU core of the calling thread.

Scheduler This component provides various thread mapping policies and is implemented in header file `scheduler.h`. The purpose of a thread mapping policy, which is abstracted by the `sched_policy` class, is to assign *logical* thread identifiers, ranging from 0 to $NumThreads - 1$, to the OS-specific CPU core identifier, in order for them to be used to bind the threads to those CPU cores using the *processor* component. This functionality is implemented by the `thr_to_cpu(int thr)` method of some subclass of the `sched_policy` class. Such concrete subclasses provide specific thread mapping policies, like spreading the threads equally among the sockets, or first filling one socket entirely before utilizing the next one. The implementation of those concrete classes is machine-dependent and on the assumption that the OS is *Solaris*.

Task Queue This component is implemented in header file `task_queue.h` and source file `task_queue.c`. This component provides a *task queue* per thread and an API to *enqueue* and *dequeue* both *map* and *reduce* tasks to those task queues. This API is implemented by class `task_queue`. An array of task queues is allocated, one for each thread. Moreover, to ensure that accesses to those task queues are thread-safe, `task_queue` also allocates an array of **locks** one for each task queue. The `enqueue(task_t const& task, thread_loc const& loc, int total_tasks=0, int lgrp=-1)` method inserts the task passed as parameter to either the task queue at the index specified by the `loc` parameter or to some index that

depends on the number of task queues and the total number of tasks to be inserted in the task queues. The `dequeue(task_t& task, thread_loc const& loc)` method searches the task queues for tasks. The search begins with the task queue of the calling thread trying to obtain a task from the *front* of that task queue, and then cycling through all other task queues trying to steal a task from the *back* of those task queues. In all cases, the thread first obtains the lock for each target task queue before accessing it.

Thread Pool The thread pool component is implemented in header file `thread_pool.h` and in source file `thread_pool.cpp`. $NumThreads$ threads are created with logical identifiers from 0 to $NumThreads - 1$. The thread pool uses a thread mapping policy specified by the *scheduler* component to bind those threads to CPU cores using the binding functionality of the *processor* component. The same thread pool is used during both *map* and *reduce* phases. In order to specify which method each participating thread of the thread pool should execute, the `thread_pool` class which implements the thread pool, provides the `set(thread_func thread_func, void** args, int num_workers)` method that changes the current function to be executed by the thread pool. In order to begin the *map* and *reduce* phases and, accordingly, to wait for their termination, the `thread_pool` class provides the `begin` and `wait` methods respectively. Each participating thread of the thread pool executes the `loop(void *arg)` method and executes the assigned thread function each time a start signal has been received by the `begin` method.

Containers and Combiners There is a tight integration between the *containers* and *combiners* components, which are implemented, respectively, in header files `container.h` and `combiner.h`. Each container uses an *input type* that is equivalent to a thread local version of that container and used in isolation by a thread during the map phase. Specifically, the map reduce scheduler, requests from the container one thread local *input type* container for each one of the threads in the thread pool using the method `input_type container::get(thread_id)`. This thread local *input type* container is passed to the `emit()` method to add *(key,value)* pairs. At the end of the map phase, the thread local *input type* container is added to the *global* container so that it can be used during the reduce phase. This is accomplished using the `container::add(thread_id,input_type)` method. For each key, the containers store a *combiner* for the values associated with that key. For the reduce phase, the threads need to reduce the values for one key

and for that they use the `iterator begin(out_index)` method of the container. The iterator type is responsible for collecting the values for that key from all threads in the container and provide a single iterator over all those values. The iterator provides the `bool next(K& key, output_type& values)` method that is used to iterate over the *(key,value)* pairs. For each key we gain access to its values through the combined interface (i.e `output_type = Combiner<V>::combined`). The containers offered are: `has_container`, `array_container` and `fixed_hash_container`. A combiner object is used to group all values for one particular key. The `buffer_combiner` queues up all elements to be reduced at the end and the `associative_combiner` combines the values into a single value at the moment they are added into the container. The `buffer_combiner` uses many combiners per key, i.e one for each thread. When time comes to reduce those combiners (i.e when an iterator is requested) the `buffer_combiner` starts collecting all those combiners into a single *combined* object using the combiner.combineinto(combined) method. To combine the values during iteration, the combined object provides an iterator API with a `next` method that traverses all the values in the combined object. The difference of the `associative_combiner` to the `buffer_combiner` is that the combined iterator will return a single combined value from all internal combiners.

Map Reduce Scheduler The *map reduce scheduler* is responsible for creating the thread pool, the task queue as well as the global container. Then it executes in sequence the map phase, the reduce phase and lastly the merge phase. The `map` phase creates a map task for each *chunk* of input data and enqueues it in the task queue. After all map tasks have been created, the workers in the thread pool are given the signal to start using the `start_workers` method. After the workers have finished the map phase, the reduce tasks are generated, one for each row of the global container. Then the workers are signalled to start again now executing reduce tasks. The last step is to execute the map phase which is accomplished by again creating merge tasks in the task queue and starting the workers in the thread pool to execute them. In order to use the same thread pool for all three phases, the thread pool provides a `set` method that can be used to specify the function to be executed by the worker threads and the input argument for each worker thread.

3.2 Writing MapReduce Applications with the Phoenix++ API

To illustrate the usage of the *phoenix++* we are using the *word count* application again. To begin with, we need to define types for the input, the key and the value. The input is of type `struct wc_string` which represents a piece of text. The key is a single word which is represented by the type `struct wc_word`. Since the key is used in a hash-based container where keys must be compared with each other, we need to overload the comparison operators for type `struct wc_word`. Moreover, the key type must be hashable and for that we provide a functor of type `struct wc_word_hash` that computes a hash value for a given word. The value represents the number of occurrences of each word and can be any integer type, like `uint64_t` for example. Those types are provided in listing 3.1

Listing 3.1: Types for word count application

```
1  // a passage from the text. The input data to the Map-Reduce
2  struct wc_string {
3      char* data;
4      uint64_t len;
5  };
6
7  // a single null-terminated word
8  struct wc_word {
9      char* data;
10
11     // necessary functions to use this as a key
12     bool operator<(wc_word const& other) const {
13         return strcmp(data, other.data) < 0;
14     }
15     bool operator==(wc_word const& other) const {
16         return strcmp(data, other.data) == 0;
17     }
18 };
19
20
21 // a hash for the word
22 struct wc_word_hash
23 {
24     // FNV-1a hash for 64 bits
25     size_t operator()(wc_word const& key) const
```

```
26      {
27          char* h = key.data;
28          uint64_t v = 14695981039346656037ULL;
29          while (*h != 0)
30              v = (v ^ (size_t)(*(h++))) * 1099511628211ULL;
31          return v;
32      }
33  };
```

To package the *word count* map reduce application we implement a class that inherits from the base class *MapReduce* or *MapReduceSort* depending on whether we want our output to be sorted by value or not. For this example we are going to use *MapReduceSort* because we want to output the *top-10* occurring words in the document. *MapReduceSort* is a template class implementing the *curiously recurring template pattern*, and requiring as template parameters the type of the deriving class, the type of the input, the type of the key, the type of the value and the type of the hash container to use. In our case we use the hash container type that is provided by *phoenix*++ but we instantiate it for our types and the custom hash functor type `struct wc_word_hash`. This class is named `WordsMR` and is provided in listing 3.2.

To specify the *reduce* function which in our case is just a summation, we are using the `sum_combiner` type template argument to the hash container type. Alternatively, we could implement a method `void reduce(key_type const& key, reduce_iterator const& values, std::vector<keyval>& out)` that receives a key and its values via the `reduce_iterator` iterator object, and we append the result of the aggregation to the output container `out`. The *map* function receives a piece of the input document of type `data_type`, splits it into its constituent words, and for each word produces the desired $(key, value)$ pair using the `emit_intermediate` function. To complete our `WordsMR` class we need to specify a function that *splits* the input document to smaller pieces each of which is passed as input to a separate invocation of the *map* function. This function is the `int split(wc_string& out)` function. We also need to provide a function to sort the output by value during the *merge* phase.

Listing 3.2: The WordsMR class

```
1  class WordsMR : public MapReduceSort<WordsMR, wc_string, wc_word, uint64_t,
       hash_container<wc_word, uint64_t, sum_combiner, wc_word_hash>
2  {
3      char* data;
```

```
4       uint64_t data_size;
5       uint64_t chunk_size;
6       uint64_t splitter_pos;
7
8   public:
9       explicit WordsMR(char* _data, uint64_t length, uint64_t _chunk_size) :
10          data(_data), data_size(length), chunk_size(_chunk_size),
11              splitter_pos(0) {}
12
13      void map(data_type const& s, map_container& out) const
14      {
15          for (uint64_t i = 0; i < s.len; i++)
16          {
17              s.data[i] = toupper(s.data[i]);
18          }
19
20          uint64_t i = 0;
21          while(i < s.len)
22          {
23              while(i < s.len && (s.data[i] < 'A' || s.data[i] > 'Z'))
24                  i++;
25              uint64_t start = i;
26              while(i < s.len && ((s.data[i] >= 'A' && s.data[i] <= 'Z') || s.
                    data[i] == '\''))
27                  i++;
28              if(i > start)
29              {
30                  s.data[i] = 0;
31                  wc_word word = { s.data+start };
32                  emit_intermediate(out, word, 1);
33              }
34          }
35      }
36
37      int split(wc_string& out)
38      {
39          /* End of data reached, return FALSE. */
40          if ((uint64_t)splitter_pos >= data_size)
41          {
42              return 0;
43          }
```

```
44
45          /* Determine the nominal end point. */
46          uint64_t end = std::min(splitter_pos + chunk_size, data_size);
47
48          /* Move end point to next word break */
49          while(end < data_size &&
50              data[end] != ' ' && data[end] != '\t' &&
51              data[end] != '\r' && data[end] != '\n')
52              end++;
53
54          /* Set the start of the next data. */
55          out.data = data + splitter_pos;
56          out.len = end - splitter_pos;
57
58          splitter_pos = end;
59
60          /* Return true since the out data is valid. */
61          return 1;
62      }
63
64      bool sort(keyval const& a, keyval const& b) const
65      {
66          return a.val < b.val || (a.val == b.val && strcmp(a.key.data, b.key.
                data) > 0);
67      }
68  };
```

Last but not least, we need to instantiate our class in order to execute the *word count* application. This is illustrated in listing 3.3. In the illustrated code the variables `fdata` is a pointer to the input document's contents which have been *memory mapped* to the main memory, and the variable *finfo* is of type *struct stat* which provides OS specific information for the input file.

Listing 3.3: Executing the word count mapreduce application

```
1  std::vector<WordsMR::keyval> result;
2  WordsMR mapReduce(fdata, finfo.st_size, 1024*1024);
3  mapReduce.run(result)
```

3.3 The Phoenix++ Reduce Phase Algorithm

Figure 3.1 showcases the memory organization of the global container structure using by the *Phoenix++* runtime. During the *map* phase, the *emit* function uses the local container of the executing thread to store the *(key,value)* pair. At the end of the *map* phase, each participating thread adds its local container to the *global* container maintained by the runtime so that there exists a global view over the *(key,value)* pairs during the *reduce* phase. The *global container* is a two dimensional array with as many columns as there are threads in the thread pool and as many rows as the hash space size for the keys. Each column can be viewed as a local hash container for the thread whose logical identifier is specified by the index of that column. That is, if a thread t_i emits a key that hashes to row j, then that key is added to cell $[j][i]$. During the *reduce* phase, when a thread performs a reduction over the values for a specific key, it needs to iterate over all values produced from all participating threads for that particular key. This is achieved by iterating over the cells whose row is specified by the hash of that key. The reduce algorithm consists of producing one reduce task for each row of the *global container*, distributing them over the *task queues* and having the threads execute those tasks. Considering the suitability of this reduce phase algorithm for *NUMA* architectures in accordance with the factors affecting performance as we have discussed them in chapter chapter 2, we conclude that the iteration over one row for reducing all the values produced from all threads for that row results in increased memory traffic from remote locations.

