RICE UNIVERSITY

Optimized Event-Driven Runtime Systems for Programmability and Performance

by

Sağnak Taşırlar

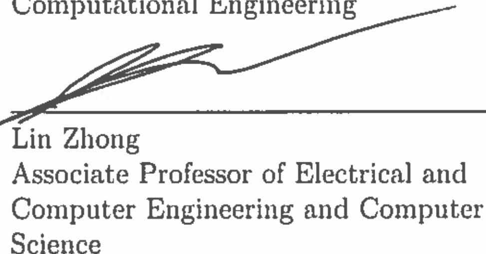
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar
Professor of Computer Science
E.D. Butcher Chair in Engineering

Keith D. Cooper
L. John and Ann H. Doerr Professor of Computational Engineering



Lin Zhong
Associate Professor of Electrical and Computer Engineering and Computer Science

Houston, Texas

October, 2015

ABSTRACT

Optimized Event-Driven Runtime Systems for Programmability and Performance

by

Sağnak Taşırlar

Modern parallel programming models perform their best under the particular patterns they are tuned to express and execute, such as OpenMP for fork/join and Cilk for divide-and-conquer patterns. In cases where the model does not fit the problem, shoehorning of the problem to the model leads to performance bottlenecks, for example by introducing unnecessary dependences. In addition, some of these models, like MPI, have a performance model which thinly veils a particular machine's parameters from the problem that is to be solved.

We postulate that an expressive parallel programming model should not overconstrain the problem it expresses and should not require the application programmer to code for the underlying machine and sacrifice portability. In our former work, we proposed the Data-Driven Tasks model, which constitutes expressive and portable parallelism by only requiring the application programmer to declare the inherent dependences of the application. In this work, we observe another instantiation of macro-dataflow, the Open Community Runtime (OCR) with work-stealing support for directed-acyclic graph (DAG) parallelism.

First, we assess the benefits of these macro-dataflow models over traditional fork/join models using work-stealing, where we match the performance of hand-tuned parallel libraries on today architecture through DAG parallelism. Secondly, we ad-

dress work-stealing granularity optimizations for DAG parallelism to address how work stealing can be extended to perform better under complex dependence graphs. Lastly, we observe the impact of locality optimizations for work-stealing runtimes for DAG-parallel applications.

On our path to exascale computations, the priority is shifting from minimizing latency to energy saving as the current trend makes powering an exascale machine very challenging. The trend of providing more parallelism to fit power budgets succeeds if applications can be declared to be more parallel and also scale. We argue that macro-dataflow is a framework that allows programmers to declare unconstrained parallelism. We provide an underlying work-stealing runtime to execute this framework for load balance and scalability, and propose heuristics to extend the default work-stealing approach to better perform with DAG parallel programs. We present our results on a multi-socket many-core machine and a many-core accelerator to showcase the feasibility of our approach on architectures signaling what future architectures may resemble.

Acknowledgments

I would like to thank my PhD adviser Vivek Sarkar and my committee members Keith Cooper and Lin Zhong for taking the time to listen to my defense and provide feedback.

This has been such an arduous road that plenty who would not fit in this page have made bearable. I dedicate this thesis to these ‘unknown soldiers’.

Contents

Abstract ii

Acknowledgments iv

List of Illustrations viii

1 Introduction 1

1.1 Motivation 1

1.2 Contributions 4

1.3 Organization 4

1.4 Thesis Statement 5

2 Background 6

2.1 Parallel Programming Models 6

2.1.1 Thread based parallelism 7

2.1.2 Data parallelism 8

2.1.3 Task parallel languages 10

2.1.4 Macro-Dataflow 11

2.2 Series-parallel graphs 12

2.3 Futures 13

2.4 Habanero 14

2.4.1 Constructs 14

2.4.2 Runtime 14

3 Dependence: A Declarative Approach to Synchronization 16

3.1 Motivation 16
3.1.1 Imperative vs Declarative 16
3.1.2 Restricted set of task-graphs 19
3.2 Macro-Dataflow models 21
3.2.1 Habanero C with Data-Driven Tasks and Futures 21
3.2.2 Open Community Runtime 26
3.3 Results 29
3.3.1 Methods and environments 29
3.3.2 An inefficient Fibonacci 31
3.3.3 Smith-Waterman/Needleman-Wunsch sequence-alignment . . 33
3.3.4 Cholesky decomposition 37

4 Efficient work-stealing in Event-Driven Runtime Systems 42

4.1 Introduction 42
4.2 Granularity 44
4.3 Successor task heuristics 48
4.3.1 Pessimistic descendance 49
4.3.2 Optimistic descendance 50
4.4 Task queues 51
4.4.1 Deques 52
4.4.2 Prioritized Data Structures 53
4.5 Experiments 55
4.5.1 Methods and environments 55
4.5.2 An inefficient Fibonacci 56
4.5.3 Smith-Waterman/Needleman-Wunsch sequence-alignment . . 62
4.5.4 Cholesky decomposition 68

5 Locality-aware Scheduling in Event-Driven Runtime Sys-

tems **81**

5.1 Introduction 81
5.2 Task queues 83
5.2.1 Deques 83
5.2.2 Explicitly prioritized data-structures 86
5.2.3 Recursive data structures 88
5.3 Task scheduling heuristics 89
5.3.1 Alternative push policies 91
5.4 Task stealing heuristics 92
5.4.1 Victim selection 92
5.4.2 Task extraction 93
5.5 Results 96
5.5.1 Sequence Alignment 97
5.5.2 Cholesky decomposition 103

6 Conclusions & Future Work **110**

6.1 Conclusion 110
6.2 Future Work 111
6.2.1 Less contentious task queues 111
6.2.2 Better cache simulation 112
6.2.3 Energy modeling 112
6.2.4 Topology-aware stealing 112
6.2.5 Distributed memory support 113

