Combining Recursively Parallel Runtimes with Blocked-based Dependence Analysis

Nikolaos Papakonstantinou

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Department of Computer Science
Voutes Campus
700 13 Heraklion, Crete
Greece

Thesis Advisors: Prof. *Angelos Bilas*, Dr. *Polyvios Pratikakis*

This work was partially supported by **Institute of Computer Science, Foundation of Research and Technology Hellas**

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Combining Recursively Parallel Runtimes with Blocked-based Dependence Analysis

Thesis submitted by
Nikolaos Papakonstantinou
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Nikolaos Papakonstantinou

Committee approvals: _____
Angelos Bilas
Professor, Thesis Supervisor

_____
Polyvios Pratikakis
Assistant Researcher, Thesis Supervisor, Committee Member

_____
Panagiota Fatourou
Assistant Professor, Committee Member

Departmental approval: _____
Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, June 2015

Abstract

In this work we combine recursive task-parallelism with dynamic dependence analysis to expose more parallelism from our programs. Early runtime systems as Cilk use the recursively task-parallelism but task synchronization is manual and the programmer is rensposible for defining the synchronization points. On the other hand, runtime systems such as BDDT and SMPSs also use dependence analysis to solve the dependencies between tasks, but they suffer from the *single master scaling problem*. We combine these two models and we present a dependence analysis algorithm for inferring runtime dependencies between recursively parallel tasks. We implement the dependence analysis in PARTEE, a scalable runtime system that supports implicit synchronization between nested parallel tasks. We explore the changes required for a Cilk-like runtime system to support task dependencies and evaluate the performance of the resulting runtime system. We find that in cases where task dependencies are irregular, PARTEE outperforms Cilk, a task-parallel runtime without implicit task synchronization, by up to 54%.

Περίληψη

Στην εργασία αυτή συνδυάζουμε τον αναδρομικό παραλληλισμό εργασιών με την δυναμική ανάλυση εξαρτήσεων για να εξάγουμε περισσότερο παραλληλισμό από τα προγράμματά μας. Παλαιότερα συστήματα χρόνου εκτέλεσης όπως το Cilk χρησιμοποιούν τον αναδρομικό παραλληλισμό εργασιών αλλά ο συγχρονισμός των εργασιών είναι χειροκίνητος και ο προγραμματιστής είναι υπεύθυνος για των ορισμό των σημείων συγχρονισμού. Από την άλλη, συστήματα χρόνου εκτέλεσης όπως το BDDT και το SMPSs χρησιμοποιούν την ανάλυση εξαρτήσεων για να λύσουν τις εξαρτήσεις μεταξύ των εργασιών, αλλά υποφέρουν από το πρόβλημα κλιμάκωσης μοναδικού χρονοδρομολογιτή. Εμείς συνδυάζουμε αυτά τα δύο μοντέλα και παρουσιάζουμε ένα παράλληλο αλγόριθμο ανάλυσης εξαρτήσεων για να συμπεράνουμε εξαρτήσεις κατά τον χρόνο εκτέλεσης μεταξύ αναδρομικών παράλληλων εργασιών. Υλοποιήσαμε την δυναμική ανάλυση στο PARTEE , ένα κλιμακώσιμο σύστημα χρόνου εκτέλεσης το οποίο υποστηρίζει έμμεσο συγχρονισμό μεταξύ εμφωλευμένων παράλληλων εργασιών. Ερευνήσαμε τις απαιτούμενες αλλαγές για ένα σύστημα χρόνου εκτέλεσης όμοιο του Cilk που να υποστηρίζει εξαρτήσεις μεταξύ των εργασιών και αξιολογήσαμε την απόδοση του συστήματος χρόνου εκτέλεσης που προέκυψε. Παρατηρούμε ότι σε περιπτώσεις που οι εξαρτήσεις των εργασιών είναι ακανόνιστες, το PARTEE υπερισχύει του Cilk, ενός παράλληλου συστήματος χρόνου εκτέλεσης χωρίς έμμεσο συγχρονισμό εργασιών, έως και 54%.

Acknowledgements

This work was carried out at the Computer Architecture and VLSI (CARV) laboratory of the Institute of Computer Science (ICS) of the Foundation of Research and Technology Hellas (FORTH), and was financially supported by a FORTH ICS scholarship.

There are several people I would like to thank. First of all I would to thank my thesis advisor and committe member, Dr. Polyvios Pratikakis for his support and guidance throughout my graduate studies. I would also like to thank the other two members of the committee, prof. Angelos Bilas and prof. Panagiota Fatourou for their time and effort they put to evaluate this work.

A special thanks also goes to Foivos S. Zakkak for his help and advice for the design and implementation of this work. In addition, I would like to thank Christi Symeonidou and Antonis Psathakis for reviewing this work and also for their comments both for the text and presentation.

Finally, I would like to thank my family, friends and colleagues for supporting me during my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Task-parallelism offers a high-level abstraction for expressing parallelism to the programmer compared to threads and processes. Hence, task-parallel programming models gain increasing traction with parallel programmers. Early task-parallel models, like Cilk [1], required manual synchronization using locks and barriers, whereas recent task-parallel runtimes support implicit synchronization using dependence analysis to discover and resolve task dependencies. Examples of such runtimes are the SMPSs [2] and BDDT [3] runtime systems. Such runtime systems, however, face what is known as the *single master scaling problem*. The dependence analysis and scheduling are executed by a single core, which often fails to create tasks fast enough to keep all the available *worker* cores busy. This leads to idling worker cores and bad performance at high core counts.

To address this problem, earlier task-parallel models like Cilk [1] use nested parallelism. Cilk uses nested parallelism to distribute the creation of parallel tasks and scale to higher core counts. With nested parallelism, each task can produce new children tasks, forming a task tree. Cilk programs create tasks recursively, using the `spawn` directive. For the synchronization of tasks, Cilk provides the `sync` directive, which blocks the current task until all its child-tasks reach completion. However, the lack of dynamic dependence analysis, reduces the exposed parallelism of some Cilk programs.

Figure 1.1 presents a Cilk toy example, that demonstrates the limitations of Cilk. In this example, there are three functions, `qux`, `foo`, and `bar`. The `qux` function takes two parameters and stores the value of the second to the first. The `foo` function takes three parameters and concurrently stores the value of the third to the first and the second, by *spawning* two instances of `qux`. The `bar` function *spawns* an instance of `qux` passing through its parameters.

In task-parallel programs, similarly to sequential programs, the execution starts from the `main` function. The only deference is that whenever a `spawn` directive is executed it creates a concurrent task that may run in parallel with the sequential `main`, much like a thread creation. Additionally, whenever a `sync` directive is executed the sequential `main` blocks until all its spawned tasks reach completion, similarly to thread-join. In Cilk-like programs this behavior is recursive—each task can also *spawn* new tasks and *sync* with them.

```
1  void qux(int *k, int *l) {
2    *k = *l;
3  }
4
5  void foo(int *x, int *y, int *z) {
6    spawn qux(x, z)[out:x, in:z];
7    spawn qux(y, z)[out:y, in:z];
8  }
9
10 void bar(int *k, int *l) {
11   spawn qux(k, l)[out:k, in:l];
12 }
13
14 int main(void) {
15   // ...
16   spawn foo(x, y, z)[out:x,y, in:z];
17   sync;
18   spawn bar(k, x)[out:k, in:x];
19   spawn qux(l, m)[out:l, in:m];
20   // ...
21 }
```

Figure 1.1: Cilk-like Code Example

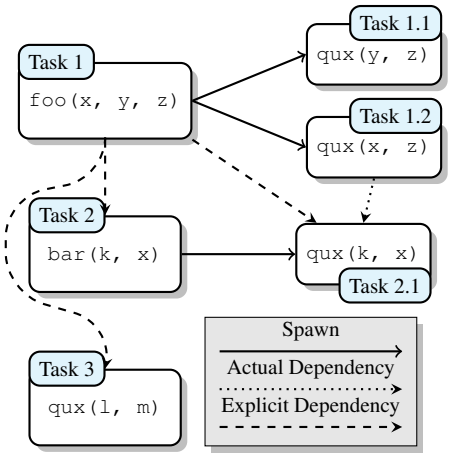


Figure 1.2: Task Dependency Graph

In our toy example the `main` function first *spawns* an instance of `foo` to store the value of `z` to `x` and `y`, and then *syncs*—blocks until the *spawned* instance of `foo` completes. After `sync`, `main` *spawns* one instance of `bar` to store the value of `z` to `k`, and one instance of `qux` to store the value of `m` to `l`. Note that in this example we annotate the parameters passed to each *spawning* instance with the keywords `in`, `out`, and `inout`. With `in` we mark the parameters read by the *spawning* function, with `out` those written, and with `inout` those that are both read and written.

In Cilk, the `sync` directive is used to resolve the dependency on `x`, between the spawns of `foo` and `bar`. This `sync` directive, however, also delays the *spawn* of the `qux` function which has no dependencies on the `foo` function's *spawned* instance. In Figure 1.2 we sketch the task dependency graph of the toy example presented in Figure 1.1. For each task instance we draw a rounded rectangle. We use solid arrows, to represent task *spawns* and demonstrate the parent-child relationship between the *spawned* tasks. We use dashed arrows to show the explicit dependencies between tasks, as a result of the `sync` directive. Finally, we mark *actual dependencies*—dependencies we need the runtime system to detect and resolve in order to correctly execute the program—with dotted arrows. Note that the notion of dependencies is transitive, meaning that for any task t that depends on a task t' all child-tasks of t also depend on t', we further discuss this property in chapter 2. In our sketch, for clarity, we omit all the explicit dependency edges starting from *Task 1.1* and *Task 1.2*. Instead we only present those explicit dependency edges starting from *Task 1*.

In Cilk, the use of the `sync` directive effectively adds an explicit dependency between the active child-tasks and the code following it. However, in our example the (curved) explicit dependency introduced between *Task 1* and *Task 3* is not an *actual dependency* and could be eliminated. The only *actual dependency* in our example is the one between

Task 1.2 and *Task 2.1*.

Motivated by similar scenarios and the fact that correctly instrumenting recursively parallel code with *sync* directives is tedious, we present a dependence analysis algorithm for inferring runtime dependencies between nested parallel tasks. As we further discuss in chapter 3, Cilk does not represent tasks directly but only through stack frames, thus it does not lend itself easily to implicit synchronization and runtime task-dependence analysis used in modern systems. As a result, we implement our dynamic dependence analysis in PARTEE, a scalable runtime system supporting nested task parallelism. PARTEE inherits the A-steal [4] and Blumofe's and Leiserson's [5] *work-stealing* algorithm for scheduling tasks. We evaluate PARTEE and show that its performance is comparable to Cilk and BDDT.

Specifically, the contributions of this work are:

- An algorithm for inferring runtime dependencies between tasks in Cilk-like, recursively task-parallel programs.
- A parallel region-based memory allocator, that increases the memory locality and concurrency of task-parallel runtimes.
- The development of a new double-ended queue algorithm called FCDQ, and its evaluation comparing with existing double-ended queue algorithms.
- The evaluation of PARTEE, a runtime system implementing the proposed dynamic dependence analysis using the proposed parallel region-based allocator.

The rest of this thesis is organized as follows. In Chapter 2, we present our dependence analysis algorithm for inferring dependencies between tasks in Cilk-like, nested task-parallel programs. In Chapter 3, we present the key features of PARTEE, our runtime system along with a parallel region-based memory allocator that increases the memory locality and concurrency of PARTEE and we discuss the limitations of the Cilk runtime system, regarding the implementation of the dynamic dependence analysis. In Chapter 4, we evaluate PARTEE and compare it to the Cilk and BDDT runtime systems. In Chapter 5, we discuss the related work and we conclude in Chapter 6.

Chapter 2

Dynamic Dependence Analysis

Similarly to the SMPSs and BDDT runtime systems, which are the state of the art, to avoid unnecessary edges in the task dependency graph we propose the use of data-flow annotations in the source code and a dynamic dependence analysis in the runtime system. The dependence analysis we propose in this section is different from the state of the art in the notion that it operates on nested task graphs.