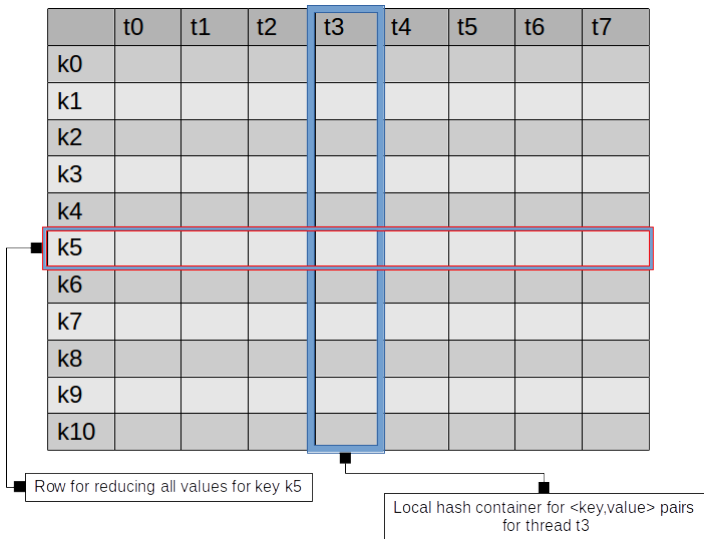

Figure 3.1: Memory organization for the global container storing *(key,value)* pairs. Each thread t_i uses the cells with column index i. A key is stored at the row specified by its hash value modulo the row size.

3.4 Related Work

In this section we begin by briefly discussing the Mao et al. [2010] implementation of the MapReduce parallel programming model. In *Metis* the authors target MapReduce applications which involve a relatively large number of intermediate key/value pairs and a relatively low amount of computation, that is, situations in which the run time is not dominated by the application code in functions Map and Reduce, but by the overhead of the library itself. Metis uses as an intermediate data structure a hash table with a b+-tree in each entry in order to get the benefits of both a hash table and a tree. To avoid the copy phase between the map and the reduce phases, all threads use the same size for their local hash tables and rely on the b+-tree for good lookup time on each entry of the hash table. That strategy of avoiding the copy that happens in Phoenix++ at the end of the map phase to the beginning of the reduce phase is something that we could also adopt to Phoenix++ as well. In another note, in Arif and Vandierendonck [2015] they use OpenMP parallel loops to implement the map phase with or without a task construct within the parallel for loop. Then, they use the OpenMP 4.0 feature for user defined reductions for the reduce phase. However, they state that depending on the data type of the reduction object the user defined reductions may not be evaluated in parallel in current OpenMP implementations. Therefore, performance and applicability of the OpenMP application model depends

on the MapReduce application. In such cases they had to resort to custom solutions in order to avoid using expensive synchronization objects like locks and/or critical sections. We try instead to optimize further the *Phoenix++* runtime in order to get the benefits of its simplicity with an increased efficiency.

Chapter 4

Improving the Reduce Phase of Phoenix++

In this chapter we describe the methods we evaluated for improving the reduce phase of the Phoenix++ implementation on NUMA architectures. The first set of methods rely on hierarchical algorithms and the second set of methods consist of task distribution policies that dictate to which thread a reduce task should execute. Implementation details are provided in the next chapter.

4.1 Hierarchical Tournament-Based Reduce Agorithms

In order to minimize the amount of memory traffic that needs to be read/written from remote NUMA nodes, we propose a hierarchical approach to the reduce phase that is based on the well-known tournament barrier algorithm.
Since the *global container* is two-dimensional we have in essence two dimensions over which we can partition and produce smaller reduce tasks that range over a sub array of the global array structure.

The Horizontal Approach The *horizontal* approach, as showcased in figure 4.1, consists of two phases. In the first phase we restrict the reduction only within a single NUMA node hence avoiding inter-node communication. To accomplish that we perform the following modifications:

- Restrict work stealing in the task queue subsystem to victim threads within the same NUMA node as the thief thread.
- For each key row produce as many reduce tasks as there are NUMA nodes. The reduce task for a key row iterates only over the columns of the threads that belong to the same NUMA node as the thread executing that reduce task.

For the second phase, we operate as the original reduce algorithm. That is, we produce one reduce task for each key row. For that reduce task, the thread executing it still iterates over the entire set of columns, but for each NUMA node there exists only one column that contains the output of the reduce phase from the first phase.

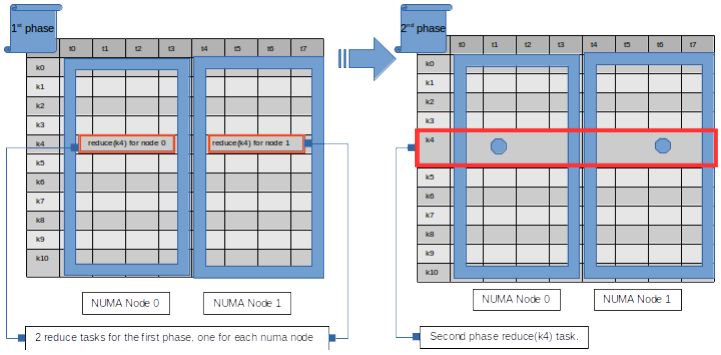

Figure 4.1: The 2-phase horizontal approach to the tournament-based reduce algorithm. In the first phase, we produce 1 reduce task for each row and for each NUMA node separately. In the second phase, we produce 1 reduce task for each row and all NUMA nodes combined.

The Vertical Approach The *vertical* approach, as showcased in figure 4.2, also consists of two phases but each phase is hierarchical in nature. We refer to the first phase as *intra-node reduction phase* and the second phase as the *inter-node reduction phase*. For both phases we use a *tournament-based hierarchical reduction with binary fan-out* algorithm as the reduction algorithm.

Intra-Node Reduction Phase A *local* reduction is executed by each NUMA node separately. At the end of the reduction, we have one *winner* from each NUMA node that has performed the reduction over all keys for the set of values contained in its home NUMA node.

Inter-Node Reduction Phase A *global* reduction having the winners of the previous phase as participating threads.

Possible Improvements A straightforward extension to the aforementioned algorithms, is to consider more levels of the hierarchical cc-NUMA architecture in addition to the level of the NUMA nodes. For example, we could restrict reduction within private and shared caches prior to the level of the NUMA node.

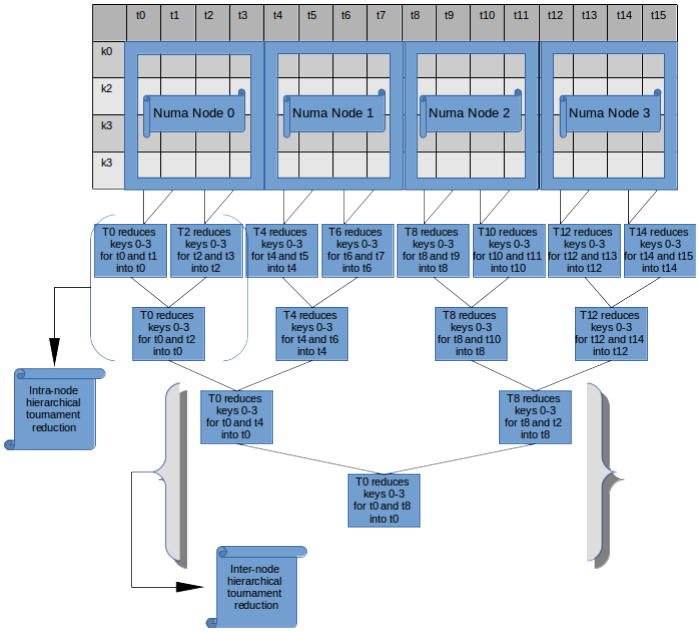

Figure 4.2: The intra-node and inter-node reduction phases of the vertical approach. Each phase consists of separate executions of a tournament-based hierarchical reduction with binary fan-out. To begin with, each NUMA node performs a local reduction and, then, a global reduction is performed from the winners of each local reduction.

4.2 Task Distritbution Policies for the Reduce Phase

The current reduction algorithm generates one task per row of the global container and the thread that picks that task makes a reduction over all columns for that particular row. In this section we describe different strategies, called *task distribution policies* on how to assign those tasks to threads. These strategies are supplemented by *thread mapping policies* and work stealing victim selection policies.

4.2.1 Thread Mapping Policies

A *thread mapping policy* specifies how $NumThreads$ threads are mapped to the computational units of the target NUMA system.

Contiguous We first fill up fully one numa node before proceeding to the next one.

NUMA-Spread We equally split the threads among the available numa nodes.

With a *contiguous* thread mapping policy we have faster communication and synchronization between the worker cores because most of them share at least a last level cache. On the other hand, assigning all threads to one numa node entails memory contention on the local memory controller of that numa node, and as a result we may not be able to take advantage of the full bandwidth of that numa node. To remedy this situation, we can use the *numa-spread* thread mapping policy, whereby we take advantage of all available numa nodes and we split the threads among them. Hence, there is less memory contention on each numa node but, on the other hand, there is more communication and synchronization cost among threads residing in distinct numa nodes.

4.2.2 Work Stealing Victim Selection Policies

When a worker runs out of work from its own local queue it becomes a *thief* and starts searching for work in the queues of other workers. We can specify different policies on how a worker thief chooses its victim which we call *work stealing victim selection policies*. In the following, we assume that we have $NumThreads$ threads with identifiers in the range $[0, NumThreads - 1]$.

No The thief doesn't perform actually no work stealing.

Sequential The thief chooses each victim thread sequentially in *thread-id* order. In more detail, a thief thread with identifier tid, $0 \leq tid < NumThreads$, chooses its victims in this order: $tid + 1, tid + 2, ..., NumThreads - 1, 0, 1, ..., tid - 1$.

Random The thief chooses randomly with equal probability from among its co-workers.

Numa-Aware The thief thread steals successively from different numa nodes starting for its own numa node. Each time it advances to a new numa node, it steals randomly from the threads belonging to that numa node. We would use this approach when we want to restrict remote communication as much as possible by first draining all work from our local numa node and then proceeeding to the next numa nodes.

Numa-Only The thief thread steals randomly only from threads belonging to the same numa node as itself. We would use this approach if we want to eliminate complete remote memory references.

4.2.3 Description of the Task Distribution Policies

In this section we describe policies on how to distribute reduce tasks over the worker threads.

Random-Based This is a simple rudimentary approach to reduce task distribution, whereby each reduce task is assigned uniformly at random to one of the available workers. With this approach we can potentially achieve good load balancing among the workers since each worker will get roughly the same amount of reduce tasks. Nevertheless, we have no control on the communication cost and the amount of local and remote references each thread will occur.

Interleave-Based This is another simple rudimentary approach much like the previous one. Instead of assigning randomly each reduce task to one of the available workers, we just *interleave* them among them. That is, the $i-th$ reduce task is assigned to the thread with *thread-id* $i\%NumThreads$.

Locality-Based The purpose of this reduce task distribution policy is to assign each task to the thread that will incur the least access cost to it. The algorithm is actually very simple:

1. For each key in any order
    (a) Assign the key to the thread with minimum access cost to that key

The problem with this approach is that depending on the distribution of key data over the columns of the global container we may have *oversubscription*, that is the effect of assigning too much work on a small portion of the worker threads. To remedy this situation we can rely on the *work-stealing policies* but another approach would be to devise another reduce task distribution policy to counterattack that effect.

Balanced-Based With this reduce task distribution policy we attempt to avoid *oversubscription* by avoid assigning reduce tasks to already overloaded threads. However, we still want to take advantage of locality whenever possibly and therefore we need to prioritize reduce tasks according to their data volume and access cost and thread assigning in a similar manner. The algorithm is the following:

1. Sort the keys by their data volume
2. For each key in sorted order do
    (a) Sort the threads in increasing access cost for the current key
    (b) Assign the key to the first thread in the above sort order that is not too overloaded

Mixed Locality and Balanced Based A potential disadvantage of the previous *balanced-based* reduce task distribution policy is that a reduce task may not be assigned to a thread that has *exclusivity* to that key, meaning it has a large portion of that key's data in the numa node containing that thread. To that end, we add one additional rule to the thread assignment process of the *balanced-based* policy, whereby the first thread in the increasing access cost sorted order, i.e the thread with the least cost to the key, gets assigned the key regardless of whether it is overloaded or not, if and only if that thread has exclusivity to that key. The revised algorithm now becomes this:

1. Sort the keys by their data volume
2. For each key in sorted order do
   (a) Sort the threads in increasing access cost for the current key
   (b) If the least access cost thread has exclusivity to that key then assign the key to that thread. Otherwise, assign the key to the first thread in the above sort order that is not too overloaded

Mixed Locality and Interleave Based With this reduce task distribution policy we apply the *exclusivity* rule of the *Mixed Locality and Balanced Based* policy to the *Interleave-Based* policy. The revised algorithm is:

1. For each key in any order do
   (a) If the least access cost to the that key has exclusivity to that key then assign the key to that thread. Otherwise, assign that key based on the interleave policy.

Chapter 5

Implementation Details

The purpose of this chapter is to describe the changes and additions that i made to the *phoenix++* source code in order to implement the proposed algorithms.