Bibliography **114**

Illustrations

3.1 Actual dependences of getting dressed, figure credit [1] 17

3.2 Expressed dependence of getting dressed, figure credit [1] 18

3.3 Subset relation for {a,b,c} 19

3.4 Subset relation for {a,b,c} expressed with C++11 futures 20

3.5 Task graph for calculating $fib(24)$ with cut-off 20 31

3.6 $fib(50)$ with cut-off 25 results for the Xeon Machine using Intel CilkPlus 32

3.7 $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 32

3.8 $fib(45)$ with cut-off 25 results for the XeonPhi Machine using Intel CilkPlus 33

3.9 $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 33

3.10 Dependence graph for a 4 by 4 tiled string matching 34

3.11 Score graph for a 5 by 7 string matching 36

3.12 Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 36

3.13 Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 36

3.14 Dependence graph for a 5 by 5 tile cholesky factorization 38

3.15 Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine . . . 39

3.16 Cholesky decomposition results for a 12K by 12K matrix using Intel MKL for the Xeon machine 39

3.17 Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine 40
3.18 Cholesky decomposition results for a 6K by 6K matrix using Intel MKL for the XeonPhi machine 40

4.1 Possible decomposition and mapping of a divide and conquer problem, figure credit [2] 43
4.2 A pathological task graph that is recursively left-skewed 45
4.3 Task graph of a divide-and-conquer application run under *work-first* policy 46
4.4 Snapshot of a stolen $task_A$ and its immediate successors $task_B$ and $task_C$ 47
4.5 A pathological heap for leaf-first breadth-first stealing 55
4.6 $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 lock-free vs locked ready task queues 57
4.7 $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 partially sorted descendance heuristic impact on execution time . . . 58
4.8 $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 partially sorted descendance heuristic impact on steal attempts 59
4.9 $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 sorted descendance heuristic impact on execution time 59
4.10 $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 sorted descendance heuristic impact on steal attempts 60
4.11 $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 lock-free vs locked ready task queues 60
4.12 $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 partially sorted descendance heuristic impact on execution time . 61

4.13 $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 partially sorted descendance heuristic impact on steal attempts . 61
4.14 $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 sorted descendance heuristic impact on execution time 61
4.15 $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 sorted descendance heuristic impact on steal attempts 62
4.16 Optimistic descendance count graph for sequence alignment 63
4.17 Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 lock-free vs locked ready task queues 64
4.18 Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 partially sorted descendance heuristic impact on execution time 64
4.19 Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 partially sorted descendance heuristic impact on steal attempts 65
4.20 Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 sorted descendance heuristic impact on execution time 66
4.21 Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 sorted descendance heuristic impact on steal attempts 66
4.22 Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 lock-free vs locked ready task queues 67
4.23 Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 partially sorted descendance heuristic impact on execution time 67
4.24 Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 partially sorted descendance heuristic impact on steal attempts 68

4.25 Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 sorted descendance heuristic impact on execution time 69
4.26 Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 sorted descendance heuristic impact on steal attempts 69
4.27 Optimistic descendance counts for cholesky factorization 70
4.28 Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine lock-free vs locked ready task queues 72
4.29 Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine partially sorted descendance heuristic impact on execution time . . . 73
4.30 Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine partially sorted descendance heuristic impact on steal attempts 74
4.31 Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine sorted descendance heuristic impact on execution time 74
4.32 Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine sorted descendance heuristic impact on steal attempts 75
4.33 Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine lock-free vs locked ready task queues 76
4.34 Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine partially sorted descendance heuristic impact on execution time . . . 77

4.35 Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine partially sorted descendance heuristic impact on steal attempts 77
4.36 Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine sorted descendance heuristic impact on execution time 78
4.37 Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine sorted descendance heuristic impact on steal attempts 78
4.38 Cholesky decomposition results for a 6K by 6K matrix with 64 by 64 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine partially sorted descendance heuristic impact on execution time . . . 79
4.39 Cholesky decomposition results for a 6K by 6K matrix with 64 by 64 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine partially sorted descendance heuristic impact on steal attempts 79
4.40 Cholesky decomposition results for a 6K by 6K matrix with 64 by 64 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine sorted descendance heuristic impact on execution time 80
4.41 Cholesky decomposition results for a 6K by 6K matrix with 64 by 64 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine sorted descendance heuristic impact on steal attempts 80

5.1 Task graph of a divide-and-conquer application run under *work-first* policy 84
5.2 Task graph of a divide-and-conquer application run under *help-first* policy 84
5.3 Non series-parallel task graphs 85

5.4 Locality implications of former heuristics on matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 98
5.5 Locality implications of former heuristics using locality priorities on matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 100
5.6 Locality implications of former heuristics using non-local locality aware pushing on matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 101
5.7 Locality implications of former heuristics using locality priorities and non-local locality aware pushing on matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 102
5.8 Locality implications of former heuristics on Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine 104
5.9 Locality implications of former heuristics using locality priorities on Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine . . . 105
5.10 Locality implications of former heuristics using non-local locality aware pushing on Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine 107
5.11 Locality implications of former heuristics using locality priorities and non-local locality aware pushing on Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine 109

Chapter 1

Introduction

1.1 Motivation

If Zeno of Elea were alive today, predictions on Moore's law's end would have made it into his set of paradoxes. Since we can still cram more components onto integrated circuits, and while clock frequencies have leveled off with the end of Dennard scaling, there is now widespread adoption of parallel processors in multiple forms including: multi-core architectures, graphics units as today's vector machines and heterogeneous combinations of big/small cores. This adoption is providing the proverbial oxygen to parallel programming models in their equivalent of the Cambrian explosion. Following the allegory, different parallel programming models fill their ecological niches.