In order to resolve dependencies between tasks we first need to define the task notion. In this work, similarly to SMPSs and BDDT, a task is considered to be an atomic unit of work—it cannot be interrupted—along with its memory footprint (the data it accesses). To express the kind of accesses performed to the memory footprint of the task we employ three different tags: *in*, *out* and *inout*. With *in* we mark the memory segments that are read by the task, with *out* those written, and with *inout* those that are both read and written.

There are also two other special tags, *safe* and *stride*. Memory segments tagged with *safe* can be omitted from the dependence analysis process, since they do not produce any dependencies between tasks. The *safe* tag is optional and is used as a hint to the runtime, either manually or using a compiler, in order to improve performance. Task arguments tagged with the *stride* tag are used to infer the memory piece of an array which the argument wants to accesses. Figure 2.1 illustrates a *stride* argument. The actual part of the memory, which the argument needs, is the hatched area included in the red rectangular, but the memory mapping enforces the dependence analysis to check the whole green area. The *stride* tag helps programmers to solve this problem and the dependence analysis to check only the hatched area in the red rectangular without checking extra blocks of the array, unnecessary to the argument. Note that every argument must have one tag of either *in*, *out* or *inout* and additionally can be tagged as *safe* or *stride*.

We proceed on checking whether the relationship between the associated access and that of the new task is (*a*) read-after-read (RAR), (*b*) read-after-write (RAW), (*c*) write-after-write (WAW), or (*d*) write-after-read (WAR). In the case of a RAR relationship we associate the new task with the corresponding memory segment, by appending its *task descriptor* to the *readers list* of that memory segment, and add the freshly *spawned* task to the task-queue. In all other cases there is a dependency between the *to-be-spawned* task and the task(s) associated with the memory segment, meaning that it cannot be spawned

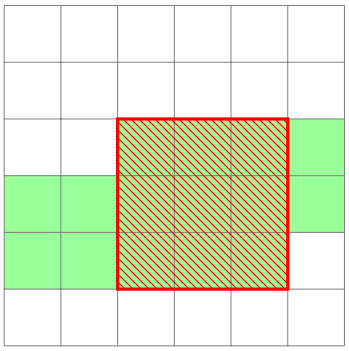


Figure 2.1: Stride Argument Example

until there are no more associated tasks with the corresponding memory segment.

```
1  void qux(int *k, int *l) {
2    *k = *l;
3  }
4
5  void foo(int *x, int *y, int *z) {
6    spawn qux(x, z)[out:x, in:z];
7    spawn qux(y, z)[out:y, in:z];
8  }
9
10 void bar(int *k, int *l) {
11   spawn qux(k, l)[out:k, in:l];
12 }
13
14 int main(void) {
15   // ...
16   spawn foo(x, y, z)[out:x,y, in:z];
17   spawn bar(k, x)[out:k, in:x];
18   spawn qux(l, m)[out:l, in:m];
19   // ...
20 }
```

Figure 2.2: Cilk-like Code Example without manual synchronization

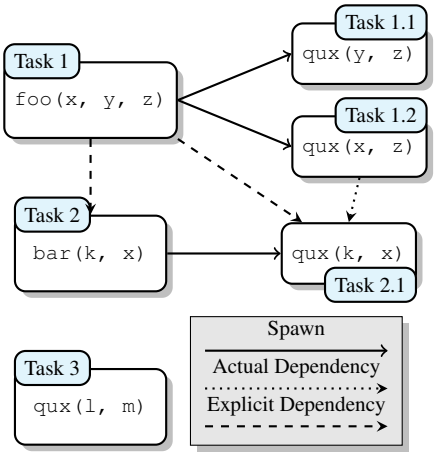
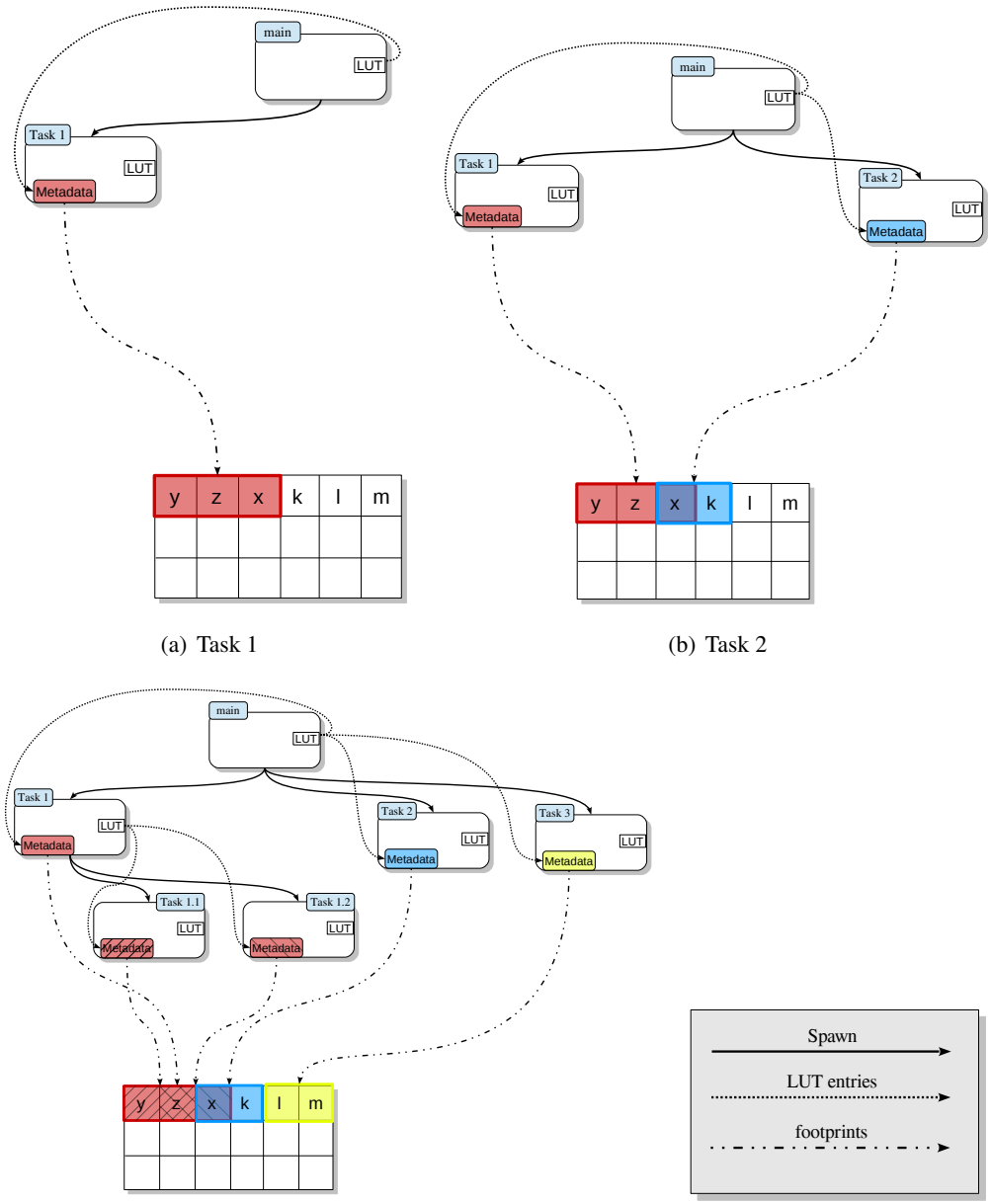


Figure 2.3: Task Dependency Graph

In this chapter, to argue about dynamic dependence analysis, we consider the `sync` directive in line 17 of Figure 1.1 to be absent. Figure 2.2 presents the same example without the `sync` directive. In this case, the `main` function will first *spawn Task 1*, then *Task 2* and finally *Task 3*. At the *spawn* of *Task 1* (Figure 2.4(a)) the corresponding *task descriptor* will be created and the memory segments it accesses (based on its footprint annotation) will be associated with its *task descriptor*. Note that *Task 1* can be directly *spawned*, since there are no other active tasks and thus no dependencies. That means, *main* and *Task 1* run *in parallel*. Later, at the *spawn* of *Task 2* (Figure 2.4(b)), we create the corresponding task descriptor and check, for each memory segment it accesses, whether it is already associated with any *task descriptors* or not. In our example, assuming *Task 1* is still running, the memory segment where `x` is stored will be associated with the task

descriptor of *Task 1*.



(a) Task 1

(b) Task 2

(c) Task 3

Figure 2.4: Task Graph

In our example, we will detect a RAW dependency (line 24 on Figure 2.6) between *Task 1* and *Task 2*, and add *Task 2* in the *notify list* of *Task 1*. Finally, at the *spawn* of *Task 3* (Figure 2.4(c)) we will proceed with the respective checks and find that there are no associations for the memory segments that *Task 3* accesses and will proceed with the

spawn.

Regarding the nested tasks *Task 1.1* and *Task 1.2*, the process is similar, however in this case we find an association for each of the memory segments where `x`, `y`, and `z` are stored with the *task descriptor* of *Task 1*. These associations however do not represent a dependency. They are the result of the programming model which requires ancestor-tasks to include, in their memory footprint, the memory segments included in all the memory footprints of their descendent-tasks. That is, for any task t, its memory footprint is the union of its own and its children's memory footprints. Two tasks are said to be *dependent* when the one is not a descendant of the other and the intersection of their memory footprints is not the empty set. This is required to ensure the correct execution order of tasks. For instance, assume that *Task 1* did not include `x` in its memory footprint, since it does not directly access it. In this case *Task 2* and *Task 2.1* could be spawned before the spawn of *Task 1.2*, since there would be no association of the memory segment where `x` is stored with any *task-descriptor*. In general, in the case where associations of memory segments with ancestor-tasks' *task descriptors* are detected by the dependence analysis, they are ignored. As a result, dependencies may be detected only between sibling-tasks or their descendent-tasks. Note, however, that a descendant-task's spawn depends on the spawn of its parent—if its parent is not spawned yet, then it cannot be spawned either. In the example of Figure 2.3, *Task 2.1* cannot be spawned before the dependency between *Task 1* and *Task 2* gets resolved. This property further narrows down the dependencies we need to detect and resolve between sibling tasks.

Figure 2.5 presents a graph with all spawned tasks, their entries in their parent's look-up tables (LUT), their footprints in memory and the conflicted memory segments between sibling tasks. Specifically, `main` task spawns 3 new tasks, *Task 1*, *Task 2* and finally *Task 3*. Each of these tasks defines its memory footprint. *Task 1* needs the red area, *Task 2* needs the blue area and *Task 3* needs the yellow area. *Task 1* and *Task 2* are conflicted in `x` block. *Task 1* spawns two new tasks, *Task 1.1* and *Task 1.2*, which define their memory footprints , red left hatched and red right hatched respectively. Both *Task 1.1* and *Task 1.2* need `z` block for read (`in` argument), thus they can be executed in parallel. When these two tasks finish, *Task 2* is scheduled and spawns *Task 2.1*, which defines its memory footprint (blue vertical hatched area) and scheduled because there are not any dependent tasks. *Task 3* can be scheduled at anytime because it has not any dependencies with the other five tasks, which means that *Task 3* can be executed in parallel either with *Task 1*, *Task 1.1*, *Task 1.2*, *Task 2* or *Task 2.1*. Figure 2.6 and Figure 2.7 present the psedo-code of our dependence analysis with more details.