5.1 Topology Related Subsystem

In order to obtain information about the topology of the NUMA system and perform thread binding i used the *Portable Hardware Locality* library.

Thread Binding To perform thread binding i choose for each *thread-id* the processing unit on which it is going to be executed, which is represtented by the type `hwloc_const_cpuset_t`. Then, i call the method `hwloc_set_cpubind` with appropriate flags that indicate that the thread binding must be strict, meaning that the operating system scheduler is not free to migrate that thread to another processing unit. That action is implemented in the `thread_binding` class.

Thread Mapping Policies The mapping of *thread-ids* to `hwloc_const_cpuset_t` is implemented by a subclass of the `thread_mapping_policy` class. That base class provides the interface for the mapping and subclasses implement specific thread mapping policies. Two subclasses have been implemented. The first, named `contiguous_thread_mapping_policy` , assigns the threads to the available processing units in sequence filling completely each NUMA node before proceeding to the next one. The second, named `numa_spread_thread_mapping_pol` , spreads the threads among the available NUMA nodes equally and then recursively among lower level hardware elements, like shared L3 and L2 caches until a processing unit is encountered.

The integration of those implementations of thread mapping policies and the current implementation of the *phoenix++* runtime is implemented as follows. The threading system that provides the workers to the map reduce scheduler is implemented by the `thread_pool.hpp` and `thread_pool.cpp` component. I added to the constructor of the `thread_pool` object one parameter for the `thread_mapping_policy` and one parameter for the `thread_binding` object. Both are passed as pointers in order to accept subclasses. When each thread is created in `thread_pool` it receives a *thread-id* and it uses the `thread_mapping_policy` object to obtain the `hwloc_const_cpuset_t` object that specifies to which processing unit it must bind do, and then uses the `thread_binding` object to bind itself to that processing unit.

Auxilliary details for hwloc In order to implement the thread mapping policies, the following *hwloc* methods were useful:

hwloc_get_type_depth To obtain the the level in the NUMA system where a particular type of hardware elements are. For example, to know where the processing units are in the NUMA hierarchy.

hwloc_get_nbobjs_by_depth To obtain how many hardware elements are in a particular level. For example, to know how many processing units or NUMA nodes exist.

hwloc_get_obj_by_depth To obtain a particular hardware element within a level. For example, to obtain the second NUMA node or the *k-th* processing unit.

hwloc_get_next_obj_by_type To iterate over the hardware elements in a particular level.

5.2 Task Queue System

The task queue system is responsible for providing the queues where the workers search for map and reduce tasks to execute. I have altered the order by which each worker searches for tasks in the queues. The task queue system is implemented in the component `task_queue.hpp` and `task_queue.cpp`. The `dequeue` method of the `task_queue` object is the implementation of the work stealing policy for each worker. Whereas in the original code each worker searches the queues in a predetermined order, i have added the ability to the `task_queue` object to use various work stealing policies. In order to do that, i have provided a `ws_victim_selection_policy` base class that provides the interface for the work stealing victim selection policy. Several subclasses have been implemented, among which are the `numa_aware_ws_victim_selection_policy` and `numa_only_ws_victim_selection_policy` concrete implementations. The interface exposed to the `task_queue` system is the method `next` that returns the sequence of thread-ids from which to steal. The `task_queue` object now accepts in its constructor a `ws_victim_selection_policy` object. The `dequeue` method uses that object to obtaint the next *thread-id* from which to steal.

The `numa_aware_ws_victim_selection_policy` policy dictates that a worker first steals from other threads within its own NUMA node and then from other threads that belong to other NUMA nodes in increasing distance order. To obtain the distance order of between NUMA nodes, the `numa_aware_ws_victim_selection_policy` policy uses an object of the class `topology_distance_matrix` which i implemented for that purpose. That class provides a single method that accepts two *thread-ids* and returns their distance in the NUMA system. In order to obtain the distances between processing units, i use the `hwloc_distances_get_by_type` method from *hwloc* to get the distances between each pair of processing units and then the method `hwloc_distances_obj_pair_values` to obtain the distance for a particular pair of processing units. Last but not least, to find the NUMA node where a processing unit is, i use the `hwloc_get_numanode_obj_by_os_index` method from *hwloc* that retrieves the NUMA node index as reported by the operating system and returns the respective *hwloc* object. To retrieve the operating system index of the NUMA node i use the `hwloc_cpuset_to_nodeset` method to convert the cpu index of the processing unit to the index of the NUMA node where it belongs.

5.3 Tournament Vertical Reduce Implementation

The vertical tournament reduce implementation is provided in `tournament_reduce.hpp`.

The existing *phoenix++* implementation of the reduce phase first generates the reduce tasks and then starts the workers. Each worker executes a `reduce_callback` method. To keep the same scheme of implementation, one reduce task is generated for each worker thread and they are requested not to perform work stealing in order to ensure that each worker thread executes only one reduce task. The reduce callback executes the `reduce` method of the `tournament_reduce` class implemented in `tournament_reduce.hpp`. For the implementation of the reduction in class `tournament_reduce` we need to know from which thread each thread will reduce at each level of the reduction and access to other objects like locks and barriers. These arguments are stored in the generated reduce tasks themselves and generated by the `tournament_reduce_args_generator` class provided in `tournament_reduce.hpp`. That class needs to group threads based on the NUMA node they belong to and for that i use the *hwloc* library.

5.4 Reduce Task Distribution Policies Implementation

To begin with, the task distribution policies require the knowledge of the amount of keys stored in the global container. To that end, i have implemented the `key_distribution` class in component `container_key_distribution.hpp` that is an array of equal size and shape as the global container and each cell contains the amount of keys stored in the respective cell of the global container. The `container` object creates an object of type `container_key_distribution` and each time data is added to the container, as in the `add` method, the `container_key_distribution` object is updated to reflect the added data. To accurately keep track of the amount of both keys and values i had to also update the `combiner` classes to keep track of the number of values they store. A reduce task distribution policy is represented by the abstract class `reduce_task_distribution_policy` which provides the following interface to the map reduce scheduler:

reduceTasksCount Get the number of reduce tasks to put in the queues.

getReduceTaskAt Get the *i-th* reduce task.

getReduceTaskDestAt Get the id of the task queue where to put the *i-th* task.

Therefore, the modifications to the reduce phase of the *phoenix++* implementation are minimal. That implementation constisted of a for loop that enqueued the reduce tasks to the queues and then a call to the `start_workers` method that signals the worker threads to begin executing the reduce tasks. I only had to change the for loop to enqueue the reduce tasks as reported by the subclass of the `reduce_task_distribution_policy` used.

Several concrete subclass implementations of the `reduce_task_distribution_policy` abstract class exist, among which are the following:

1. `interleave_based_reduce_task_distribution_policy`
2. `locality_based_reduce_task_distribution_policy`
3. `balanced_based_reduce_task_distribution_policy`
4. `mixed_locality_and_balanced_based_reduce_task_distribution_policy`

Those implementations use the `key_distribution` object provided by the global container in order to know how many keys exist in each cell of the global container. In order to compute the cost of accessing a particular cell of the global container the `topology_distance_matrix` object is used which returns the distance between the thread that makes the access and the thread that holds that cell of the global container. In that way, the reduce task distribution policies have been separated from the low level imlementation details provided by the *hwloc* library and use more portable and higher level of abstraction objects like the `topology_distance_matrix` and `key_distribution` objects.

Chapter 6

Experimental Evaluation

6.1 Machine Description

For the performance evaluations i used the following machine configuration.

parade A Dell EMC PowerEdge R840 server. This machine consists of 4 NUMA nodes each of which contains 32 processing units and 64 GB of RAM, for a total of 128 processing units and 256 GB of RAM. All processing units within a NUMA node share a 22MB L3 cache. The processing units within a NUMA node are organized in 16 cores each of which contains 2 threads. The two threads within a core share the L1 instruction and data cache (each 32 KB) and the L2 cache (1024 KB). The processor architecture is Intel Xeon Gold 6130.

parallax A Dell EMC PowerEdge R740 server. This machine consists of 2 NUMA nodes. The machine hosts 2 Intel Xeon Gold 6130 CPUs, each of which is equipped with 16 cores and 32 threads for a total of 32 cores and 64 threads. All cores within a CPU share a $22MB$ L3 cache. Each NUMA node has $32GB$ RAM for a total of $64GB$ RAM.

paragon This machine is comprised of 2 AMD Opteron 6161 CPUs each of which has 12 cores at $1.8GHz$ for a total of 24 cores. Each CPU package consists of 2 NUMA nodes each equipped with $4GB$ of RAM for a total of $16GB$ of RAM (each CPU has $8GB$ of RAM). All cores within a NUMA node share a $5118KB$ L3 shared cache memory.

6.2 Workload Descriptions

For the performance evaluations we used the associative sum combiner where a thread reduces all values mapped to the same key during the map operation, and during the reduce operation the threads reduce the values mapped to the same key that were produced by different threads.

A workload is determined by the following characteristics:

Total Number of Emits The total number of key value pairs to be emitted by the workload.

Key Range The key range is $[0, TotalNumberOfEmits)$.

Emit Filler Policy Determines what percentage of the total number of emits is to be produced by each thread. I used the following configurations:

Equal All threads make the same number of emits.

One-Numa-Heavy A large percentage of the total emits is made by threads belonging to one numa node only. This configuration is introduced to test the effects of unbalanced load.

Key Filler Policy Determines which key to emit for each of the emits a thread makes. I used the following configurations:

Equal-Prob At each emit all keys in the key range have an equal probability of being produced.

Disjoint-Subranges The keys are partitioned according to their final place in the buckets used at the reduce operation, and the buckets are disjointly partitioned to the threads. This configuration is introduced to test the effects of locality.

Since we are concerned with the reduce phase and a combiner is used at the map phase, each thread only emits each key only once. Therefore, in order to populate the global container with large amounts of data i need to enlarge the key range. I control the amount of data with the total emits to be made and as a result the key range is made equal to the total emits. The value associated with each key is around 8000 bytes and a key is a plain integer. For $32GB$ i need 4000000 emits and for $64GB$ i need 8000000 emits to be made. This determines the workload data size. Then i used 4 configurations for *emit-filler* and *key-filler* policies. The configurations are:

Emit Filler Policy	Key Filler Policy	Description
Equal	Equal-Prob	This represents the typical random case
Equal	Disjoint-Subranges	This represents the scenario where a thread has exclusitivity to some keys
One-Numa-Heavy	Equal-Prob	This represents the scenario where the load is unbalanced among the NUMA nodes

6.3 Evaluation Results

6.3.1 Superiority of the Task Distribution Policies over the Tournament-Based Approaches

In this section, we present evidence that the task distribution policies perform better compared to the tournament-based approaches. This can be seen in figure 6.1.

The *horizontal* approach is most similar to the task distribution based policies. The reason that it performs worse compared to the most closest approach which is the *interleave-based* task distribution policy, is due to the *barrier* in between the two phases and the extra overhead associated with the management of the data structures used to merge the keys since those data structures need to be created, filled and destroyed twice.

The *vertical* approach performs much worse than all other approaches because it too has extra overhead similar to the *horizontal* approach regarding the intermediate barrier between the two phases and the management of the merge data structures. However, the *vertical* approach has an additional disadvantage, that of having less threads to perform reduce work during the second phase. Especially in the kind of workload used in the experiment, which has too many keys, this plays an important role.

The next figures showcase the same results for the *parallax* machine using $16GB$ data size, where it can be clearly seen that the two tournament based methods perform

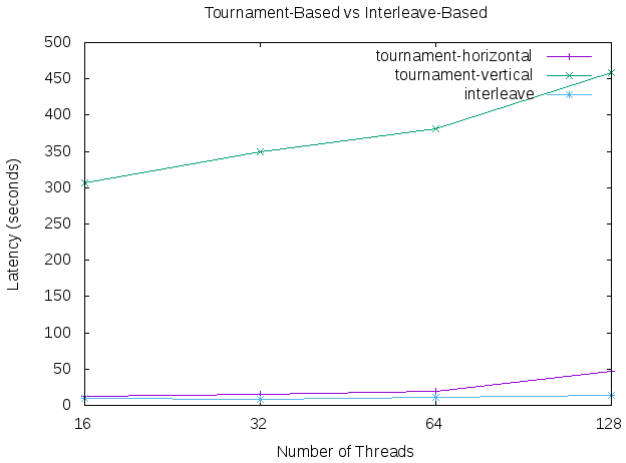

Figure 6.1: Latency in seconds for the reduce phase for 64 GB data size, Equal emit filler policy and Equal-Prob Key Filler Policy

consistently worse in all three cases.