The first niche among shared-memory parallel programming models are thread-level models, in which the user maintains the threads of execution at a coarse granularity. Adopters of these models have to use mutexes, semaphores and the like to declare critical regions around parts of their code that are not safe for concurrent execution. This is hard to accomplish and error-prone; moreover, concurrency bugs are also rather hard to debug. Additionally, the coarse granularity of these models lead to load imbalances, and therefore under-utilization of processors. If the implementer over-subscribes the underlying machine by declaring their parallelism at a finer grain through more threads to resolve these imbalances, they would be penalized by the not-so-cheap context-switching overheads.

Another class of parallel programming models, namely data-parallel models, targets problems for which the same computation is performed on a set of data. This class of computations is the easiest to declare, and often results in *embarrassingly parallel* computations. The constructs necessary for these problems can be as simple as higher-order functions in lambda-calculus (e.g. `map`(`forall`), `reduce`), and many large-scale systems have been built to perform data-parallel computations efficiently. Though these models have been quite popular, they are only applicable to a constrained subset of problems. For problems with phases of data-parallel regions, better performance utilization requires the use of more general programming models.

On the other hand, task parallel models are used to declare problems where the parallelism occurs among different tasks. One can argue that the data-parallelism model discussed above is a special case of task parallelism which uses different invocations of the same task on subsets of program data. Task parallelism, however, can enable more complicated parallel program structures to be created by indicating which tasks can be executed in parallel with other tasks, and implicitly (or explicitly) imposing a synchronization among these tasks. These models can build programs with dependency graphs that are more general than the data-parallel models mentioned above.

Distributed memory models delegate the mapping and scheduling of tasks and data to users, for the most part. These models tend to suffer less from pitfalls of threading, like non-determinacy, because the low-level semantics of the underlying protocols have to be known to the programmer. Though there is a community of *ninja* programmers who enjoy the control and performance they get from these models, delegating control of scheduling and mapping to the user is inherently unscalable, and leads to productivity and performance losses for most mainstream programmers.

The increasing cost of cache-coherence and possibly non-uniform distance, and therefore cost, of memory from compute increases the likelihood of future systems not being uniform shared-memory models. However, for the sake of programmer productivity, future machines are likely to have islands of coherence. For today's parallel systems, a common example of this hybrid parallelism can be found in the use of MPI across nodes on a cluster, where a node utilizes thread-level or fork/join parallelism like OpenMP. At the time of this work, this hybrid model is referred to as MPI+X, where X represents a shared-memory programming model.

Dataflow models are devised for massive parallelism and the early work has been on the hardware side. On the software end of dataflow models, we can describe this model as data driving the computation on a task graph built by the user. For an everyday computer user, a spreadsheet application is a good example. For practical and historical reasons, dataflow architectures did not catch on, and exploiting the fine granularity of a dataflow model on today's architectures would be prohibitively expensive. A compromise is a macro-dataflow model, where dataflow graphs are build based on tasks suitable for today's machines, rather than on instructions. Macro-dataflow models enable task dependences to be expressed as a general directed acyclic graph (DAG), and are more expressive than fork/join task parallel models. Additionally, macro-dataflow models can be used to bridge between shared-memory and distributed-memory models via a single paradigm.

A macro-dataflow parallel programming model can express arbitrary graphs of computation, however this does not solely guarantee performance. In our former work [3], we have shown how a macro-dataflow parallel programming model could be implemented on a work-sharing runtime for competitive parallel performance. We extend this work for work-stealing, as work-stealing suffers less contention compared

to a work-sharing runtime and is therefore more scalable by design. However, work-stealing runtimes are designed for task-parallel models with certain restrictions (e.g., series-parallel computation graphs) that do not necessarily apply to macro-dataflow models.

This thesis explores the implementation of macro-dataflow models on top of work-stealing runtimes, with extensions to obtain better performance through granularity and locality optimizations to the underlying runtime.

1.2 Contributions

In this thesis, we

- Introduce new work-stealing runtime algorithms for macro-dataflow parallelism.
- Contribute our ideas to Habanero languages and a library implementation of a publicly available macro-dataflow runtime: version 0.7 of the Open Community Runtime (OCR).
- Introduce work-stealing heuristics for granularity and locality optimizations for unrestricted DAG-parallelism
- Present results for various scheduling policies on theoretical and practical metrics for multiple benchmarks on multiple platforms

1.3 Organization

Chapter 2 provides the background for our work and provides the foundation for the unaccustomed reader. Chapter 3 presents our vision to make dependence a first order construct, how that leads to our macro-dataflow implementations. Chapter 4

describes our approaches to further fit macro-dataflow on work-stealing runtimes. Chapter 5 discusses locality optimizations for work-stealing macro-dataflow runtimes. Lastly, chapter 6 concludes our work and provides future directions to our research.

1.4 Thesis Statement

Macro-dataflow parallelism and event-driven runtime systems offer programmability and performance benefits for applications with complex dependence structures. These runtime systems can also be extended to address new granularity and locality concerns for programs with dependence structures that are more general than fork-join parallelism.

Chapter 2

Background

As we introduce new parallel programming models and how they compare to established work, section 2.1 will discuss background information on parallel programming models, in further detail than in section 1.1. We discuss series-parallel graphs and their properties in section 2.2 because of their correspondence to nested fork/join task parallel programming model, and then introduce futures in section 2.3, as new variants of this construct will be introduced later in chapter 3. Finally, we summarize the Habanero language constructs and runtime in section 2.4 to familiarize the reader with the language and the underlying runtime that provide the foundation for this thesis.