To keep track of the dependencies and the memory footprints of tasks we employ a *task descriptor* comprising of: (*a*) the code to be executed; (*b*) a description of all its memory footprint (its memory accesses); (*c*) a *dependencies counter*, counting how many tasks this task depends on; (*d*) a *fired dependencies counter*, counting how many dependencies of this task have been fired—the task owning them, released them; and (*e*) a *notify list*—a list of *task descriptors* that have dependencies on this one. Our proposed dependence analysis requires that at run-time, whenever a new task is spawned, it gets associated with such a *task descriptor*. Our dependence analysis is able to trace back which tasks (if any) access a memory segment and the corresponding type of the access

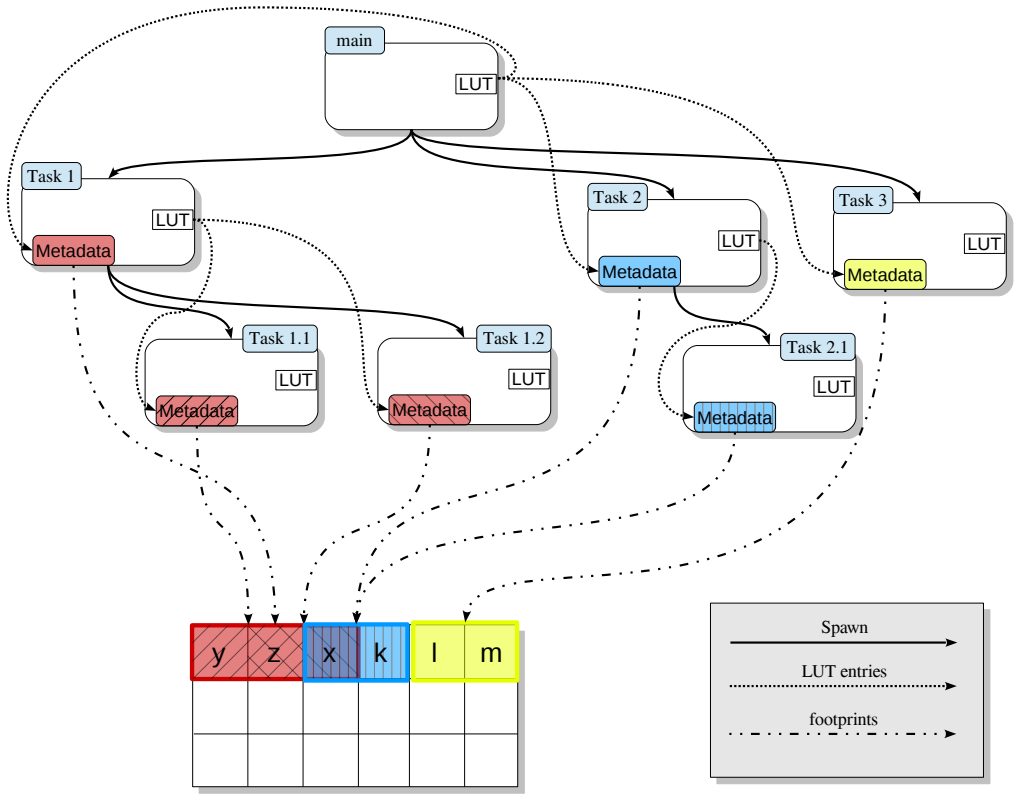


Figure 2.5: Final Task Graph

(i.e., *in*, *out*, or *inout*). For this reason, it requires that every memory segment is associated with the *task descriptor* of the active task accessing it. Since a memory segment can have multiple readers but only a single writer, we keep a *readers list* for each memory segment associating it with its reader-tasks.

Specifically, in case of RAR relationship (line 14 on Figure 2.6) we only append the task in th *readers list* of the corresponding memory segment association. On the other hand, in the case of a WAR relationship (line 17 on Figure 2.6), we first append the new *task descriptor* to the *notify list* of each of the *task descriptors* in the *readers list* of the corresponding memory segment association. Then we erase that association and associate the memory segment with the new *task descriptor*. Since that task depends on the memory segment's readers, we increase its *dependencies counter* by one for each *task descriptor* in the *readers list*. In the case of a WAW relationship (line 28 on Figure 2.6), we first append the new *task descriptor* to the *notify list* of the *task descriptor* associated with the corresponding memory segment. Then we erase the previous association and associate the memory segment with the new *task descriptor*, increasing its *dependencies counter* by one. This technique results in a chain of *task descriptors*, sorted by their *spawn* order, that one notifies the other when they reach completion. These chains reflect the data flow between tasks in the program execution, and force an ordering between the tasks access-

```
1   Create_Dependencies(task_t task){
2
3     for each argument in task.args
4       if argument.type == safe
5         continue; // skip argument
6       else
7         start_blk = argument.start;
8         stop_blk = argument.stop;
9         for each blk from start_blk to stop_blk
10          if slab[blk].metadata == NULL;
11            slab[blk].metadata = task.metadata;
12          else if slab[blk].metadata.type == in
13            // RAR
14            if argument.type == in
15              slab[blk].metadata.reader_lst.add(task);
16            // WAR
17            else if argument.type == out or argument.type == inout
18              for each reader in slab[blk].metadata.reader_lst
19                reader.to_notify.add(task);
20                task.waitfor_cnt++;
21              slab[blk].metadata = task.metadata;
22          else if slab[blk].metadata.type == out or slab[blk].metadata.type == inout
23            // RAW
24            if argument.type == in
25              slab[blk].metadata.owner.to_notify_lst.add(task);
26              task.waitfor_cnt++;
27            // WAW
28            else if argument.type == out or argument.type == inout
29              slab[blk].metadata.owner.to_notify_lst.add(task);
30              task.waitfor_cnt++;
31              slab[blk].metadata = task.metadata;
32    if task.waitfor_cnt == 0
33      schedule(task);
34  }
```

Figure 2.6: Create Dependencies

```
1   Fire_Dependencies(task_t task){
2     for each tsk in task.notify_lst
3       add_and_fetch(tsk.fired_cnt,1);
4       if tsk.waitfor_cnt == tsk.fired_cnt;
5         schedule(tsk);
6   }
```

Figure 2.7: Fire Dependencies

ing common memory segments to eliminate data races and ensure correctness. Whenever a task reaches completion it traverses its *task descriptor's notify list* and increases the *fired dependencies counter* of each of the traversed *task descriptors* by one (line 2 on Figure 2.7). If the *fired dependencies counter* of a *task descriptor* becomes equal to its *dependencies counter*, then the corresponding task can be executed. Any task may depend on more than one task, and thus the *fired dependencies counter* may be modified by many threads. To avoid a race on *fired dependencies counter* of any *task descriptor* we increase this counter by using add-and-fetch primitive, which is supported by the GCC compiler.

The add-and-fetch primitive is implemented with the help of the *xadd* instruction (supported by the x86 architecture), as follows: the increment value is added to the result of *xadd* instruction. In addition, we use *lock free notify list* in any *task descriptor* which we will present later in Section 3.4.2.

2.1 Locality And Concurrency Optimization

Storing all the associations between *task descriptors* and memory segments in a single data structure is expected to significantly decrease concurrency, due to the high contention of accesses to that data structure. In order to minimize the contention, we propose the use of a distinct look-up table (LUT) per task, stored in its *task descriptor*. This LUT holds the associations between memory segments and *task descriptors* of child-tasks—first level descendent-tasks—of the task owning it. This design takes advantage of the hierarchy of nested task parallelism and the fact that we only need to detect and resolve dependencies between sibling-tasks. We create a *task descriptor* for the `main` function to make it the root of the task hierarchy. This *task descriptor* holds the associations between memory segments and the *task descriptors* of tasks directly spawned by the `main` function. Similarly, for every child-task, the association of its *task descriptor* with memory segments is stored in its parent's *task descriptor*.

By employing this hierarchical design when *spawning* a new task, we only need to query the *spawning* task's *task descriptor* for any associations between a memory segment and sibling-tasks' *task descriptors*. Since tasks are atomic, in the adopted programming model the *spawn* of each child task will be executed by its parent task. As a result, the queries for dependencies do not require any synchronization with other tasks. That also holds for updates to the memory segment associations with *task descriptors*, as well as their *readers lists*. This behavior, combined with thread pinning, results in increased spatial and temporal locality, consequently increasing the number of cache hits and thus significantly improving the performance and energy efficiency of the dependence analysis.

This design reduces the contention points to two. The *notify list* and the *fired dependencies counter*. First, a task spawn might race with the completion of the task it depends on. This happens when a task being spawned needs to be inserted in the *notify list* of the task it depends on, whereas the latter tries to empty its *notify list* and notify the tasks waiting for it that it reached completion. Second, tasks reaching completion may race each other in increasing the *fired dependencies counter* of a task that happens to depend on both of them. We handle the latter using the atomic add-and-fetch instruction. Regarding the *notify list* we discuss our proposed solution below in subsection 3.4.2.

2.2 Limitations

```
1    // ...
2    spawn qux(x, z)[out:x, in:z];
3    *z = *x;
4    spawn qux(y, z)[out:y, in:z];
5    // ...
```

Figure 2.8: Task depending on the output of its child

Only detecting dependencies between sibling-tasks imposes a limitation to the expressiveness of the programming model. In Figure 2.8, we give an example where a code segment (line 3) of a task depends on the output of one of its child-tasks (line 2). In such cases, we fail to detect this dependency and require the developer to add this code segment in a child-task, so that the dependence analysis will be able to detect and resolve it. Alternatively, an explicit `sync` directive between the child-task and the corresponding code segment also suffices. Depending on the level of available parallelism in the task one case might be preferred over the other. For instance, if a task spawns many independent child-tasks and this is one of the few existing dependencies, the creation of a new child-task should be preferred.

Chapter 3

The PARTEE: *PARallel Task Execution Engine*

We implement our dynamic dependence analysis in PARTEE, a scalable runtime system supporting nested task parallelism. The Cilk runtime system and scheduler are not straightforward to extend in such a way. First, the Cilk scheduler does not use a clear representation of a task at run-time. Instead, Cilk uses Cactus Stacks [6] for scheduling and job stealing. This representation, however, does not capture the notion of a single task clearly, making the reordering of tasks difficult to satisfy dataflow dependencies. To rectify this, we design PARTEE to use *task descriptors*, as described in chapter 2, which are straightforward to reorder. Second, Cilk programs are not easily composable, i.e., one cannot link libraries written with Cilk with other pure C applications, because Cilk changes the calling convention and requires compiler support to call functions correctly. In contrast, PARTEE is implemented as a library, based on Pthreads, and applications using it can be linked with other applications without any special requirements. PARTEE maintains the same *spawn/sync* abstraction as Cilk. Thus, the developer inserts spawn calls to create tasks and sync calls to synchronize them. To support task dependencies, we extend the spawn primitive with task footprints.

3.1 Task Scheduling

PARTEE inherits the A-steal [4] and Blumofe and Leiserson's [5] *work-stealing* algorithm for scheduling tasks. The runtime system consists of P software threads, each pinned to one of the P available hardware threads (given as a command line argument). We call these software threads virtual processors (VP). Each VP owns a task-queue, in which it puts newly spawned tasks. Each VP can spawn and execute tasks to and from its own task-queue. In case its task-queue is empty it tries to steal work from another VP. Figure 3.1 presents how PARTEE's scheduler works.

To *spawn* a new task, a VP needs to create a new *task descriptor* and perform the dependence analysis on it, as described in chapter 2. For this process we use SCOOP [7], a source-to-source compiler which enables us to use `#pragma` directives for the task anno-

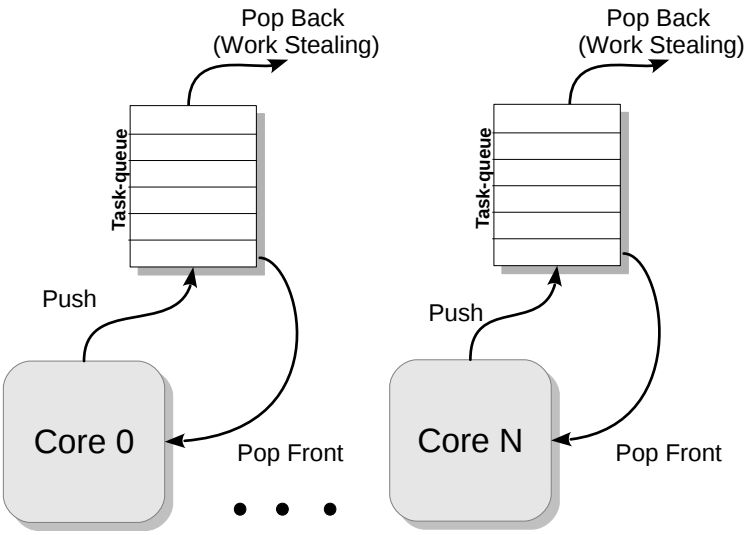


Figure 3.1: PARTEE's Scheduler

tation. SCOOP, using the task annotations, generates code that creates the *task descriptor*, initializes it, and finally passes it to the dependence analysis. Additionally, SCOOP performs a static analysis on the task footprints and is able to exclude arguments that they are safe to omit from the dynamic dependence analysis—they do not create any dependencies. SCOOP also checks if any task can be executed without check if it has dependencies with other tasks. A task can be executed immediately, if we can ensure that (*a*) it has not dependencies, which means that the newly spawned task is the first child of the parent task and (*b*) the task-queue which the worker owns *is full*. If those two conditions are true then, the task is safe to be executed immediately. Figure 3.2 presents how the source code was transformed, after compiling it with SCOOP.