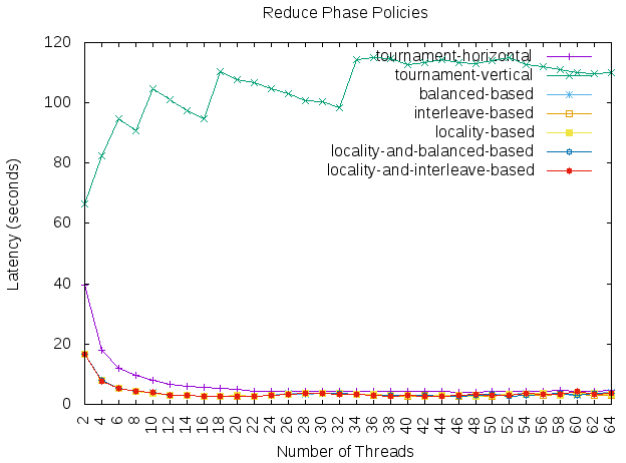

Figure 6.2: Latency in seconds for the reduce phase for 16 GB data size in parallax, Equal emit filler policy and Equal-Prob Key Filler Policy

6.3.2 Results for parade with 32 GB data size

Equal emit filler policy and Equal-Prob Key Filler Policy This case represents the typical random case where the distribution of keys among the rows and columns of the global container used during the reduce phase is totally random. In this case we do not expect to gain anything from the numa-aware reduce task distribution policies. In fact, we expect to loose a little performance due to the overhead associated with those policies.

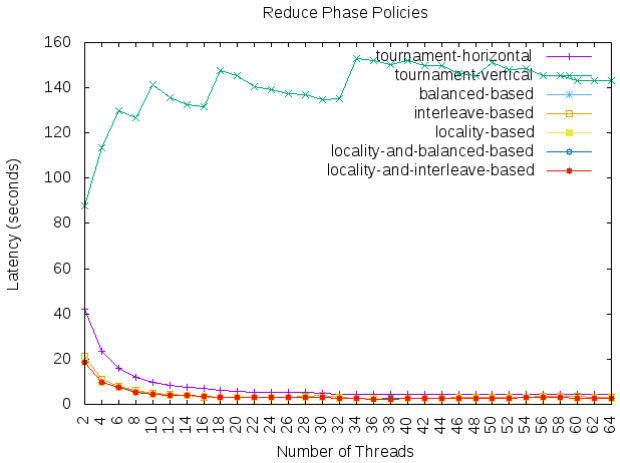

Figure 6.3: Latency in seconds for the reduce phase for 16 GB data size in parallax, Equal emit filler policy and Disjoint-Subranges Key Filler Policy

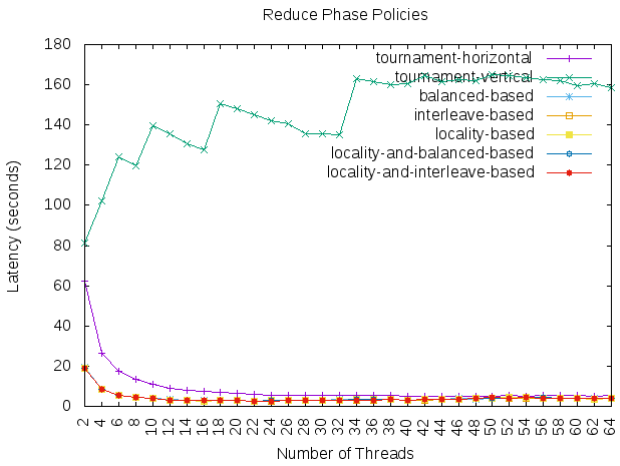

Figure 6.4: Latency in seconds for the reduce phase for 16 GB data size in parallax, One-Numa-Heavy emit filler policy and Equal-Prob Key Filler Policy

The latency in seconds for the reduce phase including the overhead for calculating the reduce task distribution for each policy is depicted in figure 6.5.

We can see the results in more detail in the following table.

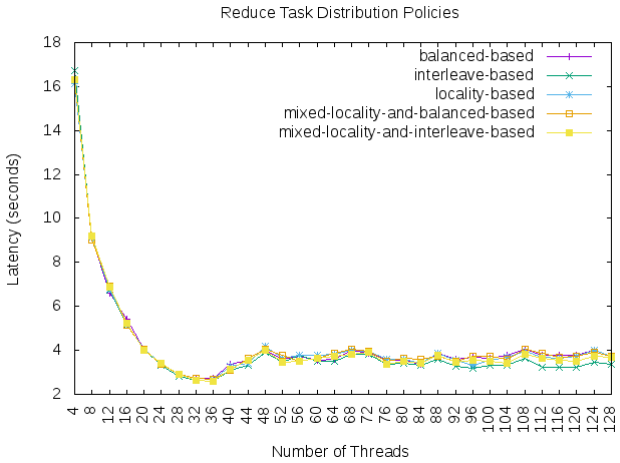

Figure 6.5: Latency in seconds for the reduce phase for 32 GB data size, Equal emit filler policy and Equal-Prob Key Filler Policy

NumThreads	balanced	interleave	locality	locality-balanced	locality-interleave
4	16.2661555025	16.7332478485	16.1398775815	16.3289557	16.325156577999998
8	9.111661975499999	9.121280459	9.1214282095	9.032698263	9.1859138915
12	6.625793674	6.723201617	6.7798502890000005	6.9366528745	6.883620022000001
16	5.433512949	5.138815534	5.169219404	5.1306766095	5.2222823075
20	4.0481774285	4.0492843725	4.050868277	4.0514895965	4.022246476
24	3.388849652	3.3211678275	3.401215155	3.3652237915	3.4110387180000004
28	2.9135616984999997	2.8159430524999998	2.919349384	2.914855799	2.908029307
32	2.7073231609999997	2.659490345	2.6497552325	2.7132312174999997	2.6514609875
36	2.718864892	2.6085971580000002	2.672250138	2.6795830814999997	2.60839149
40	3.3557263214999997	3.0998812310000003	3.275075967	3.0712456475	3.1492632025000002
44	3.540601787	3.3314660755	3.3997618995	3.647467302	3.5383319369999997
48	4.026589875	3.9355055215	4.178533807999999	4.0645516315000005	4.022773564
52	3.591362171	3.4452291329999998	3.6412572975	3.797028508	3.467818649
56	3.7219843519999998	3.7554648895	3.787086224	3.512058486	3.5238600599999996
60	3.532760346	3.5252443785	3.756755341	3.627297216	3.6277459705000004
64	3.6159698750000002	3.5192826735000002	3.834681959	3.8494807855	3.7131822215
68	4.0014073165	3.809073871	4.0161882885	4.0405370835	3.8055073774999997
72	3.8580711450000003	3.82425492	3.9823764345	3.9817610505000003	3.921851199
76	3.6135307545	3.3764502605	3.615334346	3.501741446	3.374402482
80	3.565205189	3.4145505189999996	3.5856262115	3.6544149150000003	3.517573284
84	3.3890513254999997	3.3092474265	3.4296412015	3.573717508	3.4317738
88	3.8711905925	3.5815290685	3.8789551785	3.7490814125	3.7628978799999997
92	3.6046693835	3.2823779825	3.5345663915000003	3.493089103	3.4639087615
96	3.6650389065	3.2040200800000003	3.3088709515000003	3.7108550275	3.5310633129999998
100	3.648010686	3.302303556	3.539762083	3.7185041305	3.5077066109999997
104	3.7774765500000003	3.305549535	3.6793132550000003	3.5959457265	3.410167274
108	4.0388992355	3.623246954	3.954736493	4.05387472	3.825161198
112	3.784549707	3.249946039	3.704752775	3.8657560655000003	3.6539764874999996
116	3.796292847	3.210207365	3.6423795255	3.7023293885	3.5321953544999998
120	3.755297382	3.2172323195	3.713374273	3.728622863	3.5076769480000003
124	4.0275539555	3.46633625	4.016979291	3.9275861709999997	3.7163723135
128	3.747281737	3.355288215	3.7229975635	3.744876186	3.6781396625

Equal emit filler policy and Disjoint-Subranges Key Filler Policy This case represents the scenario where threads exhibit locality to certain keys, meaning that, for example, one map worker was responsibility for the majority of keys produced for one key bucket. The latency in seconds for the reduce phase including the overhead

for calculating the reduce task distribution for each policy is depicted in figure 6.6.

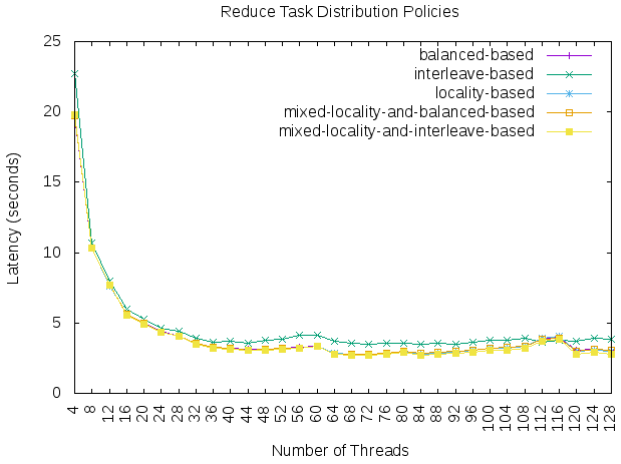

Figure 6.6: Latency in seconds for the reduce phase for 32 GB data size, Equal emit filler policy and Disjoint-Subranges Key Filler Policy

We can see the results in more detail in the following table:

NumThreads	balanced	interleave	locality	locality-balanced	locality-interleave
4	19.687326785	22.7304624965	19.7045343785	19.763066166999998	19.7990690275
8	10.325862896	10.6932500395	10.3101409605	10.318773625	10.312859876000001
12	7.668566317	7.9923733795	7.6314445395	7.6667226369999995	7.663789127499999
16	5.589628571	6.017512969	5.613625021500001	5.608220409	5.5696022355
20	4.962040891	5.2447188815	4.969394081	4.979019946999999	4.936453842
24	4.3844411585	4.6336943195	4.3797572825	4.373301087	4.3540077735
28	4.0557456965	4.4386107755	4.0575881875	4.047730252	4.0379581855
32	3.5575095230000002	3.948826586	3.50944623	3.5330318590000003	3.5049988460000003
36	3.235410589	3.6540050304999996	3.2396109510000004	3.2569416579999997	3.2183263495
40	3.1856893294999997	3.677700425	3.1766414985	3.1818665055000004	3.166885961
44	3.1030762165	3.589071049	3.0929783835	3.0955293435	3.0583891995
48	3.1200896065	3.8099944	3.1013150725000003	3.1185041680000003	3.0616662975
52	3.2102268809999996	3.8726788885	3.1774487075	3.1858986625	3.1479162335
56	3.2595422975	4.108815245000001	3.2258865825	3.2403303575	3.1846244365
60	3.3185806610000004	4.1107306185	3.3260357805	3.3162349300000002	3.31424552
64	2.842929216	3.7200131355000003	2.8441140655	2.809530976	2.746895233
68	2.7492233419999996	3.5362235144999996	2.767029471	2.7979962565000003	2.6875544185
72	2.8073682995	3.5123689575	2.7594880225000002	2.7690506685	2.6897926135
76	2.876121828	3.5331467035	2.8427658375	2.8537458155	2.761891779
80	2.9931275470000003	3.587594481	2.9668132035	2.969076715	2.8852200210000003
84	2.8287124285000003	3.5101032945	2.787128354	2.8374688975	2.6960400175
88	2.88794808	3.5544177845	2.8679323855	2.903017898	2.763785204
92	2.9931310495	3.5229603899999997	2.9453253555	2.9710694474999997	2.850064292
96	3.064424269	3.6275073664999997	3.029898549	3.0589106375000004	2.9293574204999997
100	3.2308264820000003	3.7705583595	3.2125644935	3.2285390415	3.061520937
104	3.2823640794999998	3.746823552	3.2190249719999997	3.2717442785	3.0607294645
108	3.369518625	3.925103393	3.316777218	3.3542142715	3.17513548
112	3.9337646655	3.6464694705	3.8900170115000003	3.8563960495	3.7320100590000003
116	4.0029642735	3.7669261714999998	4.044387988	3.9138046385000003	3.85122316
120	3.032542503	3.709184157	2.9575421	2.9606437115	2.7622068449999997
124	3.1642717275	3.896094824	3.0944602784999997	3.1248264715	2.9371124725
128	3.0877561350000002	3.8494298055	3.021930235	3.0473799305	2.812756615

As it can be seen by the figure, the simplest *interleave-based* policy performs worst among the other NUMA-aware methods, and those methods manage to scale better.