2.1 Parallel Programming Models

An extensive background on parallel programming models is beyond the scope of this document, so we choose to restrict the scope only to models that are of immediate relevance to our work. One might argue that these are simple, orthogonal classifications of established work, since many models adopt multiple different aspects of parallel programming. As all models by utility have to be Turing complete, any problem can be expressed by all of them but in varying difficulties of expression and performance characteristics. We will compare the choices these models make, the intent behind them and how these affect the ease of expression and performance.

2.1.1 Thread based parallelism

Native thread libraries (e.g. PThreads)

In order to utilize multiple execution units on a parallel architecture, these models require the user to declare multiple threads of execution explicitly. For the sake of simplicity, let us assume that there is a one-to-one correspondence between execution units and threads of execution unless stated otherwise.

If these multiple threads of execution share data amongst each other, which they do if they are not embarrassingly parallel, the user has to guarantee the safety of accesses through language constructs like mutexes, semaphores for critical sections and et cetera. These are common pitfalls where the user has to have mastery of these constructs and their underlying semantics to write correct code that performs scalably [4, 5].

Another concern is granularity, since a typical PThreads user statically partitions the problem into equal or nearly-equal parts, assuming that perfect load balance leads to the best performance. The first optimistic assumption here is that a reasonable static partitioning is possible ahead of program execution; in practice, complex and dynamic dependence patterns may make this hard to achieve. Even then, perfect load balance may still lead to suboptimal performance due to variability in task execution times e.g., if the computation is run on a multi-programming environment, where a thread may lag behind, or on a machine where an execution unit may speed up and slow down dynamically, or may contain a critical section that exhibits different execution times depending on the (nondeterministic) ordering constraints for lock acquisition.

However, these models are relevant to our work because we advocate the delega-

tion of the user's responsibility of load balance through over-subscription, dynamicity, granularity management and proper handling of shared data across threads of execution to the runtimes of higher level programming models. Implementations of task parallel programming models' runtimes are themselves applications that utilize thread level parallelism.

OpenMP [6]

provides annotations in the form of pragmas to achieve parallelism. There are pragmas to declare what computation can be run by which thread, divide work across threads and what distribution to use, where critical sections are, et cetera. These pragmas provide syntactic sugar to a thread library approach and can easily be integrated into legacy code, though with compiler support. Therefore the performance and the expressibility concerns from PThreads remain, but with less user involvement. As the model allows nested fork/joins operations and recently tasks, the programming model can now be used as more than a thin wrapper for PThreads.

2.1.2 Data parallelism

When the same computation is to be applied to a set of data, we can classify this as data parallelism. As suggested before, this kind of parallelism can be declared with a higher-order function as in lambda-calculus': `map`.

Initial work on data parallelism has been on vector machines and vectorizing compilers to apply an instruction not just to a single register but to a wider size of data. Parallelism is achieved by the computation being applied to multiple data elements simultaneously, if they are independent from each other. This can be declared by the user through annotations, or array languages, or by optimization through vectoriz-

ing compilers. Vector parallelism has recently bounced back in popularity as many vendors are increasing the width of SIMD-parallelism supported by hardware.

Within the last decade, the use of graphics units for generic computation has gained serious traction, aptly named GPGPU. These graphics cards can also be classified as vector machines attached as accelerators to CPU host machines. These architectures are hierarchical in nature, where the leaf level are vector units. Though the programming models for these machines (CUDA, OpenCL) have been relaxing the constraints for expressible programs, the nature of these models is to support very wide data-parallelism. The user, through syntactic sugars, map subsets of data through hierarchical decompositions of the architecture.

Another realm of data-parallelism is one level above on systems hierarchy: the cluster. For the sake of brevity, we will try to keep discussions on distributed memory data parallelism. However, the most commonly utilized parallel programming model is Flynn taxonomy's SPMD, namely single program multiple data. The data is scattered throughout explicit tuples of memory domain and execution units (e.g. MPI ranks), where every execution unit works on its assigned data. In the business world, the same paradigm to process big amounts of data lead to data-parallel programming models like Map-Reduce, whose name is influenced by the higher order functions `map` and `reduce`, that are inherent to data-parallel models.

It may be obvious to the reader that these models only solve a subset of problems that can be expressed in a data-parallel fashion. Additionally, multiple stages of data-parallel regions in a problem may require special attention to avoid bottlenecks and loss of locality across parallel stages.

2.1.3 Task parallel languages

As we hinted in section 2.1.1, task parallelism can be described as a special case of thread-level parallelism to address its limitations.

Task parallel models introduce new constructs to control flow for explicit parallelism but in a finer granularity than thread based models, hence the name 'task'. User declares parallel procedures and how they are synchronized, where an underlying runtime handles the mapping and scheduling aspects. These address the concerns mentioned in section 2.1.1.

Most common of these constructs are flavors of *fork* and *join*. A fork operation creates an alternate and parallel flow of control to the context from which it is called. In contrast, a join operation merges more than one flows of control onto one. These constructs occur in thread based models, too but the differentiating factor is the granularity used for best practices.

As the user delegates the mapping and scheduling of contexts enough to oversubscribe the threads of execution, a runtime program claims this responsibility. Runtimes have to address the problems aforementioned in section 2.1.1, mapping tasks to threads (i.e. scheduling) for better utilization (i.e. load balance). There are two common scheduling techniques for load balancing these dynamically unfolding fork/join graphs.

Work-sharing is a 'push' model; as tasks become ready and handed to a scheduler, they are either pushed to a shared task queue for pick up by a set of workers (like a list scheduler) or eagerly distributed amongst workers. The former approach introduces contention over the shared task queues and the latter does not take into account how busy the other lists are. For example, our initial implementation of macro-dataflow [3]

utilized a global task pool scheme for its ease of implementation.