3.2 Task Queue

We implement the *task queue* of every VP as an *array-based*, *single producer-multi consumer double-ended queue* (deque). We use a fixed sized array-based deque to limit the size of the *task queue*. Using a fixed size *task queue*, reduces the spawning of task descriptors'. The creation of a task descriptor increases the execution time and the memory usage. In section 3.1, we describe how we solve this problem using SCOOP.

In section 3.1, we mention that each VP owns a *task queue*, in which it pushes the newly created task descriptors, and pop them one by one in order to execute the tasks. Additionally, if a VP has no tasks to execute, it chooses a random VP, a victim, to *steal* work from. Motivated by this execution model, we design the deque as a *single producer*, since only one VP (the owner) can produce tasks and push them in its task queue, and *multi consumer*, because multiple VPs can consume tasks contained in it. It is important to note that the other VPs pop tasks from a different end of the deque than the owner and

```
1  void foo(int* x, int *y) {
2    y = x...
3  }
4
5  int main(void) {
6    ...
7  #pragma css malloc
8    x = malloc(128*sizeof(int));
9  #pragma css malloc
10   y = malloc(128*sizeof(int));
11
12 #pragma css task in(x[128]) out(y[128])
13   foo(x, y);
14   ...
15 }
```

(a) Source Code

```
1  void foo(int* x, int *y) {
2    y = x...
3  }
4
5  int main(void) {
6    ...
7    tpc_task_descriptor *__cil_tmp18;
8    x = tpc_malloc(128*sizeof(int));
9    y = tpc_malloc(128*sizeof(int));
10
11   if( parent->number_of_children != 0 ||
12       is_Full(task-queues[thread_id]) == 0 ){
13     __cil_tmp18 = tpc_task_descriptor_alloc(8);
14     __cil_tmp18->task = wrapper_SCOOP__foo;
15     __cil_tmp18->args = (tpc_task_argument *)(__cil_tmp18 + 1);
16     __cil_tmp18->args_num = 2;
17     __cil_tmp18->number_of_children = 0;
18     __cil_tmp18->status = 0;
19     __cil_tmp18->parent = parent;
20     (__cil_tmp18->args)->addr_in = (void *)x;
21     (__cil_tmp18->args)->addr_out = (void *)x;
22     (__cil_tmp18->args)->type = IN;
23     (__cil_tmp18->args)->size = (128 * sizeof(float ));
24     (__cil_tmp18->args) ++;
25     (__cil_tmp18->args)->addr_in = (void *)y;
26     (__cil_tmp18->args)->addr_out = (void *)y;
27     (__cil_tmp18->args)->type = OUT;
28     (__cil_tmp18->args)->size = (128 * sizeof(int));
29     (__cil_tmp18->args) ++;
30     __cil_tmp18->args = (tpc_task_argument *)(__cil_tmp18 + 1);
31     tpc_call(__cil_tmp18);
32   }
33   else {
34     foo(x, y);
35   }
36   ...
37 }
38
39 void wrapper_SCOOP__foo( tpc_task_argument *args ) {
40   int *arg1 = (int *)args->addr_in;
41   args++;
42   int *arg2 = (int *)args->addr_out;
43
44   foo( arg1, arg2 );
45 }
```

(b) SCOOP Code

Figure 3.2: SCOOP Transformation

producer VP.

Below, we present two different deque implementations which can be used in PARTEE. These two implementation based on A-steal and Blumofe and Leiserson's work-stealing algorithm and they observe all requirements mentioned above.

3.2.1 Dynamic Circular Work-Stealing Deque

Chase's and Lev's deque algorithm [8] (DCWQ) is considered as the state-of-the-art among double-ended queue implementations. Chase et al. designed a concurrent, single-producer, multi-consumer, array-based double-ended queue. DCWQ algorithm based on Arora's and Bulmofe's (ABP) [9] algorithm. DCWQ differs with ABP mainly in the expansion and shrink of the array and in the way than Chase et al. solve an ABA problem [10] which may be created. We implement the DCWQ algorithm but without the resize of the array.

DCWQ consists of a fixed size *circular* array, a `top` variable which points on the top

end of the array and a `bottom` variable which points on the opposite end of the array. Figure 3.3 illustrates the DCWQ data structure. As the original ABP, DCWQ has three relevant operations, (*i*) *pushBottom*; (*ii*) *PopBottom*; and (*iii*) *steal*, which corresponds to "popTop" in the ABP deque.

```
1 struct DCWQ{
2   data_t entries[MAX_SIZE];
3   int top;
4   int bottom;
5   int size;
6 };
```

Figure 3.3: DCWQ data structure

3.2.1.1 PushBottom

The *PushBottom* operation allows the owner VP to push some newly produced data to the deque. This operation does not need any synchronization, because only the owner VP can execute it. Figure 3.4 presents the pseudo-code of *PushBottom* operation. On line 5, they read the value of `bottom` and in line 6 Chase and Lev read the value of `top`. They compare the difference of these two variables with the `size` of the deque in order to guarantee that the deque is *not full* (line 8). If the deque is full, then the authors return `FALSE`. On the other hand, if the deque is not full, they put the new data in the first available position (line 12), and they increase the `bottom` value by one. Because of the circular-array, they always increase the value of `bottom` and they use the result of module operation of `bottom` and deque size (line 11).

```
1  boolean_t pushBottom( data_t data )
2  {
3    int b, t, i;
4
5    b = Queue->bottom;
6    t = Queue->top;
7
8    if ((b - t) >= DCWQ->size)
9      return FALSE;
10
11   i = b % DCWQ->size;
12   DCWQ->entries[i] = data;
13   DCWQ->bottom = b + 1;
14
15   return TRUE;
16 }
```

Figure 3.4: PushBottom operation

3.2.1.2 PopBottom

The *PopBottom* operation allows the owner VP to pop data from the deque without any synchronization on `bottom`. Figure 3.5 presents the pseudo-compare of *PopBottom* operation. They first decrease the `bottom` by one and store it in a local variable (line 6). Then they compare `bottom` and `top`, in order to check if the deque is empty (line 9). In case of empty deque, they reset the `bottom` (line 10) and return `NULL` (line 11). Otherwise, they get the latest data in deque (line 14) and then they check the distance between `bottom` and `top` (line 16) to guarantee that the value about to be returned is not the last element in the deque, thus it can be returned safely (line 17). Otherwise, a race may occur between the owner VP and other VPs, which try to get the last element in the deque. In case of `bottom` and `top` equality, there is only one element in the deque. In this case, they use *compare and swap* (CAS) on the `top` (line 20) and the VP that made a successful CAS will get the last element in deque. Additionally, they update `bottom` with the correct value (line 25).

```
1  data_t popBottom ()
2  {
3    data_t *ret_val = NULL;
4    int t, b;
5
6    b = --Queue->bottom;
7    t = Queue->top;
8
9    if (b < t) {
10     Queue->bottom = t;
11     return NULL;
12   }
13
14   ret_val = DCWQ->entries[ b % DCWQ->size ];
15
16   if (b > t) {
17     return ret_val;
18   }
19
20   if (!compare_and_swap(&DCWQ->top, t, t + 1)) {
21     ret_val = NULL;
22   }
23
24   Queue->bottom = t + 1;
25
26   return ret_val;
27 }
```

Figure 3.5: PopBottom operation

3.2.1.3 Steal

The *Steal* operation allows the other VPs to "steal" data of a deque that does not belong to them. Figure 3.6 presents the pseudo-code of *Steal* operation. As mentioned previously, the VPs that try to steal data from other's deques, they use the opposite end (`top`) than

the owner VPs (bottom). As a result, *Steal* operations requires synchronization on top, which is guaranteed with a CAS operation. Specifically, they first compare top and bottom, in order to check if the deque is empty (line 9). If the deque is empty, they return NULL. Otherwise, the element in which top index points to is stored in the variable ret_val (line 12).The VP which performed a successful in changing the top, returns the element (line 15), whereas the other VPs return NULL (line 18).

```
1  data_t steal ()
2  {
3    int t, b;
4    data_t ret_val = NULL;
5
6    t = Queue->top;
7    b = Queue->bottom;
8
9    if (b <= t)
10     return NULL;
11
12   ret_val = DCWQ->entries[ t % DCWQ->size ];
13
14   if (compare_and_swap(&DCWQ->top, t, t + 1)) {
15     return ret_val;
16   }
17
18   return NULL;
19 }
```

Figure 3.6: Steal operation

It is worth mentioning that, when there is only one element in the deque, they synchronize the VPs on the top variable both in *PopBottom* and in *Steal* operation. This is because, they want to avoid any race condition between VPs.

3.2.2 Fast Concurrent Double-Ended Queue

Another deque implementation with similar semantics as DCWQ is the FCDQ. Fast Concurrent Double-Ended Queue (FCDQ) is a concurrent, single-producer, multi-consumer, array-based, double-ended queue designed and implemented by us. It is concurrent in the sense that many threads may perform concurrent operations on the deque. However, FCDQ is single-producer, which means that only one thread can produce (enqueue) elements; while many threads can consume (dequeue) elements.

FCDQ has three basic mutating operations, similarly to the ABP deque: (*i*) *Enqueue*, which corresponds to "pushBottom" in the ABP deque; (*ii*) *Dequeue Front*, which corresponds to "popBottom"; and (*iii*) *Dequeue Back*, which corresponds to "popTop". Additionally, it supports two read-only operations, *isEmpty* and *isFull*, which return true if the deque is empty or full respectively.

FCDQ is inspired by Fastflow's [11] multi-producer, multi-consumer, array-based, queue algorithm. Fastflow is a standard queue, to which we add the new *Dequeue Front* operation. We also modify the implementation of the two existing operations, *Enqueue*

and *Dequeue Back*, to take advantage of the single-producer semantics of work-stealing, and to ensure race-freedom.