Examining the table above one can deduce that the best method is the *mixed-locality-and-interleave-based* and then the *locality-based* method. I

One-Numa-Heavy emit filler policy and Equal-Prob Key Filler Policy This scenario represents an unbalanced workload distribution where one NUMA node is overburden with the majority of the emits made. The latency in seconds for the reduce phase including the overhead for calculating the reduce task distribution for each policy is depicted in figure 6.7.

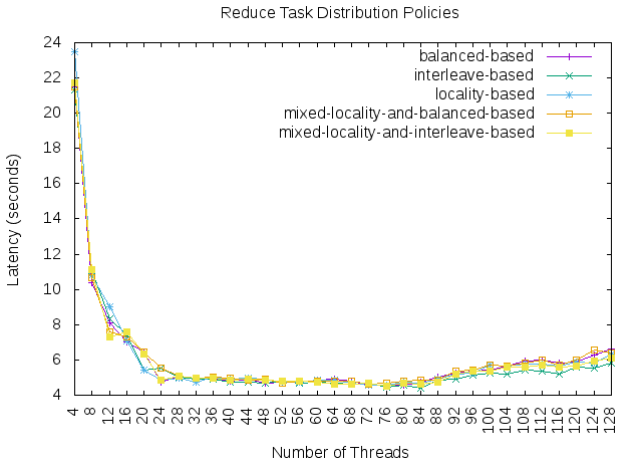

Figure 6.7: Latency in seconds for the reduce phase for 32 GB data size, One-Numa-Heavy emit filler policy and Equal-Prob Key Filler Policy

We can see the results in more detail in the following table:

NumThreads	balanced	interleave	locality	locality-balanced	locality-interleave
4	21.468672282	21.3039765395	23.4680811115	21.469354053	21.7171120725
8	10.3886202865	11.079965616500001	10.758482626500001	10.641690885	11.1085751265
12	8.099642170000001	8.348725717499999	9.01927632	7.5911743105	7.3116623569999994
16	7.033698799	7.425057831	7.100424003500001	7.331216387	7.583930195500001
20	6.4346720955	5.4219142389999995	5.426064537	6.464522519	6.36161444
24	4.7496045155	5.5378306175	4.8652394255	5.5133668435	4.874911408
28	5.1006895085	5.019565841	4.943643638499999	5.0819782725	5.066811384999999
32	4.961232364	4.8980142995	4.755139140000001	4.965307234	4.9439544385000005
36	4.9591534225	4.92915705	5.029347230000001	5.0386582205	4.98528589
40	4.8238387945	4.7603914295	4.898661553	4.942227752	4.8657833450000005
44	4.924722989499999	4.735430318	4.949032023	4.8647335755	4.928358834
48	4.6926546215	4.6947386820000006	4.925647273999999	4.8998616315	4.8415636105
52	4.744277628	4.802172329499999	4.741867601	4.6947338345	4.784099178
56	4.7484082555	4.6569797135	4.7387259545	4.820630081999999	4.7874984835
60	4.8510585975	4.721655913999999	4.873968581	4.808859864	4.7357544535
64	4.9077845270000005	4.7119386	4.8617954569999995	4.825118165499999	4.6394832595
68	4.7779861530000005	4.636177093500001	4.705311200000001	4.815251664	4.6506909489999995
72	4.597617720000001	4.699491004	4.5722904115	4.6546094965	4.6784580485
76	4.5470362075	4.4437174575	4.5494104275	4.701737411	4.5174615760000005
80	4.571441521000001	4.560626965	4.715675427	4.7852042365	4.6973960165
84	4.750654192000001	4.422425485	4.6885963175	4.8639921605000005	4.604223230500001
88	5.0271638890000006	4.8447543385	4.896832335	4.8418727100000005	4.7575950905
92	5.2618867305000006	4.929613164	5.2075015745	5.3784161515	5.217171228
96	5.3915938545	5.1326737085000005	5.226830858	5.452314680500001	5.3596608055
100	5.4273365259999995	5.225621528	5.6984717035	5.682452420500001	5.395137473
104	5.632647411500001	5.1962918795	5.6806433819999995	5.6625350655	5.5852282485
108	5.9253906485000005	5.4504656355000005	5.7619199430000005	5.828223547	5.602242727
112	5.9964693735	5.387541548	5.7758721835	6.00342242	5.722388792
116	5.8394143839999995	5.191963059000001	5.6485566255	5.695898379000001	5.593356878
120	5.9048759065	5.6083588585	5.9003235325	5.9710455620000005	5.645843256499999
124	6.275426920999999	5.5177385845	5.8060649615	6.5678452279999995	5.922269055
128	6.619004843500001	5.822076848	6.3194000085	6.418017966	6.132641139

In general all methods perform the same, with the exception of the *interleave-based* policy that performs better in high thread counts. The reason for this can be the fact that the *interleave-based* policy distributes faster the reduce tasks among the threads whereas the *locality-based* methods must rely on work-stealing which induces extra overhead. The *balanced-based* method due to its overhead doesn't match the performance of the *interleave-based* method.

6.3.3 Results for parade with 64 GB data size

Equal emit filler policy and Equal-Prob Key Filler Policy The latency in seconds for the reduce phase including the overhead for calculating the reduce task distribution for each policy is depicted in figure 6.8.

We can see the results in more detail in the following table.

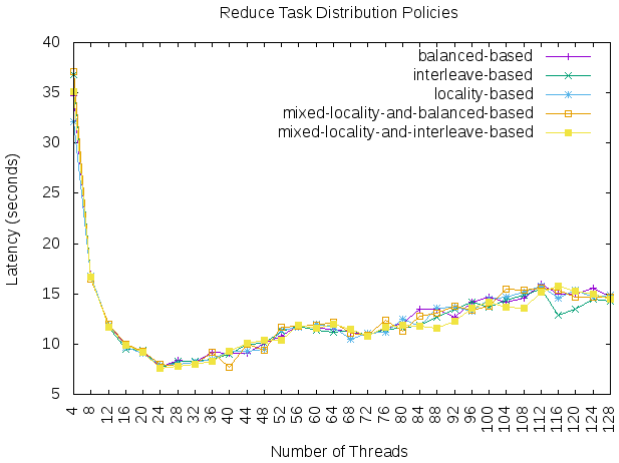

Figure 6.8: Latency in seconds for the reduce phase for 64 GB data size, Equal emit filler policy and Equal-Prob Key Filler Policy

NumThreads	balanced	interleave	locality	locality-balanced	locality-interleave
4	34.677529608	36.821672626	32.1671266685	37.088344378	35.0997213665
8	16.749758656	16.619612386	16.72555038	16.500827205500002	16.6749154825
12	11.809455245999999	11.653028399499998	11.909420725	11.9898071205	11.715453351
16	9.955870340499999	9.53611767	9.7121734305	9.984366651	9.8456540575
20	9.2657793305	9.3532108665	9.102440146	9.3132463055	9.2339810185
24	7.738913886000001	7.562351809	7.863905056	7.946846431	7.610961162500001
28	8.405104323500002	8.31213519	7.981517128	7.7857264435	7.818154969
32	8.209590233	8.3282956635	8.1981723505	8.020478376	7.988077536
36	9.2299669915	8.515301685499999	8.4791010545	9.1584927735	8.335419071
40	9.13659371	8.948233501499999	9.33057575	7.6808934099999995	9.320391467
44	9.0551944475	9.967500397	9.290926095	10.0083995735	10.0552979565
48	10.0624949515	10.067020769500001	9.500237849000001	9.390310486	10.41042675
52	10.8051922485	11.235036902000001	11.3494642515	11.650186781	10.358913577
56	11.873539625500001	11.764005863000001	11.671767645500001	11.7703137845	11.886132215
60	11.574686983	11.3435868205	11.936020756	11.846824120499999	11.545409202
64	11.468267178	11.1985197095	11.6886595995	12.2197463545	12.016568926
68	11.1706613765	11.4015031275	10.508865948	11.066778856500001	11.5142199815
72	10.926684282	10.8632011355	11.090759551000001	10.914190577	10.7389225935
76	11.617833176000001	11.451266552	11.20049761	12.330772272499999	11.657768601499999
80	12.113177858499999	11.613975098000001	12.483356738	11.3087069365	11.851580102
84	13.434732678	11.897625097999999	11.766549847	12.8015945035	11.790341671
88	13.4923687415	12.716665918	13.531675375999999	13.028127984000001	11.542086625
92	12.645428511999999	13.497193522	13.6928108845	13.8123302385	12.283367724
96	14.1969591995	14.147975453499999	13.267166725	13.3884895035	13.579218256499999
100	14.625700255	13.69032482	14.488692035	13.8180832105	14.174375946000001
104	14.2061163225	14.363820319	14.640646578	15.471457891	13.65614587
108	14.529753676999999	14.9074261465	15.1418582005	15.371165925500001	13.591373646000001
112	15.9535531075	15.4985253405	15.751974115500001	15.5820475515	15.1625304445
116	15.007130524	12.887268004	14.597336880499999	15.336159508	15.729864457000001
120	14.943190155	13.507511209	15.38661797	14.699290211000001	15.2807315385
124	15.587731372499999	14.444314299	14.6788312925	14.6522421565	14.945838956
128	14.702618634499999	14.275126181000001	14.907593967499999	14.720288106	14.4395417375

Equal emit filler policy and Disjoint-Subranges Key Filler Policy This case represents the scenario where threads exhibit locality to certain keys, meaning that, for example, one map worker was responsibility for the majority of keys produced for one key bucket. The latency in seconds for the reduce phase including the overhead

for calculating the reduce task distribution for each policy is depicted in figure 6.9.

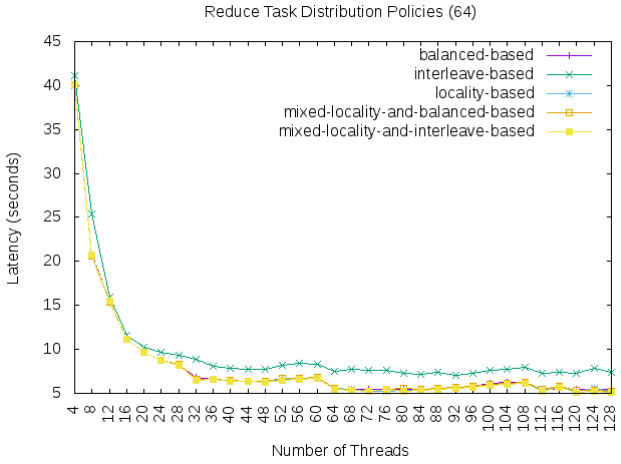

Figure 6.9: Latency in seconds for the reduce phase for 64 GB data size, Equal emit filler policy and Disjoint-Subranges Key Filler Policy

We can see the results in more detail in the following table:

NumThreads	balanced	interleave	locality	locality-balanced	locality-interleave
4	40.082530033	41.1428248235	40.025583513499996	40.0945491345	40.132937676
8	20.7930270995	25.3651353255	20.664143643499997	20.661101114	20.7710790115
12	15.507747346	15.928910784	15.473043761	15.36529367	15.526905558
16	11.124030193	11.6153368795	11.158130783499999	11.190638484499999	11.145797070499999
20	9.706668684	10.2832733465	9.6972073175	9.672478069499999	9.677859918500001
24	8.7829119845	9.695255528	8.763086246499999	8.765523587	8.722509698
28	8.213830815000001	9.364803533	8.268731982	8.264683416499999	8.223007584000001
32	6.8068996665	8.887090561	6.5540442305	6.591464069500001	6.5262431485
36	6.5896665604999995	8.049484953499999	6.58752244	6.571427116000001	6.547442531
40	6.4679976905	7.8425475295	6.4261494245	6.462867113	6.421169819999999
44	6.367247839	7.6923530145	6.3621124469999994	6.3994003710000005	6.329611763
48	6.3949106775	7.693676115000001	6.238600014499999	6.314250660500001	6.2409209315
52	6.582910829999999	8.1463715	6.6197674504999995	6.654450292	6.512379166500001
56	6.6307632420000004	8.394571969000001	6.63959487	6.6813016105	6.6027577184999995
60	6.7611689345	8.3160816575	6.796483794	6.803738195	6.7057377119999995
64	5.5797097255	7.5621137015	5.563605014	5.556207247	5.510569555
68	5.417566646999999	7.7109579964999995	5.442108516	5.3650061545	5.2922414745
72	5.5012232195	7.677072871	5.1821564155	5.218596239	5.259771011
76	5.429455737	7.6078987415	5.246261231	5.439814267999999	5.4243834755
80	5.4446145134999995	7.30259211	5.380024983	5.5193125395	5.281151943499999
84	5.427597094999999	7.116868520000001	5.355227641	5.418216483	5.320032729499999
88	5.5895843245000005	7.368817565	5.5233427035	5.564340358	5.4325452205
92	5.6822388835	7.080553765	5.680353774	5.6993891869999995	5.516795568
96	5.8255473095	7.2468327665	5.775440863	5.812953126	5.633561944
100	6.093079921999999	7.674173106	6.0320079945	6.0641844725	5.93340766
104	6.198487031999999	7.7325002695	6.173234072	6.1779070945	6.0190066375
108	6.2836718995	7.9297485695	6.2420635045	6.28962075	6.102418214
112	5.30328873	7.2339523815	5.319106861	5.422820895	5.2687147675
116	5.620547564	7.4355266575000005	5.790380212500001	5.7465733485	5.627979879
120	5.4277955344999995	7.235994584	5.2681731460000005	5.326486554000001	5.067428172
124	5.332100273	7.905528675	5.5236444745	5.395595023	5.1843084445
128	5.4127111625	7.357002245	5.2761366800000005	5.2493539160000005	5.08735125

Here we can again see the performance advantage of the NUMA-aware methods over the base *interleave-based* methods.