Work-stealing is a 'pull' model; as every thread of execution maintains its own pool of tasks and starts extracting tasks from other contexts once it runs out of work. This distributed list nature of the scheduler alleviates contention and under particular assumptions of victim selection and task extraction policies provides tight bounds on scaling and memory usage [7].

We will discuss work-stealing schedulers in further detail in the upcoming sections and chapters, since it is the basis of the runtime scheduler on which we support our macro-dataflow programming models.

2.1.4 Macro-Dataflow

We briefly introduced macro-dataflow model in section 1.1. As stated before, dataflow ideas were initiated for computer architecture for initially explored as a direction for fine-grained hardware parallelism, where operands' availability led to an instruction rather than a program counter [8]. Modern architectures have only adopted this model in a limited way with the use of out-of-order superscalar execution engines. One exception in recent literature advocating for dataflow architectures for a specific domain is Anton [9]. We will opine further about this as our motivation in section 3.1.

In today's architectures architectures, creating tasks at the level of fine-grained statement-level parallelism would overwhelm a runtime scheduler. If the user declares a scope of sufficiently granular computation as a task, just as they would in task parallel models, but instead utilizes the dataflow concept of readiness driving scheduling, we classify these models as macro-dataflow models [10, 11, 12, 13, 14, 15, 16].

The distinction of macro-dataflow is in the declaration of the parallelism. Task or

thread driven models declare control flow paths to extract parallelism, whereas macro-dataflow models require programmers to declare the inherent dependences between tasks. The comparison between macro-dataflow and task parallel models is analogous to the comparison between declarative and imperative languages: *what* vs *how*.

Macro-dataflow programming models allow a user to declare any possible dependence graph, as dependence is exposed as a user-level construct. However, task-parallel models may restrict the set of problems that can be declared without introducing additional dependences. This is further elaborated in section 3.1.

2.2 Series-parallel graphs

A series-parallel graph can be formulated inductively by defining a series or a parallel composition applied to two series-parallel graphs. The base case for this induction is the unit series-parallel graph, which only consists of a *sink* and a *source* node. Since series-parallel graphs are a subset of partially-ordered sets, one can think of sink and source nodes as greatest and least elements of a partial-order. A *series* composition of two series-parallel graphs, (g_1, g_2), merges the sink node of g_1 and the source node of g_2. A parallel composition of series-parallel graphs (g_1, g_2) merges the source node of both graphs as the resulting graph's source and merges the sink node of both graphs as the sink node of the resulting graph.

Series-parallel graphs are *proper* subsets of partially ordered sets, and can not declare all possible partially-ordered sets. One example is the *subset* relation.

In our discussion of task-parallel languages in section 2.1.3, we discussed the language constructs: *fork* and *join*. Nested use of these constructs yield task graphs that are series-parallel [17]. A fork and its matching join operation is a parallel decomposition of the graph corresponding to the context where those operations are called,

and the graph that corresponds to the context created within those calls.

2.3 Futures

A *future* is a language construct that serves as a reference to a value. The state of this value is not known till observation, and the act of observation resolves the value, thus it is a Schrödinger's value.

Futures were initially formulated in the literature as a 3-tuple, consisting of storage for the value, pending consumers and the producer [18, 19]. Therefore, a future object handle serves the storage purpose; the contexts where the future is asked to be resolved is maintained as the pending consumers, and the user is required to declare the producing task when declaring the future. When the future is asked to be resolved, if the runtime scheduler executed the producer task and updated the storage, the resolution just amounts to referencing the storage. However, if the resolution has not happened yet, the common implementation is to block the current context and execute the producer task to resolve the value and continue with the context, since the context after the futures observation is assumed to be dependent on the future's resolution. Otherwise, an analysis would be required to legitimize an ordering optimization.

Futures can be used to declare dependences and therefore build task-graphs, since they create a producer-consumer relationship between contexts of declaration and resolution [20]. However, the blocking resolution of futures would lead to inefficiencies. The blocked context can either be packaged and restored or a thread would be blocked to store the context implicitly and a new one would resolve the producer task. To alleviate these possible problems, we explore future constructs like *data-driven tasks/futures* [21] with non-blocking implementations. The model will be

introduced and elaborated in section 3.2.1.

2.4 Habanero

Habanero model is a dynamic light-weight task parallel model, implemented in various languages (Scala, Java, C++, C [22], etc) with parallelism extensions for those languages. We utilize Habanero-C [22] and Habanero-Java [23] as a basis for our data-driven tasks and Concurrent Collections work.

2.4.1 Constructs

async declares a child task which is semantically parallel to the parent task, like a *fork* operation. The body of the async construct specifies the computation to be performed in the child task. The child task can usually copy in local variables from the parent task's lexical scope; however, copying local variables from a child task to a parent task may not always be safe since a child task may outlive its parent.

finish synchronizes all the async tasks in its scope, like a *join* operation. Finish and async constructs may be nested within each other to an unbounded depth; each async task synchronizes with its Immediately Enclosing Finish (IEF).

phaser [24] declares a point-to-point synchronization object, which can define producer-consumer relationships. We mention them as a reference, since their directed dependence structure are akin to that of data-driven futures.

2.4.2 Runtime

Habanero models support work-sharing and work-stealing runtimes. Our initial work was on Habanero-Java's work-sharing implementation [21, 3]. The implementations

of the Habanero model that we utilize in this thesis builds on the work-stealing implementation.

Work-sharing runtime implementation uses a multiple producer multiple consumer first-in first-out queue as a task pool. This pool accumulates tasks that are dynamically created from multiple contexts and makes them available to worker threads looking for work to execute. In order to avoid deadlock while blocking calls are observed (e.g. synchronizing for a finish scope) a new thread is created to continue executing tasks, when a blocked thread maintains the continuation of the blocked call, waiting to be unblocked.