FCDQ consists of a circular, constant-sized array, a head and a tail index, and a length counter. Figure 3.7 illustrates the FCDQ data structure. We use the constant-sized array as the memory pool for the deque nodes. We use the head and tail indexes to keep track of the deque's position in the array. Initially, both head and tail are set to zero. We use the length counter to keep track of the nodes stored in FCDQ.

```
1 struct FCDQ{
2   data_t entries[MAX_SIZE];
3   int head;
4   int tail;
5   int total_entries;
6 };
```

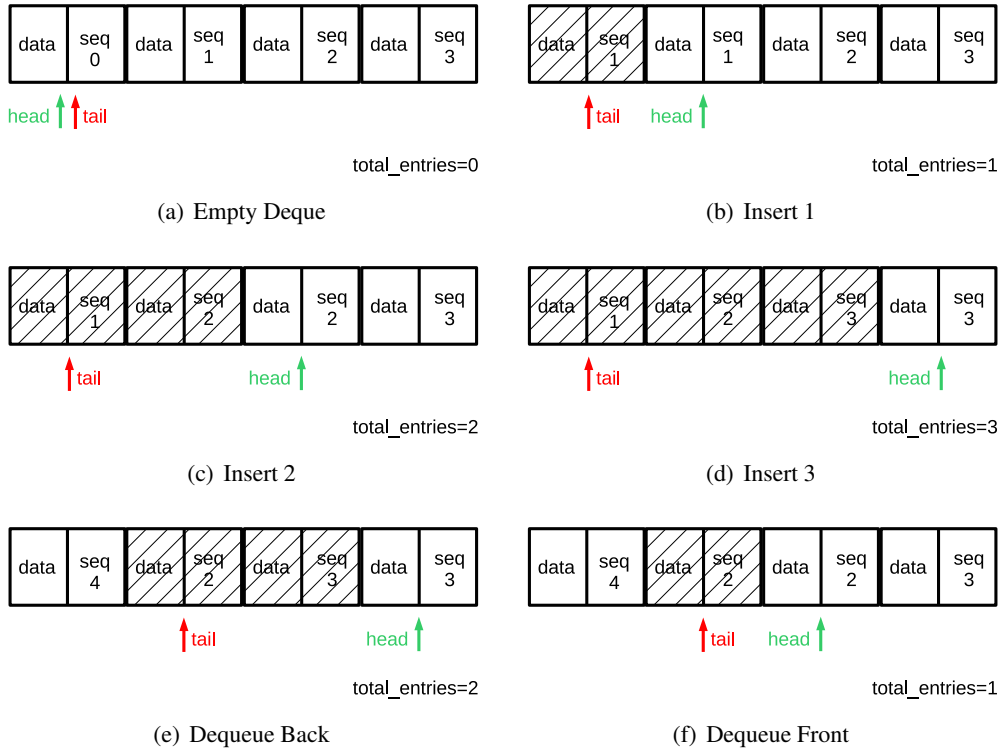
Figure 3.7: FCDQ data structure

Each node of FCDQ, in addition to its data, features a sequence number. Similarly to the Fastflow Queue, the role of the sequence number is to determine where the next operation will take place. Initially, all the entries of the array have a sequence number equal to their index in the array.

In FCDQ, we use the length counter and the tail index as the only synchronization points. Our algorithm ensures that the distance between the head and the tail pointer is the same as the value of the length counter after every operation.

Figure 3.8 presents an example demonstrating how the deque state changes as deque operations are performed. The length counter is represented by variable `total_entries`. Figure 3.8(a) shows the state of an empty deque. The head and tail pointers point to the same place and the `total_entries` counter is zero. As noted above, FCDQ is a *Single-Producer* deque. This means that only one dedicated thread, the *Enqueuer Thread*, can execute Enqueue. By executing the first Enqueue operation, the Enqueuer Thread modifies the deque state as follows: the head pointer moves to the next position, the sequence number of the first position of the array changes to one and the `total_entries` counter increases by one as shown in Figure 3.8(b). Figures 3.8(c) and 3.8(d) show similar modifications to the deque by two additional Enqueue operations. Notice that these three Enqueue operations are never executed concurrently, because only the Enqueuer Thread can execute Enqueue operations. This assumption, motivated by the A-Steal algorithm, allows for a more efficient implementation of the deque operations.

In contrast, operations *Dequeue Front* and *Dequeue Back* can be executed in parallel, because multiple threads can execute these operations. The Dequeue Back operation modifies the tail pointer and changes the sequence number of the element to which it pointed. It also reduces the `total_entries` counter by one (Figure 3.8(e)). On the other hand, Dequeue Front modifies the head pointer and moves it to the previous array position. It also changes the sequence number of the previous position. Finally, it reduces the `total_entries` counter by one (Figure 3.8(f)). To properly synchronize these operations and ensure their linearizability as efficiently as possible, we use atomic



(a) Empty Deque (b) Insert 1

(c) Insert 2 (d) Insert 3

(e) Dequeue Back (f) Dequeue Front

Figure 3.8: Working Example

instructions, as presented in Section 3.2.2.4.

3.2.2.1 Enqueue

The *Enqueue* operation is a *blocking* operation to guarantee the correctness of the algorithm. This is made necessary by the semantics of the deque; an Enqueue operation should succeed to put a new element in the deque even when a concurrent Dequeue operation failsi to remove an element of it. The pseudo-code of this operation is presented in Figure 3.9.

As mentioned above, the Enqueue operation modifies the head pointer and increases the value of the `total_entries` counter. The head pointer shows the first empty position in the array, which is the next place where the enqueuer thread can store data.

Figure 3.9 shows pseudo-code for the Enqueue operation. The aim of this operation is to iterate until it manages to increase the head pointer (lines 9–17). To do so, it stores the current value of the head pointer into local variable `head` (line 9). Then, it stores into the local variable `node` a pointer to the node found in that position (line 10). Because of the circular array, we increase the head pointer (or reduce it) and we use the result of modulo operation of head and deque's size. This pointer is used in order to access and store the sequence number of the node (line 12). If the value of the head pointer that was

```
1  boolean_t enqueue(data_t data)
2  {
3    unsigned int head, seq;
4    queue_node_t *node;
5
6    if(isFull(FCDQ))
7      return FALSE;
8
9    head = FCDQ->head;
10   node = &FCDQ->array[head % FCDQ->total_entries];
11   do {
12     seq = node->seq;
13     if(head < seq)
14       return FALSE;
15   } while(head > seq);
16
17   FCDQ->head++;
18   node->data = data;
19   node->seq++;
20   atomic_incr(FCDQ->total_entries);
21
22   return TRUE;
23 }
```

Figure 3.9: Enqueue operation

read is equal to the sequence number (line 13) then the Enqueue operation returns `FALSE` (line 14). Otherwise, the Enqueue operation will continue to loop until the value of the head pointer becomes equal to the sequence number (line 16). There are two scenarios than can produce this failure situation. The first one is that the deque is *full*. In that case, the head pointer would point to the same position as the first enqueue done, which is the same position where the `tail` pointer points. Figure 3.10 presents a snapshot of a full deque with four elements. Note that all entries are filled and head and tail pointers, point at the same entry in the array.

Figure 3.10: Full Deque

For the second scenario, assume that a Dequeuer Thread (`FDT`) has reduced the `total_entries` counter and somehow fails to make progress (e.g., crashed or was context-switched forever). If that failure happens before the `FDT` thread modified the sequence number then the Enqueue Operation will not complete either. This happens because the Enqueuer Thread waits until the end of the Dequeue operation that precedes. Hence, the other Dequeuer Threads will dequeue all the included elements and they will find that the deque is empty, because the last Enqueue operation has not completed yet,

and they have consumed all the other elements.

This scenario is an artifact of the deque semantics, as we have to ensure the correctness of the algorithm, and we cannot predict if the blocked Dequeuer Thread will complete the operation or not. It would be a correctness and invariant violation to skip the "stuck" Dequeue operation. Due to this, the algorithm of the Enqueue operation is *blocking*.

3.2.2.2 Dequeue Front

```
1  data_t dequeue_front()
2  {
3    unsigned int head, seq;
4    queue_node_t *node;
5
6    if(!atomic_add_unless(FCDQ->total_entries, -1, 0))
7      return NULL;
8
9    head = FCDQ->head;
10   node = &FCDQ->entries[(head-1) % FCDQ->total_entries];
11   do {
12     seq  = node->seq;
13     if(seq < head)
14       return NULL;
15   } while(seq > head);
16   FCDQ->head--;
17   node->seq--;
18
19   return node->data;
20 }
```

Figure 3.11: Dequeue Front operation

The *Dequeue Front* operation allows the Enqueuer thread to get the data which were recently put in the deque (LIFO operation). Again, we assume that only the Enqueuer thread will perform Dequeue Front operations. Of course, other threads may invoke Dequeue Bask operation concurrently with a Dequeue Front operation.

Figure 3.11 shows the pseudo-code for the Dequeue Front operation. Initially, Dequeue Front checks if there are available elements and reserves the element to be dequeued by atomically reducing the counter (lines 6–7). Then, similarly to Enqueue Front, it uses a loop to select the item to be dequeued (lines 9–17). The dequeued node position corresponds to the value of head reduced by one, as head always points to the first *empty* place in the array. As with Enqueue Front, Dequeue Front stores local copies of the head pointer, the item to be dequeued and its sequence number (lines 9–12). Then, it calculates the difference between seq and head variables (line 13); if the head is greater tan seq, the operation returns NULL (line 14). Otherwise it loops until head becomes equal to seq (line 15). If the operation is successful, no other thread has dequeued that element and if the difference between head and the sequence number of the node is negative, it means that another thread has raced to dequeue the data of this node and the deque is now empty.

3.2.2.3 Dequeue Back

```
1  data_t dequeue_back()
2  {
3    unsigned int tail, seq;
4    queue_node_t *node;
5
6    if(!atomic_add_unless(&FCDQ->total_entries, -1, 0))
7      return NULL;
8
9    do {
10     tail = FCDQ->tail;
11     node = &FCDQ->array[tail % FCDQ->total_entries];
12     seq  = node->seq;
13     if(seq == (tail+1)) {
14       if(compare_and_swap(&FCDQ->tail, tail, tail + 1) == tail)
15         break;
16       // ... Backoff code
17     }
18     else if(seq < (tail+1))
19       return NULL;
20   } while(1);
21   node->seq = tail + FCDQ->total_entries + 1;
22
23   return node->data;
24 }
```

Figure 3.12: Dequeue Back operation

The *Dequeue Back* operation allows any thread to get data from the rear side of the deque (i.e., the oldest data in the deque), using it in a FIFO order. Dequeue Back can be performed by multiple threads concurrently with any Dequeue Front or Enqueue operations. The pseudo-code of this operation is shown in Figure 3.12.

Similarly to Dequeue Front, this operation also ensures that there is an element to dequeue using the `total_entries` counter (lines 6–7) and then iterates attempting to dequeue the required data (lines 9–20). Dequeue Back uses a *compare-and-swap* (CAS) instruction in order to change the value of tail. The CAS instruction is needed in this case because Dequeue Back can be invoked by multiple threads. Their accesses to the rear end of the deque must therefore be synchronized, so that they obtain valid and not stale data. The CAS instruction also guarantees the correctness of the operation, as discussed in Section 3.2.2.4.

3.2.2.4 Atomic instructions

We use several atomic primitives to design and implement the three deque operations (Enqueue, Dequeue Front and Dequeue Back), to avoid the overhead of locking the deque for every operation. These atomic instructions are:

atomic_increase(var): This instruction increases the value of a variable *var* atomically, i.e., without anyone being able to interrupt the increment.

atomic_decrease(var): As in atomic_increase instruction, in this instruction the value of a variable *var* is decremented atomically.

compare_and_swap(var, oldval, newval): This instruction changes the value of a variable var to *newval* only if the value of *var* is equal with *oldval*.

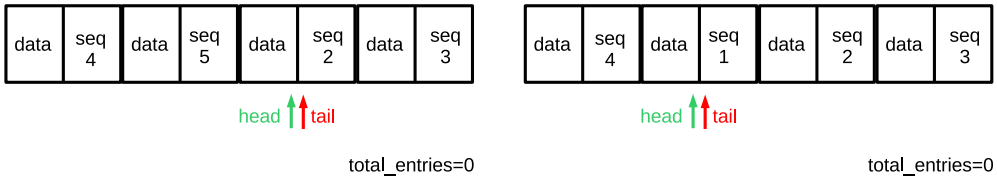
atomic_add_unless(var, add, unless): This is an easily implementable synthetic instruction used commonly in Linux Kernel. It adds the **add** value to the variable **var**, if the value of **var** is not equal to **unless**. Figure 3.13 presents pseudo-code for this synchronization primitive. As shown in line 6, it uses a re-trying loop of compare and swap instructions to ensure atomicity.

```
1  int atomic_add_unless(int *var, int add, int unless)
2  {
3    int cas = *var;
4    int old = unless;
5
6    while(cas != unless && cas != old) {
7      old = cas;
8      cas = compare_and_swap(var, old, old + add);
9    }
10
11   return cas;
12 }
```

Figure 3.13: Atomic Add Unless

The reason for using this synthetic instruction is the *Both Dequeues-One Element* problem. This problem appears when two different Dequeue operations (Front and Back) take place concurrently and there is only one element in the deque. There can be a race condition: both threads trying to get elements from opposite sides of the queue, could get this last element, if they both observe that the deque is not empty. As only one thread can get the element, the other thread blocks. To avoid this, we use this synthetic instruction, which allows us to *reserve* the element before removing it from the deque, by reducing (adding -1) the total element counter of the deque. This reservation means that each Dequeuer Thread should reduce the `total_entries` counter by one and if this reduce is successful, then it can dequeue the element as described in Dequeue Front (Section 3.2.2.2) and Dequeue Back (Section 3.2.2.3). Specifically, we add -1 to `total_entries` variable only if the `total_entries` counter is not zero. The Dequeue Front operation (Figure 3.11, line 8) uses atomic-add-unless to perform this before proceeding with the operation, as does Dequeue Back (Figure 3.12, line 8).