One-Numa-Heavy emit filler policy and Equal-Prob Key Filler Policy This scenario represents an unbalanced workload distribution where one NUMA node is overburden with the majority of the emits made. The latency in seconds for the reduce phase including the overhead for calculating the reduce task distribution for each policy is depicted in figure 6.10.

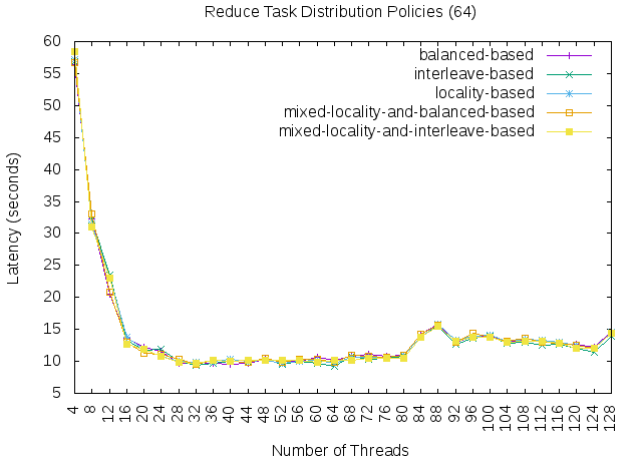

Figure 6.10: Latency in seconds for the reduce phase for 64 GB data size, One-Numa-Heavy emit filler policy and Equal-Prob Key Filler Policy

We can see the results in more detail in the following table:

NumThreads	balanced	interleave	locality	locality-balanced	locality-interleave
4	56.463624825500005	56.7943555965	57.209495652	56.7549786985	58.478843499
8	32.1967032275	32.0684719605	31.5194465975	33.1077600925	31.028366706
12	20.445610995	23.4547309075	23.0307076355	20.857040296	22.9773021295
16	13.67388358	13.230297548	13.819717927500001	13.064821958	12.6042637465
20	12.2616049415	11.689607535499999	11.9041244625	11.306844127	11.927950767
24	11.6030284365	11.8655335795	10.7651307195	11.046577739	10.7933370715
28	9.626626821	10.0914125835	9.9560933935	10.2589959365	9.918885658499999
32	9.9351220165	9.384238197	9.793642599999998	9.493912756	9.6746953715
36	9.653645899499999	9.707462940500001	9.965096378	10.143473754	10.1628849825
40	9.5231036955	9.9832258215	10.3484916065	10.040341707500001	9.9919911275
44	9.6585178345	9.860832042	10.0272368695	9.798619634000001	10.155965113
48	10.297624409	10.3090311055	10.109351650499999	10.4349132925	10.1199284825
52	10.014926443	9.487137929	9.768957255499998	9.8494671255	10.155001806000001
56	10.050370976	10.058735001999999	9.962650605	10.385509086999999	10.1186443735
60	10.68908372	9.7720437005	10.2415130285	10.353540808	9.866056085
64	10.189459845	9.2636486615	9.63127763	9.919163899499999	10.241357508
68	10.8126674185	10.804179911	10.5963388675	10.938652978499999	10.241718112000001
72	11.1680462125	10.270423134	10.6728469835	10.7501146535	10.537313263
76	10.7371740165	10.5480307645	10.6847122715	10.6562691535	10.488902034999999
80	10.9414184595	10.74159076	10.958679844999999	11.0281303265	10.518426548
84	14.1972207995	13.884077702999999	13.7001532855	14.29957481	13.7260069975
88	15.7534140405	15.445247925	15.757683387	15.6295484915	15.4870328865
92	13.041717883	12.749738473	13.2403069995	12.996534962	13.1900677835
96	13.9444854895	13.6364442505	13.901087905499999	14.4002154085	13.744404428
100	13.9180666695	13.877005011	14.064221667	13.8212558525	13.756438525
104	13.066928874	12.843891421	13.211753047999999	13.179755123	13.010617434
108	13.4868545775	12.9467056025	13.4488495895	13.620820404	13.2079349795
112	13.340665519	12.563232474500001	13.246154936	13.065686034	13.170775893
116	12.823281814000001	12.690553056999999	12.9428138625	12.855474209	12.9115954635
120	12.6774708	12.0444703805	12.476509015	12.443473969	12.003086521
124	12.232698016499999	11.432785377	12.072149549999999	12.102899219	12.010084091500001
128	14.747325475499999	13.9041026045	14.591283638	14.465425708	14.4788274215

As we can see from the figure all methods perform almost the same. This can be attributed to the fact that due to work-stealing all methods achieve balanced task distribution and due to the large size the overhead of methods like the *balanced-based* method is masked, whereas in the $32GB$ case the *interleave-based* method performed better and the overhead of the other methods was evident.

6.3.4 Results for parallax with 16 GB data size

Equal emit filler policy and Equal-Prob Key Filler Policy The latency in seconds for the reduce phase including the overhead for calculating the reduce task distribution for each policy is depicted in figure 6.11.

We can see the results in more detail in the following table.

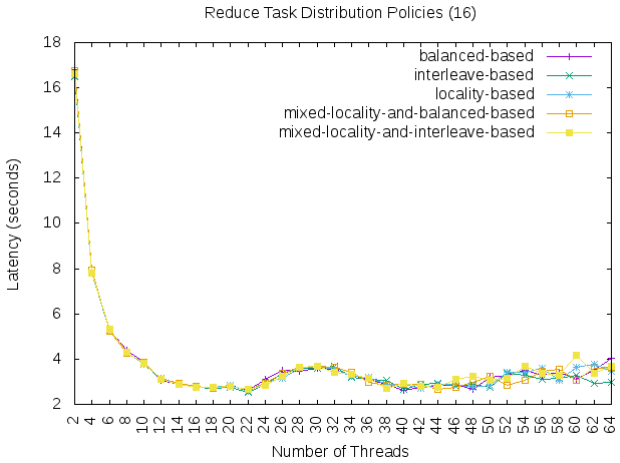

Figure 6.11: Latency in seconds for the reduce phase for 16 GB data size, Equal emit filler policy and Equal-Prob Key Filler Policy

NumThreads	balanced	interleave	locality	locality-balanced	locality-interleave
2	16.786427895666666	16.479983005	16.538170641	16.750685429333334	16.633060074
4	7.892304888333333	7.836049401333334	7.824653915666667	7.926503957666666	7.810202335333333
6	5.270982942333333	5.255421201333333	5.2315290186666665	5.245155384666667	5.321578406
8	4.396858621	4.283120759333333	4.287749750666666	4.269269845666667	4.320778886333333
10	3.8522237053333335	3.7602157923333333	3.7762734836666665	3.8639908316666665	3.836882302
12	3.0801359870000002	3.10799377	3.1309034000000002	3.106546256333333	3.094435537
14	2.9034554866666666	2.8888655973333335	2.9513096966666668	2.9257252903333333	2.9043660086666665
16	2.7812316213333332	2.7518107746666667	2.747823709	2.8090494573333333	2.7478675226666667
18	2.7121033083333335	2.7123751023333336	2.771912073333333	2.7083455703333335	2.7325262253333333
20	2.8240408826666665	2.769827674	2.836672211333333	2.810759423	2.7912270303333333
22	2.599366586	2.5434643843333333	2.588823893	2.6435534506666665	2.660713348333333
24	3.0880412953333334	2.8912901406666665	2.991043701	2.943650754	2.8509788316666667
26	3.5041611893333333	3.3787158583333334	3.173110316333333	3.375583267	3.2413473213333335
28	3.4591668666666666	3.5879768893333335	3.591419264333333	3.6530721293333333	3.6316496033333334
30	3.6154150363333333	3.576548966	3.6710796813333335	3.6663811576666667	3.674638207333333
32	3.539224813	3.682185312666667	3.5483939123333332	3.639095729	3.4237435216666667
34	3.2687213313333334	3.2144824796666667	3.2303868736666668	3.4263118726666666	3.3303602523333335
36	3.1072322206666665	3.1253975996666665	3.193357123333333	2.9601123886666665	3.138059306666667
38	2.8858600456666665	3.0667944879999998	2.8776488406666667	2.826686291	2.720589957666667
40	2.617082934	2.6537101406666666	2.8665505036666667	2.8685406253333334	2.9220978476666666
42	2.7691576393333333	2.8778874583333334	2.6896647909999998	2.884077779	2.785725810333333
44	2.833747357	2.9173307166666667	2.8677025836666665	2.6844983	2.745121347
46	2.9031009070000002	2.808482070666667	2.8868926466666665	2.756530086666667	3.0991304526666665
48	2.6816600856666666	2.8710412623333332	2.7557561316666668	2.958725896	3.258854396333333
50	3.2006256123333334	2.7480758026666665	2.863879138666667	3.249305897	3.0654768713333334
52	3.291243702	3.363987115	3.424649772	2.8238475696666665	3.1562290256666667
54	3.500209569	3.273586209333333	3.4069744836666667	3.0796498216666666	3.6710270303333337
56	3.2782117123333334	3.1277400903333334	3.615225027666667	3.4702674166666667	3.3875576003333334
58	3.4253094566666666	3.167288951	3.054341545333333	3.5506371123333333	3.2928909673333333
60	3.089826356666667	3.265682978	3.6361849989999997	3.0579455913333335	4.1881960860000005
62	3.5044781626666666	2.9418354653333334	3.7647481066666666	3.5861608333333335	3.3957334463333333
64	4.057415642333334	2.9336661056666667	3.4622537243333333	3.6598077563333336	3.6862785453333333

Equal emit filler policy and Disjoint-Subranges Key Filler Policy This case represents the scenario where threads exhibit locality to certain keys, meaning that, for example, one map worker was responsibility for the majority of keys produced for one key bucket. The latency in seconds for the reduce phase including the overhead

for calculating the reduce task distribution for each policy is depicted in figure 6.12.

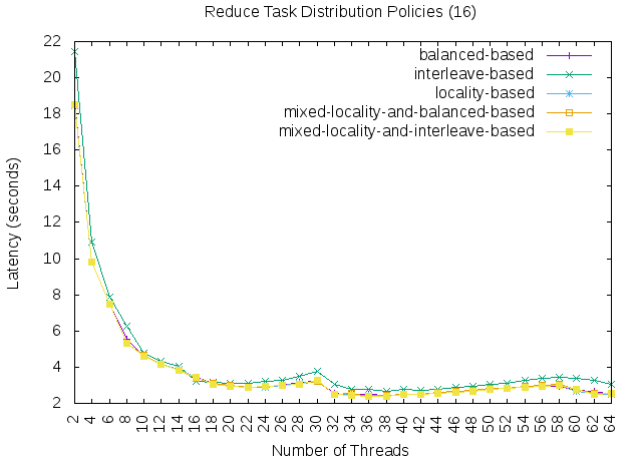

Figure 6.12: Latency in seconds for the reduce phase for 16 GB data size, Equal emit filler policy and Disjoint-Subranges Key Filler Policy

We can see the results in more detail in the following table.