Work-stealing runtime implementation uses a double-ended last-in first-out single producer multiple consumer task queue (deque) per thread of execution (worker thread, as in Cilk [25] implementation). Dynamically created tasks get pushed in a last-in fashion and work is locally extracted in a first-out fashion. When a worker thread runs out of work, it selects a random victim worker thread's deque to steal a task from first-in end of the deque.

The impact of the path chosen in the task graph traversal is elaborated in [26] and the locality implications of this are covered in [27]. The runtime also supports extensions for locality that allows a hierarchical structure (i.e. tree) of deques to be used instead of a flat list [28]. Particular traversals of this tree of tasks with explicit user placement of tasks is observed in previous work. We will elaborate on this further in comparison to our work.

Chapter 3

Dependence: A Declarative Approach to Synchronization

3.1 Motivation

3.1.1 Imperative vs Declarative

Most popular architectures of our day (x86, ARM, etc) are modifications to original von Neumann machines *. Inherent to this design are: the program counter (what to do next) and the state of the memory (what has happened before). Imperative languages declare a sequence of instructions to the machine, where what will happen next and what has happened before are how programs are declared. Hence the evolution of programming languages favored imperative languages, as they provide a better fit for the underlying architecture.

Firstly, let us observe the implications of imperative programming without delving into parallelism. An imperative program consists of a sequence of actions declared by the programmer, where implicitly in between every statement is a state change to the underlying machine. To a programmer, the sequence in a program may seem an arbitrary choice and a different order could instead have been chosen †. However for the environment, all the way from the compiler to the chip, that order is fixed, and is

*For the pedantic reader, they are modified Harvard architectures [29, 30]

†Any problem can be viewed as a partially-order set of tasks, where the partial order relation is the dependence relation. By order-extension principle, there is at least one strictly total order relation (one legal topological-sort of this relation).

considered to be the true meaning of the program. Optimizations on all these levels have to reverse-engineer the programmer's intent and check if these statements can be reordered, or otherwise manipulated to maintain the same semantics for better performance for metrics of choice.

We argue that an imperative program is an arbitrary topological sorting of the inherent dependence graph of that program. This restricts optimization decisions to the motto: 'Everything is banned unless it is allowed by dependence analysis'. We instead propose dependence as a user-lever construct to allow better optimizations, and changing the motto to: 'Everything is allowed unless it is banned by dependence declarations'.

Let us look at a sample dependence graph for the algorithm of getting dressed on figure 3.1 from [1]. The dependence relation imposes an ordering between items of clothing; for example, according to this sample graph, one can not wear a tie before a shirt is worn.

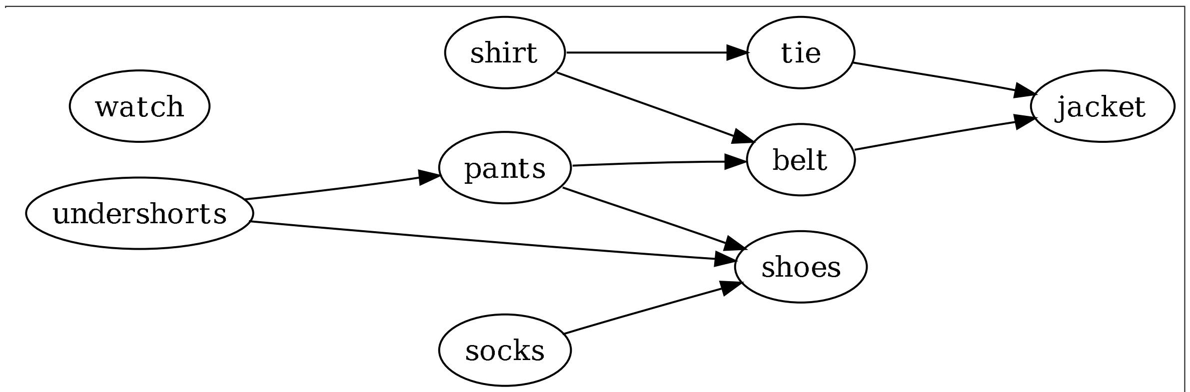


Figure 3.1 : Actual dependences of getting dressed, figure credit [1]

On imperative programs, this sample graph can only be declared in a linear fashion. One alternative result of topologically sorting the dependence graph on figure 3.1

is shown on figure 3.2, also from [1]. Now unless a compiler, a runtime system can deduce figure 3.1 from figure 3.2, figure 3.2 is the only legal schedule of getting dressed. Though, one can easily observe that the watch can be worn at any time according to the dependence graph provided.

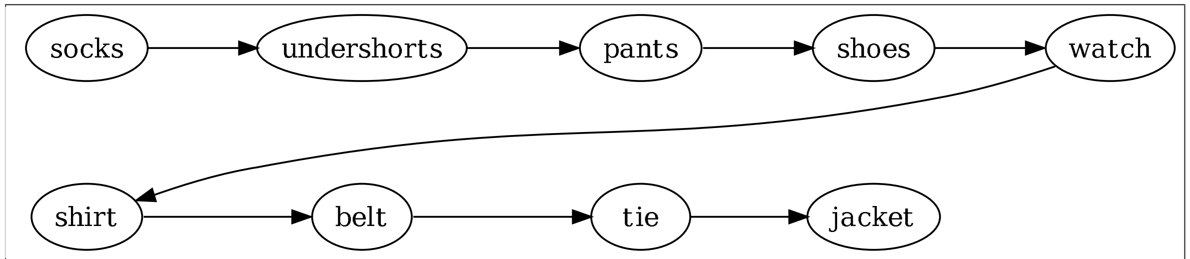


Figure 3.2 : Expressed dependence of getting dressed, figure credit [1]

The conditions on ordering and state becomes more problematic, once we consider the implications of parallel programming for parallel architectures. Given a machine with N execution units, there are N program counters to decide what will happen next and this makes utilization (load-balancing) a bigger concern. Secondly, these N execution units have intractable [31] number of possible states (schedules, and therefore states of the machine). Both these problems, in addition to the inherited pitfalls of imperative programming, led to parallel programming models that either constrain the expressivity of the model for performance and safety of that particular subset, or relinquish a lot of control (but also performance pitfalls) to the programmer.