Figure 3.8(f) shows a snapshot of the deque after a Dequeue Front operation. Figure 3.14 shows the two possible valid states after two different dequeue operations (front and back) where executed in a deque with one element. To ensure one of those two states of deque, we implement the atomic_add_unless instruction to reduce `total_entries` counter and reserve the element before dequeuing. Without atomic_add_unless instruction, the `total_entries` counter may reach a negative value, in case independent



(a) Dequeue Back (b) Dequeue Front

Figure 3.14: Possible Operations

Dequeue Front and Dequeue Back operations access it in parallel. This would compromise the correctness of the algorithm. By using the atomic_add_unless instruction, we ensure that only one thread (operation) can reduce the `total_entries` counter and thus our deque is safe and race-free.

3.2.2.5 Differences with Fastflow

As already mentioned above, PARTEE is based on Fastflow's multi-producer, multi-consumer queue, to which we have added a new operation (Dequeue Front) and modified the existing operations (Enqueue and Dequeue). The new operation added allows the enqueuer thread to dequeue elements from the front side using the data structure as a stack. This operation is necessary for the A-steal scheduling algorithm.

Compared to Fastflow, PARTEE is a single-producer deque, which means that only one thread can enqueue new elements into the deque. Fastflow's queue Enqueue supports multiple producer threads. To take advantage of the semantic difference, we modify the Enqueue operation by replacing the synchronization primitives with simple increases, as mentioned in Figure 3.9. We also add the total_entries counter which is accessible by all operations, using the atomic instructions described above.

Finally, we modified the Dequeue operation 3.2.2.3 to resolve the *Both Dequeues-One Element* problem, which described in Section 3.2.2.4. We add additional (but efficient) synchronization (lines 8–9 in Figure 3.12), which reduces the `total_entries` counter if it is not zero and thus we can guarantee the consistency of our algorithm.

3.2.3 DCWQ vs FDCQ

We implement a micro-benchmark to evaluate and compare DCWQ and FCDQ. We evaluate these two different deque algorithms by measuring the number of of total operations per second. This micro-benchmark consists of one deque of 16 elements, in which 128,000 Enqueue (PushBottom) operations and 128,000 Dequeue Front (PopBottom) and Dequeue Back (Steal) operations take place. The Dequeue operations are divided in the number of threads. Specifically, the VP which owns the deque, performs the 128,000 Enqueue operations and the $128,000/total\_number\_of\_VPs$ Dequeue Front operations. The rest of the VPs perform $128,000/total\_number\_of\_VPs$ Dequeue Back operations.

We measure DCWQ and FCDQ using this micro-benchmark. We measure the execution time of the micro-benchmark running on 2, 4, 8, 16, 32 and 64 cores ten times and then we divide the geometric mean of execution time with total operations (256,000) to get out the throughput of each deque algorithm. We also measure and compare a lock-based deque implementation using *spin locks*.

# VPs / Deque	2	4	8	16	32	64
DCWQ	10.2820	3.26159	1.61151	0.860167	0.380261	0.16550
FCDQ	7.57354	1.61107	0.92538	0.481306	0.218042	0.12883
Spin Lock	5.23662	0.94835	0.36819	0.108802	0.024245	0.00406

Table 3.1: Throughput (Mops/s)

Table 3.1 includes the experimental results of the micro-benchmark described above. We can see that DCWQ has the highest throughput in execution with 2, 4, 8, 16, 32 or 64 VPs. Spin Lock deque has the lowest throughput because of *locking* and *unlocking* of the deque before and after every operation. On the other hand, FCDQ suffers from the *centralized* variable `total_entries`, which is increased or decreased in every operation. This means that all VPs should synchronize in `total_entries` and this may increase the execution time. Unlike, in DCWQ, VPs need to synchronize only in case of one included element in the deque. This does not increase the execution time as the FCDQ's synchronization. Figure 3.15 presents the result of this micro-benchmark. Considering the results in Table 3.1 and Figure 3.15, we decide to use Chase's and Lev's (DCWQ) deque algorithm in PARTEE, since it performs better.

3.3 Synchronization

To implement the *sync* directive we extend the *task descriptor* with a counter holding the number of children of the corresponding task. Whenever a task spawns a child it also atomically increases this counter and similarly whenever a child-task reaches completion it atomically decreases its parent's counter. When *sync* is invoked the executing VP waits for the current tasks's number of children counter to become zero—all of the task's children to be completed. To avoid spinning or idling, VPs waiting at a *sync* execute tasks from their task-queue or even try to steal if the latter is empty. We also place an *implicit sync* before the end of any task, to ensure that all its children reached completion.

Note that, the *sync* primitive is implemented for compatibility with the Cilk programming model and to override limitations like those described in section 2.2. For the correct synchronization of the application, PARTEE relies on its dynamic dependence analysis mechanism.

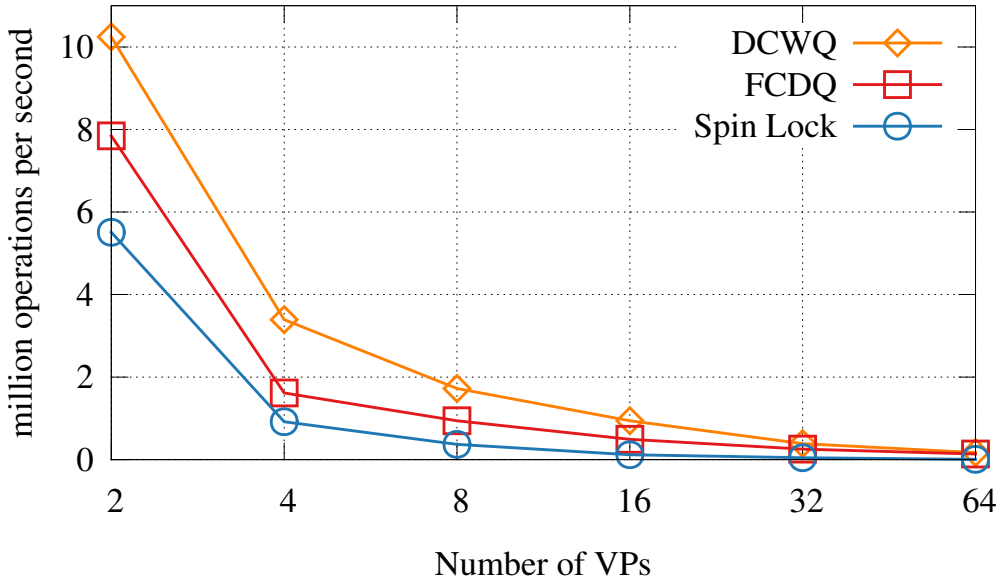


Figure 3.15: Throughput

3.4 Dynamic Dependence Analysis

The main challenges in the implementation of the dynamic dependence analysis were: (*a*) the memory allocation for runtime purposes; (*b*) the implementation of the *notify list*; and (*c*) the implementation of the look-up table (LUT); (*d*) the memory allocation for application purposes.

3.4.1 Region-Based Allocation

As we discussed in chapter 2, our dependence analysis relies on *task descriptors*. To manage the memory required to store them, we use a region-based, parallel, allocator. Our allocator is based on that of Gay and Aiken's [12]. A region is a collection of allocated objects that can be efficiently de-allocated at once. A region-based allocator, contrary to slab-based memory allocators allows for fast, *bulk* de-allocations. Thus, the runtime is able to free a region with all its subregions very efficiently and keep the freed memory in a pool for future use. In PARTEE, each *task descriptor* owns a region. In this region, it allocates all the *task descriptors* of its children and all the associations created by these tasks. That is, whenever a task reaches completion, PARTEE can safely and efficiently free all the memory allocated for its needs. Additionally, by allocating all the memory related to a *task descriptor* in a single region, results in increased memory locality, since the memory within a region is contiguous.

Since PARTEE is a parallel runtime, we need to be able to allocate and de-allocate regions concurrently. Gay and Aiken's allocator, however, is not designed for multi-threading. In their allocator, they hold two lists of free pages which are used for allocation

and de-allocation. The first list contains pages of fixed size, while the other one contains pages with variable sizes. Protecting these lists with locks results in high contention due to the multiple VPs trying to allocate memory as new tasks are being concurrently spawned. As a result, we make the free lists distributed. We essentially create n free lists, each protected by a lock. Whenever a VP needs to allocate or free a region it randomly peaks one of the n lists and tries to lock it. On successful lock it proceeds with the allocation or de-allocation. On failure, it keeps randomly peaking one of the n lists until succeeding. The value of n depends on the number of available VPs. Experimentally, we find that a value of 16 is sufficient to handle 64 VPs. A higher value of n does not impact performance but might affect the total size of memory used by the region-based allocator, depending on the application.

Gay and Aiken's allocator also creates a tree of regions ensuring that all the regions will be de-allocated before the termination of any program. This tree is also used to perform queries about the region that a memory address maps to. To achieve this, Gay et al. create a global region used as the root and add new regions as its children. Maintaining this root region up to date, however, also requires some kind of mutex exclusion. Since PARTEE does not need to query the region-based allocator about the region of an address, we drop this feature and create all the regions at top level. Regarding de-allocation before program termination, since we free a task's region at completion, it is guaranteed by the tree-like task graph that all allocated regions will be eventually freed.

3.4.2 Lock-free Notify List

We implement the *notify list* as a *multi producer-single consumer* stack. Many VPs can *push* data in this stack but only one VP can consume these data. We synchronize the producer VPs by using the *compare and swap* primitive on the *top* variable of the stack. It is worth to note that there are no concurrent *push* and *pop* operations.

As we discuss in section 2.1, we need a way to protect the *notify list* when the runtime tries to insert a task to the *notify list* of a task currently emptying its *notify list* and notifying the tasks depending on it. To achieve this without locks, we use a special node that we baptize *lock*. If the head of the list points to that node, then insertions to that list will fail, meaning that the task owning it reached completion. To ensure that the head of the list is atomically updated we use the compare-and-swap instruction to insert elements to the head of the list, as well as, to lock it.

3.4.3 Look-up Tables

As we discuss in chapter 2, the dependence analysis relies on looking-up associations between memory segments and *task descriptors*. To keep the overhead low we implement LUTs as two-level array-based *tries*. A *trie* [13] is an ordered *tree* data structure that is used to store a dynamic set or an associative array where the keys are usually strings. In our case we use the memory address of each argument as the key. To perform a look-up, we mask the memory address and use its x most significant bits to index the first level of the trie. We then use the following y bits to index the second level. Tuning x and

y values allows us to configure PARTEE to detect dependencies on different granularity. For instance, assume an 1 Gb contiguous memory divided in 524,288 (2^{19}) blocks of 2 Kb each. By setting x to 10 and y to 9, enables PARTEE to operate on block size granularity. That is, if two tasks access the same block, then PARTEE will detect a dependency. Finer granularity, results in increased run-time overhead, whereas coarser granularity may result in false dependencies, and thus reduced exposed parallelism. Figure 3.16 presents a LUT as described above.