NumThreads	balanced	interleave	localit	locality-balanced	locality-interleave
2	18.53946046966667	21.420404445666666	18.460628321	18.507045357666666	18.518536661666666
4	9.805268235333333	10.894967500333333	9.813067189333333	9.831142105	9.789495454333334
6	7.475483585	7.857575421666667	7.468917451333334	7.461005048333334	7.478274885
8	5.542279507333333	6.243151362666667	5.337988426333333	5.34882162	5.335937529333333
10	4.621346900333333	4.795403877666667	4.62688998	4.631958683333333	4.620590503
12	4.144907432666667	4.351337668666667	4.154608767666667	4.158865322333334	4.164609057666667
14	3.8709118553333335	4.044580764666667	3.85968714	3.851678927	3.8406303176666667
16	3.432263520333333	3.2375919063333334	3.4145498646666668	3.384347976	3.4443843136666668
18	3.089370116	3.183261679	3.0740253843333334	3.083982861	3.0682897343333333
20	2.9912015823333333	3.1147492113333333	2.9681198883333333	2.97540685	2.9604647816666665
22	2.8921355859999998	3.1020794453333336	2.8961739016666668	2.890815791666667	2.88486412
24	2.9221671836666667	3.1925857976666667	2.9066743086666667	2.9312780320000003	2.9191154646666666
26	2.998628892	3.2863582046666666	2.988049924	3.006199767	2.9924806690000003
28	3.1659367996666665	3.497884318	3.095565011333333	3.0654127083333336	3.047535413
30	3.2360528636666666	3.7914679666666666	3.2342852053333333	3.2054212243333335	3.2735780379999997
32	2.5397635663333333	3.055165248	2.525580466	2.5157952949999998	2.508367601
34	2.4621796236666667	2.780221496333333	2.5081005236666667	2.5294517963333334	2.4596767523333334
36	2.5005053166666666	2.767997115	2.467671327	2.405380315	2.402076888333333
38	2.463900499	2.682737944	2.433332538333333	2.43820541	2.4146922726666666
40	2.514952431333333	2.7816258966666667	2.4825694603333335	2.508590191	2.4729549353333335
42	2.5167784946666667	2.7402747326666668	2.516122020666667	2.5237900123333334	2.4839375823333336
44	2.6164121546666665	2.7958716193333335	2.5680524846666666	2.5977643853333334	2.576645179333333
46	2.6655007163333333	2.892063234666667	2.6525506	2.6652665576666665	2.6287840223333334
48	2.7422638706666667	2.9395581193333333	2.7065085926666668	2.7228091476666667	2.6891897533333333
50	2.817035709333333	3.044811517333333	2.7953982053333335	2.8150013916666667	2.7659672619999998
52	2.8569983293333334	3.117728147333333	2.8536234776666665	2.8544001816666666	2.81221451
54	2.9427761753333335	3.2801927093333334	2.932725666	2.9497576973333333	2.904593605
56	3.006564211666667	3.3747491716666667	2.961607137	3.024899441666667	2.923269997
58	2.941557882666667	3.451633618666667	3.015711414666667	3.059610301333333	2.996098041
60	2.7134386766666667	3.3909892723333335	2.666622633666667	2.774961075333333	2.7707922663333333
62	2.657878538	3.2704516133333335	2.5720884346666666	2.5778110523333333	2.481735892666667
64	2.516682006	3.0428412513333334	2.5230210266666666	2.5441667866666666	2.5555118576666667

Here we can again see the performance advantage of the NUMA-aware methods over the base *interleave-based* methods.

One-Numa-Heavy emit filler policy and Equal-Prob Key Filler Policy This scenario represents an unbalanced workload distribution where one NUMA node is overburden with the majority of the emits made. The latency in seconds for the reduce phase including the overhead for calculating the reduce task distribution for each policy is depicted in figure 6.13.

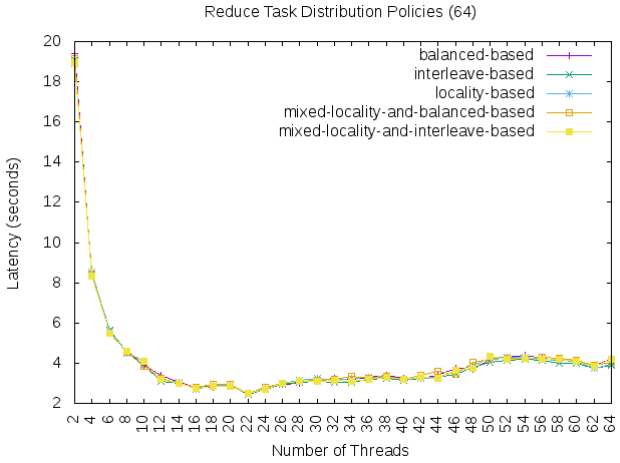

Figure 6.13: Latency in seconds for the reduce phase for 64 GB data size, One-Numa-Heavy emit filler policy and Equal-Prob Key Filler Policy

We can see the results in more detail in the following table:

NumThreads	balanced	interleave	locality	locality-balanced	locality-interleave
2	19.413821118333335	19.081338773666666	18.896907578333334	19.160612707000002	18.960139442666666
4	8.39588958	8.435618327	8.520511218333333	8.435876154	8.325520772666668
6	5.479406299333333	5.629542791	5.574157023	5.497623874333334	5.4862914069999995
8	4.609933825666666	4.544460985	4.4735303889999996	4.5422973833333335	4.5954994110000005
10	3.873598488	3.9141646686666665	4.087036257	3.8634738453333335	4.079074049333333
12	3.404913749	3.095731844	3.218559396333333	3.2257490066666668	3.186693937333333
14	3.0247508033333332	2.999385962333333	3.0380683176666667	3.007429579333333	3.0063717916666666
16	2.8075157163333335	2.7341574696666666	2.692124107	2.8159493760000003	2.7495187013333333
18	2.8203263986666665	2.902614113333333	2.8803105626666667	2.9631949656666667	2.866786354333333
20	2.9393545923333333	2.8808530366666667	2.8874719553333335	2.9528616646666666	2.840654038
22	2.5189670583333332	2.4273433803333333	2.5042033716666667	2.523259007	2.513106888333333
24	2.7589488896666667	2.7180871153333332	2.744821875	2.821057833333333	2.715179186
26	2.9500216833333335	2.9572980393333332	2.9775460663333333	3.009270577	2.975470744
28	3.0215357903333335	3.0570341026666665	3.134239017666667	3.09472586	3.0975689203333334
30	3.1402192496666665	3.108727072333333	3.2326835689999998	3.170278512	3.1034696536666666
32	3.2541898746666664	3.0473593013333335	3.166472163	3.202768219333333	3.1690096876666667
34	3.19234947	3.0516541923333333	3.258949196	3.3294827479999998	3.098336136
36	3.2918625946666666	3.191881958666667	3.301857414666667	3.2464534843333337	3.185771949
38	3.388201871	3.265745584	3.363868833333333	3.332636495	3.297339748
40	3.237484525333333	3.1685167896666666	3.1920451056666668	3.1799082370000002	3.196152667666667
42	3.233902272	3.238063578333333	3.2707778453333334	3.3982712133333335	3.3109231586666668
44	3.4199586203333334	3.295859991	3.2790830466666665	3.589166384	3.255949871
46	3.7528532009999997	3.468852259333333	3.5968239833333335	3.4674552596666666	3.6653089446666667
48	3.7002868593333336	3.7706257076666665	3.8720573243333334	4.057572874666667	3.769070746333333
50	4.181780612666667	4.064052898333333	4.223457634666667	4.203927003	4.344769030666667
52	4.286000934	4.156552495	4.299396511333334	4.2305016816666665	4.207047570333333
54	4.377483771	4.178450525	4.293449831666667	4.245746326	4.2569212793333335
56	4.225120373333334	4.164374077	4.212205362333333	4.292020746333334	4.178356286333334
58	4.139124183	4.014664925666667	4.138499121333333	4.22135407	4.174074332333333
60	4.124766688666667	3.9896611853333335	4.118694861333333	4.1624660376666665	4.093784508000001
62	3.874624748	3.7347858720000002	3.738377255666667	3.8870318796666665	3.8360932956666667
64	4.175104993666666	3.872723478333333	3.9320719706666667	4.056136696333333	4.18180757

As we can see from the figure all methods perform almost the same contrary to the small data size case for the *parade* machine using $32GB$ size where due to overhead the locality-aware methods performed a little worse in general.

6.3.5 Results for paragon with 4 GB data size

For the paragon machine we used only $4GB$ because we used the memory per NUMA node as an upper limit.

Equal emit filler policy and Equal-Prob Key Filler Policy The latency in seconds for the reduce phase for all policies including the overhead for calculating the reduce task distribution for each policy is depicted in figure 6.14.

We can again witness the superiority of the task distribution policies over the tournament based ones. In this case due to the small amount of memory involved all of the task distribution policies performed almost the same. We can see the results in more detail in the following table.

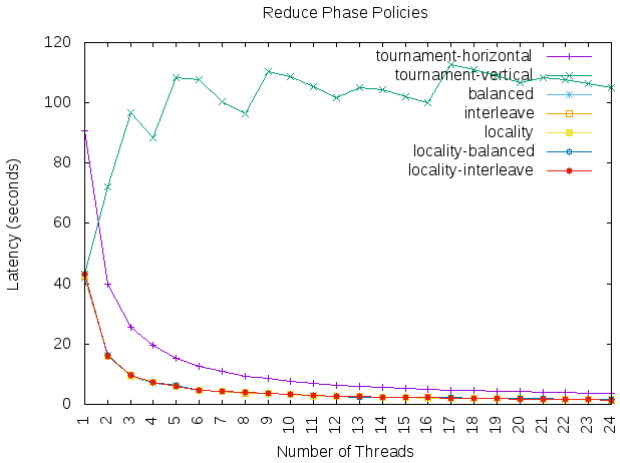

Figure 6.14: Latency in seconds for the reduce phase for 4 GB data size, Equal emit filler policy and Equal-Prob Key Filler Policy

Threads	horizontal	vertical	balanced	interleave	locality	locality-balanced	locality-interleave
1	90.63549134033333	43.272483384	41.916040718	42.383696615999995	42.980072553666666	43.212478614	43.090648801
2	40.021595354	71.97098942633333	16.144669431	16.102663419	15.938194391333335	16.148257871666665	16.007962572333334
3	25.583419441333334	96.861219301	9.429457008333333	9.507458612666667	9.46737654	9.521075516666667	9.595898434
4	19.513719860000002	88.46605083333333	7.145949401333334	7.280387982333334	7.252408600333333	7.297664836333333	7.208234923
5	15.416812676666666	108.29783369133334	6.173136538	5.933383321333333	6.047893973666667	6.159908495	5.935869062666667
6	12.748709910666667	107.70311547933333	4.7820062	4.7538305649999995	4.6708454669999995	4.753896609333333	4.649656488333333
7	10.966643022666666	100.360966	4.268790277666667	4.230761352	4.261199141666666	4.300853745666666	4.173351482666667
8	9.213544265	96.280304117	3.8456001896666665	3.7873890893333333	3.839532665	3.8732796560000002	3.8435463226666666
9	8.494754001999999	110.381888943	3.5889806666666666	3.522151411333333	3.5638903896666667	3.5801404073333334	3.580721763
10	7.7835837523333335	108.631731761	3.3225147273333335	3.226611594	3.290879311	3.302210839	3.315095246
11	6.862904507666666	105.290653957	2.8794564486666667	2.760916998	2.737010787	2.8874112303333335	2.839809437
12	6.174486703666667	101.68846789833333	2.604806623	2.565763711333333	2.5387573303333335	2.639217976333333	2.6222851206666666
13	6.044573086	105.191358167	2.7963489476666665	2.5594394586666667	2.5495239043333333	2.461456539	2.537277084333333
14	5.537694134	104.39699340599999	2.275608865666667	2.374752454	2.3974937333333335	2.4329964463333336	2.4188232496666666
15	5.264808152666666	101.97749137433333	2.3295044213333336	2.231715734333333	2.3591155010000002	2.315312387	2.3046740683333335
16	4.974551907	99.96013120433334	2.253520301	2.2133255	2.151065163	2.2173485123333334	2.2483972690000003
17	4.790418104666666	112.81385196466667	2.125752572	2.116195884	2.1267544653333332	2.1898535603333333	2.0548459046666667
18	4.815495768666667	110.887434693	2.0856683296666665	2.0170625246666667	2.102112228333333	2.0531578656666665	2.0961854733333336
19	4.382203024333333	109.02053135033334	2.055277900666667	1.9676353446666668	1.9673788166666666	1.9356632966666667	1.929774967
20	4.164595247333334	106.791067678	1.8869219176666667	1.830945912	1.8586519620000002	1.899068397	1.8171436626666666
21	3.9870851270000003	108.29344508	1.7020797986666667	1.7502550683333333	1.790028826	1.865817856	1.7555225813333333
22	3.8475802486666666	107.599913746	1.5960889903333333	1.595026011	1.574224877	1.573961608	1.646303752
23	3.7561397813333333	106.47771405333333	1.6560820183333333	1.5894847263333334	1.546393485	1.619088423	1.6796492966666667
24	3.4924706483333336	104.97931499466667	1.519804578	1.3935759626666666	1.5733790676666666	1.656640132	1.4361306153333333

Equal emit filler policy and Disjoint-Subranges Key Filler Policy onsibility for the majority of keys produced for one key bucket. The latency in seconds for the reduce phase including the overhead for calculating the reduce task distribution for each policy is depicted in figure 6.15.