Additionally, given that the necessary ordering constraints between objects are declared as dependences, the legal schedules prevent any ordering hazards. Though, we burden the programmer with expressing these dependences, the programmer does not have to guess what the underlying semantics for memory orderings are. The legal orderings must all obey the dependences specified by the programmer.

3.1.2 Restricted set of task-graphs

We argue that explicit dependence declaration allows further scheduling opportunities than popular programming models by not constraining how problems can be declared. In section 2.2, we give a brief introduction to series-parallel graphs and how nested fork/join task-parallel models create these graphs. Additionally, we also mentioned how series-parallel graphs are a proper subset of all partial orders and therefore can not describe all possible partial orders, like the subset relation in figure 3.3.

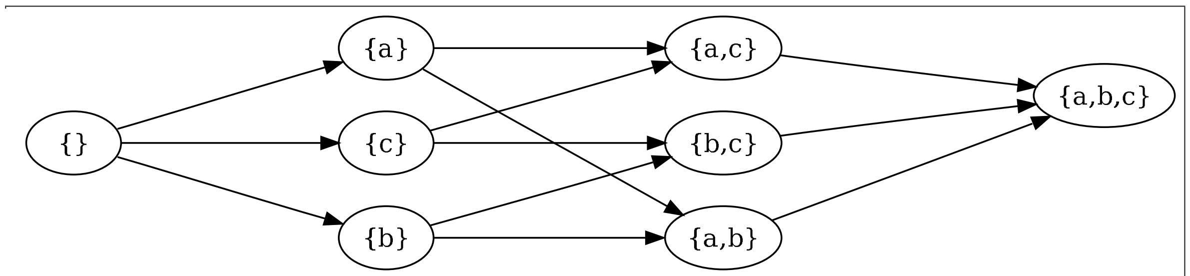


Figure 3.3 : Subset relation for {a,b,c}

There is no series of series/parallel decompositions to reduce that graph to a single node. Recognition of two-terminal series-parallel graphs is beyond the scope of this document but algorithms can be found in [32].

First-order dependence constructs however would allow any arbitrary task graph to be built, as a dependence construct would serve as the relation between any two nodes, or the directed arcs seen on the sample task graphs. We have covered the *future* construct in section 2.3 and suggested they can be used as dependence constructs. As an example for this, figure 3.4 declares the subset graph in figure 3.3, using C++11 syntax for futures.

```
#include <iostream>     #include <future>   #include <thread>

int main() {
    std::packaged_task<const char*()>  empty([](){ return "{}"; });
    std::future<const char*> f_empty = empty.get_future();

    std::packaged_task<const char*()>  a([&](){
        f_empty.wait(); return "{a}"; });
    std::future<const char*> f_a = a.get_future();

    std::packaged_task<const char*()>  b([&](){
        f_empty.wait();     return "{b}"; });
    std::future<const char*> f_b = b.get_future();

    std::packaged_task<const char*()>  c([&](){
        f_empty.wait();     return "{c}"; });
    std::future<const char*> f_c = c.get_future();

    std::packaged_task<const char*()>  ab([&](){
        f_a.wait();     f_b.wait();     return "{a,b}"; });
    std::future<const char*> f_ab = ab.get_future();

    std::packaged_task<const char*()>  ac([&](){
        f_a.wait();     f_c.wait();     return "{a,c}"; });
    std::future<const char*> f_ac = ac.get_future();

    std::packaged_task<const char*()>  bc([&](){
        f_b.wait();     f_c.wait();     return "{b,c}"; });
    std::future<const char*> f_bc = bc.get_future();

    std::packaged_task<const char*()> abc([&](){
        f_ab.wait();    f_ac.wait();    f_bc.wait();
        return "{a,b,c}"; });
    std::future<const char*> f_abc = abc.get_future();

    std::thread(std::move(abc)).detach();
    std::thread(std::move(ab)).detach();
    std::thread(std::move(ac)).detach();
    std::thread(std::move(bc)).detach();
    std::thread(std::move(a)).detach();
    std::thread(std::move(b)).detach();
    std::thread(std::move(c)).detach();
    std::thread(std::move(empty)).detach();
    f_abc.wait();
}
```

Figure 3.4 : Subset relation for {a,b,c} expressed with C++11 futures

3.2 Macro-Dataflow models

We gave a brief introduction to macro-dataflow models on section 2.1.4. Let us recap, a macro-dataflow programming model is considered to be a compromise between dataflow and imperative programming. A pure dataflow model does not suit on today's architectures, however we can have imperative tasks, which would follow dataflow semantics across but not within tasks. As macro-dataflow models explicitly declare the dependence relations between tasks, these models utilize dependence as a first-level construct.

3.2.1 Habanero C with Data-Driven Tasks and Futures

We introduced the Habanero approach to parallelism on section 2.4, through the use of `async` for task creation and `finish` for task synchronization. By introducing data-driven futures as a type, and an `await` clause for tasks to declare their dependences through a list of data-driven futures, we achieve a macro-dataflow version of the Habanero-C language.