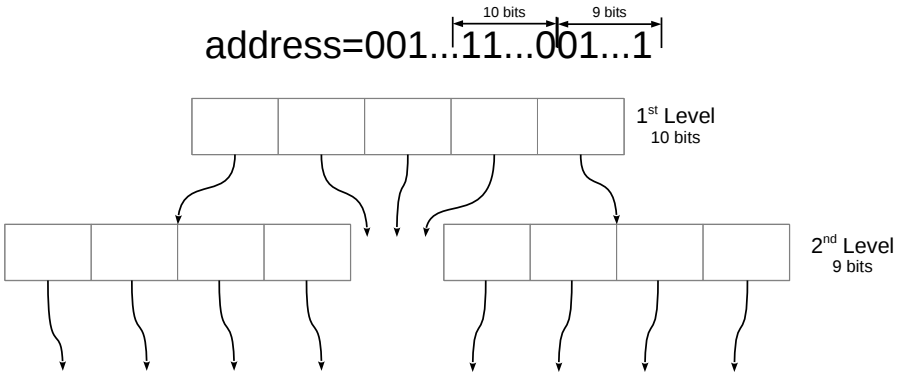


Figure 3.16: Lookup Table

3.4.4 Block-based Allocator

In subsection 3.4.3 is noted that every memory address is used in a look-up table as a key to map addresses to blocks. But it is difficult to map the whole system memory since we need contiguous memory segments that would make the size of the look-up table enormous. To solve this problem, we design and implement a *Block-based Allocator*, which allocates an amount of memory divided in blocks. So far, we can limit the memory usage of each application. This allocator is responsible for every allocation which takes place in the application and is related with tasks. This allocator totally manages 1 Gb of memory dived in block of 2 Kb.

Chapter 4

Evaluation

We evaluate PARTEE on a 4-chip NUMA system with 16 cores per chip, totaling 64 AMD Opteron Processor 6272 cores, with 256 GB RAM. We use the GCC 4.4.8 compiler and -O3 optimization level. We evaluate PARTEE using six benchmarks, Black-Scholes, Cholesky, Head Diffusion, Lu Decomposition, Matrix Multiply and Mergesort and compare it with Cilk and BDDT where possible. We run each benchmark 10 times on varying number of cores, from 1 to 64, doubling the number of cores at each step. We then estimate the geometric mean of the execution time and calculate the speedup over the geometric mean of the sequential executions.

Table 4.1 presents the execution time, calculated as the geometric mean of ten different executions. The execution time for one core is the sequential version of each benchmark and it is the same for all runtimes. Figure 4.1 presents these results on speedup graphs. On the y-axis we plot the speedup and on the x-axis the number of utilized cores. Both axis are in logarithmic scale and a gray line crossing the plot shows the linear speedup. We use green triangles to mark execution times of PARTEE; red squares to mark execution times of PARTEE with explicit *sync* directives and the dynamic dependence analysis disabled (PARTEE ND), to demonstrate the latter's impact on the execution time; blue circles to mark execution times of Cilk; and orange rhombuses to mark execution times of BDDT (where available).

Black-Scholes is an embarrassingly, non recursively parallel benchmark from the PARSEC benchmark suite [14]. The input size is $30,000,000$ elements divided in blocks of 128 elements. We use this benchmark to show the performance and scalability of PARTEE on benchmarks without dependencies. Figure 4.1(a) shows that PARTEE's performance is comparable to that of Cilk and better than that of BDDT.

Cholesky decomposition operates on matrices of $2048x2048$ elements divided in blocks of $256x256$ elements and is commonly used to solve systems of linear equations. The implementation of this benchmark comes from the BDDT's benchmark suite and is not recursively parallel. Figure 4.1(b) shows that PARTEE without the dynamic dependence analysis performs similar to Cilk. With the dynamic dependence analysis enabled PARTEE is able to expose more parallelism and improve performance. However, BDDT still outperforms PARTEE. We attribute this behavior to the fact that our dependence

Benchmark	Runtime	Cores						
		1	2	4	8	16	32	64
Black-Scholes	**PARTEE**	30668.18	15845.88	8014.42	4016.98	2016.15	1200.57	679.4
	PARTEE ND		153836.72	7989.67	4041.93	2015.75	1226.85	680.12
	Cilk		17522.67	8801.375	4380.245	2188.803	1094.211	640.479
	BDDT		11632.8	5874.58	2986.79	1555.26	1200	1187.59
Cholesky	**PARTEE**	32341.395	17045.78	9283.67	5199.49	5088.29	6797.75	7301.02
	PARTEE ND		19091.41	12385.99	10241.63	10644.17	11654.28	12454.25
	Cilk		21206.433	13229.420	9908.162	9176.782	11328.933	13691.428
	BDDT		18051.138	9571.822	5581.995	3488.009	2929.618	2145.858
Heat Diffusion	**PARTEE**	1530.611	833.34	474.73	606.27	473.49	455.08	476.81
	PARTEE ND		757.88	447.67	593.82	459.31	443.04	465.05
	Cilk		1813 .867	925.304	494.959	279.002	195.615	188.693
Lu Decomposition	**PARTEE**	3181.1904	3837.91	2440.74	1693.63	1527.09	1625.53	2216.44
	PARTEE ND		2873.13	1731.92	1072.5	742.19	645.46	652.16
	Cilk		14454.513	7246.473	3687.192	2145.206	1238.177	1141.508
Matrix Multiply	**PARTEE**	38097.5	17862.19	9128.3	4675.66	2526.13	1696.14	1978.75
	PARTEE ND		18475.35	9404.35	4853.79	2814.75	2739.33	2833.41
	Cilk		47695.213	24957.769	12604.134	7483.262	3875.863	4298.061
Mergsort	**PARTEE**	133.195	99.87	67.030	54.27	46.85	47.55	64.82
	PARTEE ND		97.2	66.75	53.18	48.27	45.93	56.42
	Cilk		143 .066	72.902	39.677	25.024	21.590	26.195

Table 4.1: Execution Time (ms) for all runtime systems

analysis design is tailored after recursively parallel programs and not after non-recursive benchmarks, like this implementation of cholesky.

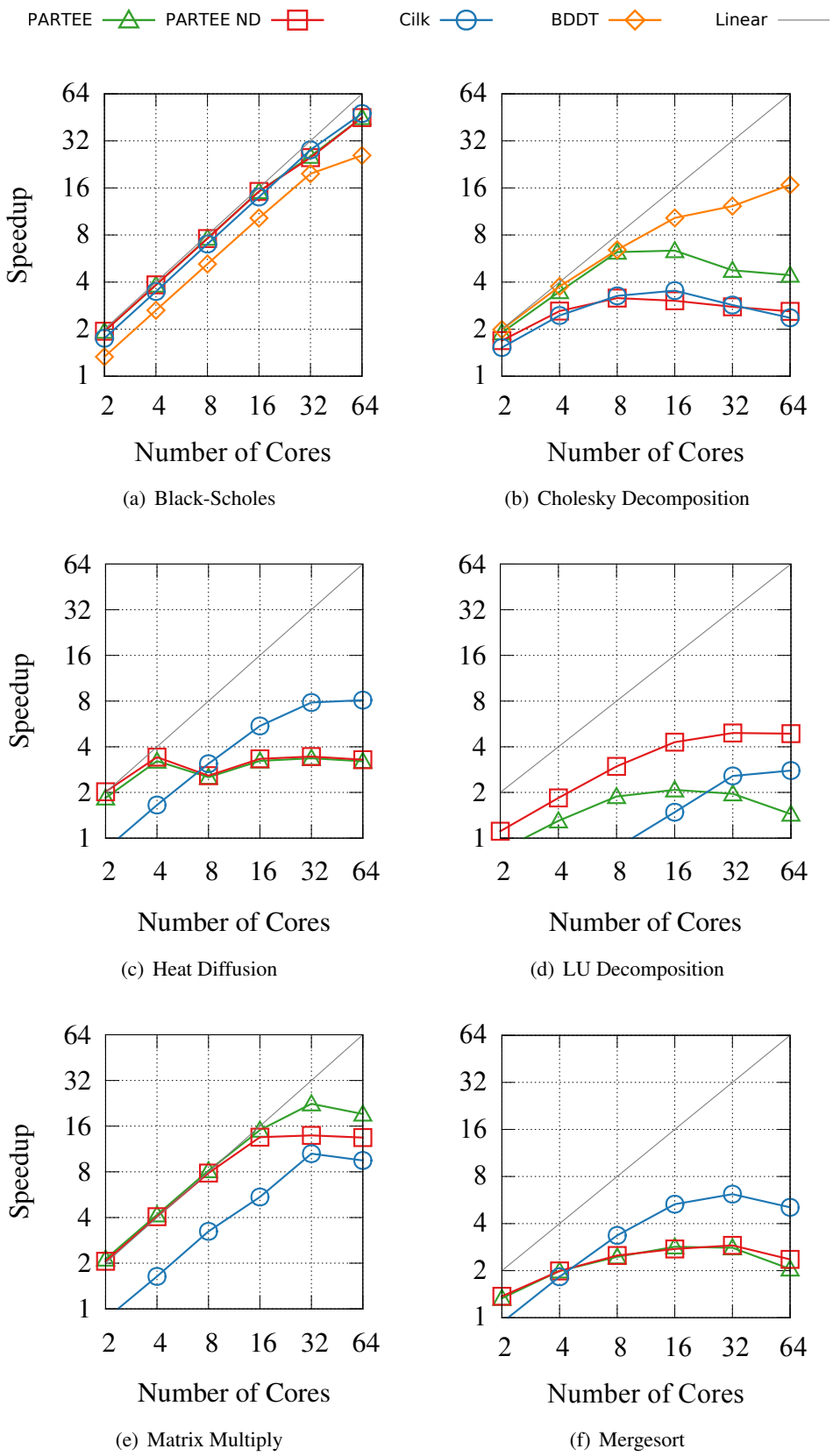
The rest four benchmarks are examples from the Cilk distribution and are all recursively parallel. As a result, we cannot compare BDDT's performance on them.

Heat Diffusion performs some stencil computation multiple times, to calculate temperature distribution in a 2D area ($2048x2048$ elements). Figure 4.1(c) shows that both versions of PARTEE fail to outperform Cilk after 4 cores. This benchmark uses two matrices, the first one as input and the second one as output. At each step, the two arrays are swapped and the stencil computation is performed again. The stencil computation in each step is embarrassingly parallel, so a *sync* directive at the end of each step suffices. PARTEE task creation overhead in this case appears to dominate and decrease the overall performance. This is possible when task arguments consist of many blocks and result in the creation of a large number of entries in the LUTs. For the evaluation of PARTEE we chose a fixed block size of 2 kilobytes, which in this case is much smaller than the minimum argument size, increasing the task creation overhead.

LU, similarly to Cholesky, also operates on matrices of $4096x4096$ elements to solve systems of linear equations. Figure 4.1(d) shows that PARTEE with the dynamic dependence analysis disabled outperforms PARTEE. This is an indication that LU has no interleaved dependencies and can be efficiently expressed using *sync* directives. We also observe that Cilk performs worse than PARTEE with the dynamic dependence analysis disabled, probably due to increased overhead in task spawning.

Matrix Multiply takes two matrices of $2048x2048$ elements as input and writes their product in a third one of the same size. Figure 4.1(e) shows that both versions of PARTEE outperform Cilk. In contrary to LU, Matrix Multiply spawns only a single type of task with a few arguments allowing the SCOOP compiler to optimize task-generation and reduce the overhead. Additionally, we observe that PARTEE is able to extract more parallelism after 16 cores than itself counterpart with the dynamic dependence analysis disabled.

Mergesort is a recursively parallel implementation of mergesort. Figure 4.1(f) shows that Cilk outperforms both versions of PARTEE in mergesort. The implementation, follows the map-reduce principle. First binary splits the array of $1,048,576$ elements and sorts the segments. Then starts merging the sorted segments producing larger sorted segments, until the whole array is sorted. By design this implementation can be sufficiently expressed using the *sync* directive. However, PARTEE fails to reach Cilk's performance even with the dynamic dependence analysis disabled, like in heat diffusion.