We can see the results in more detail in the following table.

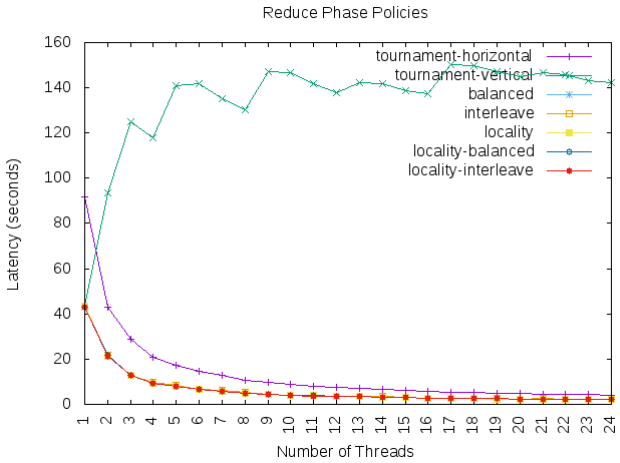

Figure 6.15: Latency in seconds for the reduce phase for 4 GB data size, Equal emit filler policy and Disjoint-Subranges Key Filler Policy

Threads	horizontal	vertical	balanced	interleave	locality	locality-balanced	locality-interleave
1	91.62044300733334	43.343438426	43.11938372633333	43.493503215666664	43.253090754	43.173374188333334	43.20182015066666
2	42.877155873333336	93.51649353566667	21.433504351333333	21.830331012666665	21.235263210333333	21.583757888666668	21.467677515666665
3	28.667718775	124.98901746533333	12.989703003333334	13.011396332	12.961631426333334	13.007899517333334	13.024078935333334
4	20.89342428033333	117.67616290866667	9.265499828666666	9.670919309333334	9.189835622333334	9.147413382	9.191824176333334
5	17.186192612	141.135307424	7.984113954666666	8.393622432333332	8.024748559999999	7.991725977000001	8.001657574
6	14.632478199333333	141.931382116	6.782692974333334	6.723542994	6.8241753313333335	6.77087026	6.7454990299999995
7	12.682169361	135.10941642400002	5.941401052	6.025229261	5.955288855666667	5.922888286666667	5.916845472
8	10.688759285333333	130.210436002	4.811971561	5.156670461333333	4.794081577333333	4.805062684666667	4.802095313
9	9.646997366666666	147.16171532466666	4.412869557333333	4.402718947333334	4.394353112666667	4.4200130736666665	4.396943084666667
10	8.750143116666667	146.90615940466665	4.062702012666667	4.065480782333333	4.065902595666667	4.054798723666667	4.069010237
11	8.050622208	141.974974944	3.771634128666667	3.8818339550000003	3.7694037303333334	3.775629282333333	3.7616052416666665
12	7.505918342666667	137.65582474266665	3.5403373670000002	3.550108929333333	3.5333673396666665	3.548194897	3.5239705076666668
13	7.017665216	142.432908031	3.3967427193333335	3.3749874763333336	3.3685519603333334	3.3791097433333332	3.368734561666667
14	6.595503323333333	141.71862997566666	3.288787279	3.3863590666666665	3.243406038333333	3.3026990339999998	3.268089181
15	6.300437159666666	138.714037389	3.1761386330000003	3.135255184	3.1244641033333336	3.1932946016666666	3.1614900586666668
16	5.733756197666667	137.35299934066668	2.7955521053333334	2.7044937066666668	2.7136395416666668	2.7282479596666667	2.681413376
17	5.319635445333333	150.06284065566666	2.632415739333333	2.6765374946666665	2.605717434666667	2.56965875	2.5188640153333335
18	5.097016679666667	149.75777278833334	2.5069452936666665	2.7552157703333333	2.485006113	2.4937039143333335	2.500978208666667
19	4.873624833333333	147.35812921166666	2.453795558	2.7763780856666664	2.381608641	2.4710700226666664	2.458057375
20	4.693303155333333	145.12259095666667	2.351225895333333	2.430292578	2.3432062276666668	2.352776543	2.4078416236666667
21	4.6014366283333334	146.74665191600002	2.295256254	2.4773770003333335	2.3057749486666665	2.3569783613333333	2.2996465716666665
22	4.363284175333333	146.00387841	2.2985089136666668	2.2219897956666665	2.2785252066666666	2.2861830316666665	2.2402235926666667
23	4.230093182	143.119887743	2.2397576643333332	2.3237822836666666	2.225419212333333	2.282097077	2.239185931333333
24	4.0997548476666665	142.42515073299998	2.296315377	2.3514512560000003	2.2640344856666665	2.224628319333333	2.1560544166666666

We can again witness the superiority of the task distribution policies over the tournament based ones. In this case due to the small amount of memory involved all of the task distribution policies performed almost the same. We can see the results in more detail in the following table.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we evaluated two sets of methods that are based on the well-known and historical tournament-based barrier algorithm, whereby we hierarchically reduce the (key,value) pairs first within NUMA nodes and then among all NUMA nodes. The second set of methods we evaluate are extensions of the current implementation of the reduce phase in the Phoenix++ runtime, whereby we implement various reduce task distribution policies that dictate to which thread a reduce task should be executed, where a reduce task implies the reduction over a specific range of keys. The purpose of those methods was to improve the reduce phase of the Phoenix++ runtime for the MapReduce programming model for shared-memry systems

We conclude that the first set of methods do not provide any performance advantages due to the extra overhead of synchronization and management of the intermediate data structures used during reduction. However, as far as the task distribution policies are concerned, in workloads that exhibit locality between the threads and the key ranges we can observe a performance improvement of up to 26.93% (i.e comparing interleave to the locality-interleave policy for 32 GB) and 30.85% (i.e comparing interleave to the locality-interleave policy for 64 GB) for 128 threads. For the typical

random case where the NUMA-Aware optimizations do not provide any benefits, we observe a performance decrease of 9.62%.

7.2 Future Work

Therefore, as future work we need to determine an efficient way of determining which of the methods to use based on the key distribution profile in the global container during the reduce phase to avoid the performance overhead for the typical random case. Furthermore, those results need to be checked against larger NUMA machines especially those with asymmetric interconnection networks. Last but not least, we need to evaluate those methods for other kinds of applications and compare with other implementations.

Bibliography

Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2002181.2002182.

Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 277–289, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-225. URL http://dl.acm.org/citation.cfm?id=2813767.2813788.

Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. Data sharing or resource contention: Toward performance transparency on multicore systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 529–540, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-225. URL http://dl.acm.org/citation.cfm?id=2813767.2813807.

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451157. URL http://doi.acm.org/10.1145/2451116.2451157.

Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. Coherence stalls or latency tolerance: Informed cpu scheduling for socket and core sharing. In *Proceedings of*

the 2016 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '16, page 323–336, USA, 2016. USENIX Association. ISBN 9781931971300.

Mihail Popov, Alexandra Jimborean, and David Black-Schaffer. Efficient thread/-page/parallelism autotuning for numa systems. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 342–353, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360791. doi: 10.1145/3330345.3330376. URL https://doi.org/10.1145/3330345.3330376.

Ananya Muddukrishna, Peter A. Jonsson, and Mats Brorsson. Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors. *Sci. Program.*, 2015, January 2016. ISSN 1058-9244. doi: 10.1155/2015/981759. URL https://doi.org/10.1155/2015/981759.

Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 125–137, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341219. doi: 10.1145/2967938.2967946. URL https://doi.org/10.1145/2967938.2967946.

Philippe Virouleau, François Broquedis, Thierry Gautier, and Fabrice Rastello. Using data dependencies to improve task-based scheduling strategies on numa architectures. In *Proceedings of the 22nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, page 531–544, Berlin, Heidelberg, 2016. Springer-Verlag. ISBN 9783319436586. doi: 10.1007/978-3-319-43659-3_39. URL https://doi.org/10.1007/978-3-319-43659-3_39.

Quan Chen and Minyi Guo. Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures. *ACM Trans. Archit. Code Optim.*, 12(2), July 2015. ISSN 1544-3566. doi: 10.1145/2766450. URL https://doi.org/10.1145/2766450.

Zoltan Majo and Thomas R. Gross. A library for portable and composable data locality optimizations for numa systems. *ACM Trans. Parallel Comput.*, 3(4), March 2017. ISSN 2329-4949. doi: 10.1145/3040222. URL https://doi.org/10.1145/3040222.

Ahmad Anbar, Olivier Serres, Engin Kayraklioglu, Abdel-Hameed A. Badawy, and Tarek El-Ghazawi. Exploiting hierarchical locality in deep parallel architectures. *ACM Trans. Archit. Code Optim.*, 13(2), June 2016. ISSN 1544-3566. doi: 10.1145/2897783. URL https://doi.org/10.1145/2897783.

Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Tumbler: An effective load-balancing technique for multi-cpu multicore systems. *ACM Trans. Archit. Code Optim.*, 12(4), November 2015. ISSN 1544-3566. doi: 10.1145/2827698. URL https://doi.org/10.1145/2827698.

E. Jeannot, G. Mercier, and F. Tessier. Process placement in multicore clusters:algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, 2014.

Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. Eagermap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems. *ACM Trans. Parallel Comput.*, 5(4), March 2019. ISSN 2329-4949. doi: 10.1145/3309711. URL https://doi.org/10.1145/3309711.

Yandong Mao, Robert Morris, and M. Frans Kaashoek. Optimizing mapreduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

Mahwish Arif and Hans Vandierendonck. A case study of openmp applied to map/reduce-style computations. In Christian Terboven, Bronis R. de Supinski, Pablo Reble, Barbara M. Chapman, and Matthias S. Müller, editors, *OpenMP: Heterogenous Execution and Data Movements*, pages 162–174, Cham, 2015. Springer International Publishing. ISBN 978-3-319-24595-9.

Mei-Ling Chiang, Chieh-Jui Yang, and Shu-Wei Tu. Kernel mechanisms with dynamic task-aware scheduling to reduce resource contention in numa multi-core systems. *J. Syst. Softw.*, 121(C):72–87, November 2016. ISSN 0164-1212. doi: 10.1016/j.jss.2016.08.038. URL https://doi.org/10.1016/j.jss.2016.08.038.

I. Ştirb. Numa-btdm: A thread mapping algorithm for balanced data locality on numa systems. In *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 317–320, 2016.

I. Ştirb. Numa-btlp: A static algorithm for thread classification. In *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 882–887, 2018.

Iulia Stirb. Extending NUMA-BTLP algorithm with thread mapping based on a communication tree. *Computers*, 7(4):66, 2018. doi: 10.3390/computers7040066. URL https://doi.org/10.3390/computers7040066.

C. Bordage and E. Jeannot. Process affinity, metrics and impact on performance: An empirical study. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 523–532, 2018.

Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot, and Thomas Ropars. Data and thread placement in numa architectures: A statistical learning approach. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362955. doi: 10.1145/3337821.3337893. URL https://doi.org/10.1145/3337821.3337893.

E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux. An efficient algorithm for communication-based task mapping. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 207–214, 2015.

M. Diener, E. H. M. Cruz, and P. O. A. Navaux. Locality vs. balance: Exploring data mapping policies on numa systems. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 9–16, 2015.

Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas. Reducing data movement on large shared memory systems by exploiting computation dependencies. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 207–217, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357838. doi: 10.1145/3205289.3205310. URL https://doi.org/10.1145/3205289.3205310.

R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 107–118, 2013.

W. Wang, J. W. Davidson, and M. L. Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 419–431, 2016.

S. Bak, H. Menon, S. White, M. Diener, and L. Kale. Multi-level load balancing with an integrated runtime approach. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 31–40, 2018.

Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, page 9–16, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307000. doi: 10.1145/1996092.1996095. URL https://doi.org/10.1145/1996092.1996095.

R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207, 2009.

C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.