(a) Black-Scholes

(b) Cholesky Decomposition

(c) Heat Diffusion

(d) LU Decomposition

(e) Matrix Multiply

(f) Mergesort

Figure 4.1: Speedup Over Sequential

Chapter 5

Related Work

In this chapter, we present some of the research that has been carried out to date and is related to our work. Since we present a task-based runtime system, a deque implementation and a memory allocator, we provide research papers on all topics.

5.1 Task Parallelism

Task parallelism focuses on distributing execution processes (threads) across different parallel computing nodes in order to for the program to utilize better the nodes and their components and provides higher performance. Many different models have been proposed, where the programmers need to explicitly specify which block of code want to be executed in parallel.

Cilk [1] is a multi-threaded language accompanied by a task-parallel runtime system. Cilk provides a clean and simple way to express parallelism through the *spawn* and *sync* directives. Cilk supports nested spawning of tasks but requires explicit synchronization through the *sync* directive. PARTEE combines nested parallelism with a dynamic dependence analysis algorithm to automatically detect and resolve dependencies between tasks.

BDDT [3] is a task-parallel runtime system that employs a block-based dynamic dependence analysis to dynamically discover and resolve dependencies between tasks. BDDT, however, suffers from the single master problem, in which, only one thread creates tasks and thus it does not support nested parallelism. PARTEE employs a similar approach to BDDT but also supports nested parallelism, overcoming the single master problem.

SMPSs [2] is a runtime system similar to BDDT. SMPSs, however, does not handle unaligned memory addresses and requires the size of the argument to be a power of 2. If these two requirements are not met, then SMPSs may detect false dependencies between tasks and reduce performance. In comparison, PARTEE's block-based dependence analysis, supports arbitrary argument sizes and memory addresses, and may be configured to avoid false dependencies in every application.

Sequoia [15] is another parallel programming language. Sequoia requires the pro-

grammer to describe: *a*) the task graph as a hierarchy of nested parallel tasks; *b*) the memory hierarchy of the targeted machine; and *c*) the data distribution among tasks. Sequoia then inserts implicit barriers to ensure the correct execution of the application. We believe that PARTEE's programming model is more intuitive and portable. Additionally, in applications with irregular dependencies we expect PARTEE to expose more parallelism then Sequoia.

Wool [16] is a task-based parallel runtime system similar to Cilk. Wool uses the *fork-join* model to express the parallelism of applications. It also supports nested spawning of tasks, but like Cilk, it requires implicit synchronization. On the other hand, Wool supports another model of *work stealing*, which called *leap-frogging* and differs from *random* work stealing of Cilk in victim selection. In comparison, PARTEE offers also the nested spawning and an invisible synchronization by automatic detection and resolve of dependencies between tasks.

The OpenMP [17] is a runtime system that uses compiler directives to express the shared memory parallelism of loops and tasks of sequential programs. The thread management is transparent but the synchronization of loops and tasks is user's responsibility. In comparison PARTEE uses a transparent dependence analysis instead of manual synchronization.

Threading Building Blocks (TBB) [18] is a C++ template library which offers the concurrent execution of multiple operations which treated as "tasks". TBB uses also a task stealing model similar to the work stealing model applied in Cilk. Additionally, TBB offers a collection of components for parallel programming. Similar to OpenMP, the thread management is transparent but the synchronization of loops and tasks is user's responsibility. Instead, PARTEE offers a transparent synchronization of tasks.

5.2 Double Ended Queues

A lock-free deque implementation is described in Herlihy and Shavit's book, *The Art of Multiprocessor Programming* [19]. This implementation can be used to implement the A-steal algorithm but it requires a *double compare-and-swap* primitive, which is not supported in our architecture.

FastFlow [11] uses an implementation of lock-free multi-producer and multi-consumer queue. We based our implementation on this work and we enhanced it by adding new operations and by modifying the existing operations. The differences between FastFlow and FCDQ are presented extensively in subsubsection 3.2.2.5.

Michael and Scott [20] present several versions of parallel queues. More recently, Morrison et al. [21] presents FIFO queues with high concurrency using fetch-and-add instead of compare-and-swap primitives. Hendler et al. [22] present a lock-free stack implemented as an array, similarly to FCDQ. Shafiei [23] presents several array-based implementations of concurrent general-purpose data structures. Moreover, more general techniques applying to many data structures have been presented for lock-free synchronization of lists [24]. In contrast to these algorithms, FCDQ specifically targets implementations of A-steal schedulers which, by having single-threaded semantics on one end

of the deque, allow for further optimization.

5.3 Memory allocators

Hoard [25] is considered one of the best memory allocators. Hoard consists of small thread local heaps, which use the global system heap when the run out of memory. Each thread uses its own heap to satisfy its allocation and deallocation requests. When a heap runs out of memory is requests some more memory segments from the global system heap. Hoard improves the throughput on allocation and deallocation, but it does not fit in our model because Hoard does not support the *bulk* free model.

Michael [26] presents a scalable lock-free allocator that guarantees progress when threads are delayed. These delayed threads killed or deprioritized by the scheduler. Similarly to Hoard, this allocator does not fit in PARTEE because it does not support the *bulk* free model.

MAMA [27] is another parallel allocator which tries to "annihilate" the cost of allocate and free requests by combining these requests. Like Hoard, MAMA consists of multiple heaps which satisfy allocate and free requests from different threads. These requests are combined in one and served from one thread. By combining allocate and free requests of the same heap may "annihilate" the cost of these requests because the requested memory of allocate request may satisfied from the free memory of the free request. This allocator does not fit in PARTEE, because it does not support the *bulk* free model.

In contrast with PARTEE's region allocator, Titanium [28] uses "private" regions. Private regions cannot be used in PARTEE, because only the owner thread can allocate objects in them. Titanium uses private regions to reduce the overhead of the garbage collection. It also supports "shared" regions which are implemented using global barrier-like synchronization.

Myrmics Memory Allocator [29] and DRASync [30] are two hierarchical message-passing allocators which supports regions. Those two allocators introduce the region idea, as defined by Gay and Aiken, on distributed systems.

Chapter 6

Conclusions

In this thesis, we present the design and the implementation of an hierarchical block-based dynamic dependence analysis for recursively task parallel runtime systems. This idea inspired from the large increase of the multi-core systems and the requirement of the performance of these systems. BDDT and SMPSs are the two runtime systems which introduce the idea of dependence analysis in shared-memory systems, but their implementation suffered from the *single master scaling problem*, a problem which try to eliminate by making every virtual process "master".

We also present a custom, parallel, region-based allocator we used to increase the locality and concurrency of a new Cilk-like runtime system. During the development of this thesis we came up against memory allocation and deallocate overheads, either time overheads or space overheads. This lead us to develop a parallel region-based allocator to minimize these overheads.

Additionally, we develop a new double-ended queue algorithm called FCDQ and we evaluate it with the existing double-ended queue algorithms. As noted above BDDT and SMPSs suffers for the *single master* scale problem, we follow the scheduling policy which introduced by Cilk. This model requires a double-ended queue assigned in every virtual process, which used for task scheduling. We evaluate FCDQ which existing double-ended queue algorithms and we choose the algorithm with best throughput.

Finally, we design and implement, PARTEE, a recursively, task-based, parallel runtime system which combines the dependence analysis, the region-based allocator and the best double-ended queue algorithm and we found that in cases where task dependencies are irregular, PARTEE outperforms Cilk, a task-parallel runtime without implicit task synchronization by up to 54%.

Bibliography

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the 1995 conference on Principles and Practice of Parallel Programming*, ser. PPOPP '95, 1995, pp. 207–216.

[2] *SMP Superscalar (SMPSs) v2.3 User's Manual*, 2010.

[3] G. Tzenakis, A. Papatriantafyllou, H. Vandierendonck, P. Pratikakis, and D. Nikolopoulos, "BDDT: Block-level Dynamic Dependence Analysis for Task-Based Parallelism," in *Proceedings of the 2013 International Conference on Advanced Parallel Processing Technology*, ser. Lecture Notes in Computer Science, 2013.

[4] Y. He, W.-J. Hsu, and C. Leiserson, "Provably efficient two-level adaptive scheduling," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, E. Frachtenberg and U. Schwiegelshohn, Eds. Springer Berlin Heidelberg, 2007, vol. 4376, pp. 1–32. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71035-6_1

[5] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 356–368.

[6] I. Angelina, L. Silas, B. Zhiyi, H. Charles, and E. Leiserson, "Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems," in *International Conference on Parallel Architecture and Compilation Techniques*, 2010.

[7] F. Zakkak, D. Chasapis, P. Pratikakis, A. Bilas, and D. Nikolopoulos, "Inference and Declaration of Independence in Task-Parallel Programs," in *Proceedings of the 2013 International Conference on Advanced Parallel Processing Technology*, ser. Lecture Notes in Computer Science, C. Wu and A. Cohen, Eds., vol. 8299. Springer Berlin Heidelberg, 2013, pp. 1–16.

[8] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05. New York, NY, USA: ACM, 2005, pp. 21–28.

[9] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '98. New York, NY, USA: ACM, 1998, pp. 119–129. [Online]. Available: http://doi.acm.org/10.1145/277651.277678

[10] IBM, "IBM System/370 Extended Architecture, Principles of Operation," Tech. Rep. Publication No. SA22-7085, 1983.

[11] M. Aldinucci, M. Torquati, and M. Meneghin, "Fastflow: Efficient parallel streaming applications on multi-core," *arXiv preprint arXiv:0909.1187*, 2009.

[12] D. Gay and A. Aiken, "Memory management with explicit regions," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 313–323.

[13] E. Fredkin, "Trie memory," *Communications of ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81.

[15] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, 2006, p. 4.

[16] K.-F. Faxén, "Wool-a work stealing library," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 93–100, Jun. 2009.

[17] OpenMP Architecture Review Board, "Openmp application program interface version 4.0," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, Jul 2013.

[18] Intel, "Threading building blocks," 2014, version 4.2, https://www.threadingbuildingblocks.org/.

[19] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[20] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96. New York, NY, USA: ACM, 1996, pp. 267–275.

[21] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 103–112.

[22] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04. New York, NY, USA: ACM, 2004, pp. 206–215.

[23] N. Shafiei, "Non-blocking array-based algorithms for stacks and queues," in *Proceedings of the 10th International Conference on Distributed Computing and Networking*, ser. ICDCN '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 55–66.

[24] P. Fatourou and N. D. Kallimanis, "Highly-efficient wait-free synchronization," *Theor. Comp. Sys.*, vol. 55, no. 3, pp. 475–520, Oct. 2014.

[25] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *SIGPLAN Not.*, vol. 35, no. 11, pp. 117–128, Nov. 2000. [Online]. Available: http://doi.acm.org/10.1145/356989.357000

[26] M. M. Michael, "Scalable lock-free dynamic memory allocation," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: ACM, 2004, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/996841.996848

[27] S. Kahan and P. Konecny, ""mama!": A memory allocator for multithreaded architectures," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '06. New York, NY, USA: ACM, 2006, pp. 178–186. [Online]. Available: http://doi.acm.org/10.1145/1122971.1122999

[28] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, "Titanium language reference manual," Berkeley, CA, USA, Tech. Rep., 2001.

[29] S. Lyberis, P. Pratikakis, D. S. Nikolopoulos, M. Schulz, T. Gamblin, and B. R. de Supinski, "The myrmics memory allocator: Hierarchical,message-passing allocation for global address spaces," in *Proceedings of the 2012 International Symposium on Memory Management*, ser. ISMM '12. New York, NY, USA: ACM, 2012, pp. 15–24.

[30] C. Symeonidou, P. Pratikakis, A. Bilas, and D. S. Nikolopoulos, "Drasync: Distributed region-based memory allocation and synchronization," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: ACM, 2013, pp. 49–54.