In our former work [21], we introduced data-driven futures as a means to make dependence a first level construct analogous to I-structures [33].

Futures in the literature has been described as three-tuples, a reference to a value, the computation that resolves to that value and the awaiting tasks for that value. Commonly, when a future is asked to be resolved, the calling context is blocked and the resolving computation for the value is executed to supply the awaiting task or tasks. A blocked task's state has to be stored either by creating a continuation on-the-fly and yielding to a child task, or by blocking a thread to preserve the state and create a new thread to run children tasks to prevent a possible deadlock. Both these options introduce overheads.

Whereas, a data-driven future is a reference to a value and the list of awaiting tasks. It is two-tuple, and the resolving computation is not known at declaration time. A computation that has a handle for a data-driven future can resolve it explicitly. Not requiring futures to declare their resolving computation makes the case for a non-blocking semantics even stronger. A blocking future can be immediately resolved since the producer is known, however a data-driven future, if we provided a blocking implementation, would block till the producer computation registers itself as a producer. We guarantee non-blocking semantics to data-driven futures by requiring them to be only resolved within data-driven tasks.

A data-driven task is an explicitly parallel task with an await clause, where the await clause consists of all the data-driven futures that would be asked to be resolved within that task. This task is only scheduled when all its await clause data-driven futures are resolved. Therefore it follows dataflow semantics: the availability of input data fires the task.

Expressibility

When a user declares shared data between tasks via data-driven futures, this allows them to express true dependences. As there is a single producer for a data-driven future in an error-free program, that may be considered the source of the dependence, where all the data-driven tasks listing that data-driven future are sinks of dependences on that value. Moreover, there is no requirement for data-driven futures to abstract just data; they can also be used as *void* data-driven futures to impose an ordering, for example to prevent hazards on a shared, non single assignment datum by declaring anti-dependences or output dependences.

If you depict your dependence graph as a DAG, a data-driven future abstracts a

directed edge. Hence dependence relations that can be established are all possible graphs that can be expressed by directed edges as shown in figure 3.4 in section 3.1.

Scheduling

Data-driven futures and tasks utilize non-blocking scheduling semantics by requiring data-driven futures be accessed only within registered data-driven tasks. Common futures allow unrestricted resolution from any context, but if the resolving task has not completed, they block. This preserves the calling context at the cost of delaying the continuation that is not dependent on the future's resolution and tying up an execution unit in case it is not the one resolving the future.

When a data-driven task is declared with its await clause, a frame to contain its context gets implicitly created, just like a common async. Additionally, a list of data-driven future references gets passed to this async that serves as a synchronization frontier. Eagerly the task tries to register to the first unready data-driven future by iterating over its list, checking readiness condition. Once an unready data-driven future is reached, the task registers itself to that data-driven future and the control returns to the parent task. If all the dependences have been met at the time of creation, the task is simply passed onto the scheduler, just like a normal async would.

When a data-driven future is resolved by the producer task creating its value, the producer task grabs the list of pending tasks and iterates their synchronization frontier, as described in the paragraph above. If the pending tasks have all their dependences met, they are passed onto the scheduler, if not they linger in the heap to be picked by their following dependences' producers.

This scheduling follows the semantics of dataflow; the tasks are fired when their data becomes ready. Most parallel programming models require the data dependences

to be met at the point of task creation, burdening the programmer to structure their code accordingly, following the spirit of the imperative-language causing topological-sort argument covered in the section 3.1 above.

Work-stealing support

In past work [3], we provided a proof of concept for data-driven futures on the Java incarnation of Habanero ideas, Habanero-Java. We ported data-driven futures ideas to the C incarnation, Habanero C. The compiler support for continuation creation at fork points are designated to take await clauses into account and attach it to the frame allocated on the heap for the parallel task, as described in the scheduling section above. As also discussed above, once a task is created, its parent task eagerly iterates over the await list to find the data-driven future to which the task should register itself. The last task that satisfies the last dependence in the list of dependences spawns the task as if it has no dependences. Porting the data-driven futures to the Habanero-C infrastructure helped us achieve work-stealing support.

Safety

Shared objects across tasks when expressed through data-driven futures, do not suffer from race conditions. As a data-driven future can be resolved only once and by a single producer, its value can not be written over, preventing race-conditions. Data that is not resolved or at the state of being written, can not be observed by the consumers, since consumer's synchronization frontier iteration is succeeded by the data resolution.

Though the schedule is input and the underlying hardware dependent (and therefore undeterministic), the unfolding computation is deterministic when data-driven

futures are utilized. The orderings declared by the users through data-driven futures are obeyed and can not change. Additionally, the data computed and consumed can not change as described on the paragraph above.

On the expressiveness tab above, we likened data-driven future's to directed dependence edges. This power, however may be misused by the programmer to create cycles and cause deadlocks. Though there is no such thing as a circular true dependence, if the user mistakenly declares so, the program will deadlock [‡]. We have not provided a runtime mechanism for deadlock detection, but one can be programmed easily by tagging data-driven futures at runtime based on the ordering relations observed. These tags will follow a partial-order. Once a relation in between two tasks are introduced that violates this partial order, a deadlock is introduced.

Distributed memory implementations

We collaborated to provide a data-driven task and future implementation on the distributed memory supporting incarnation of the Habanero-C language. A distributed data-driven future is a data-driven future with a user-defined home rank. Any data-driven task registering to a distributed data-driven future sends the registration request to the home rank. Once a data-driven future is resolved, the home rank sends the value to all the registered unready tasks by sending the values to the ranks of where those tasks originated.

[‡]This is a restricted form of deadlock, where some tasks may only be *blocked* at the start because their preconditions have not been satisfied.

3.2.2 Open Community Runtime

Open Community Runtime is a collaboration involving academia and industry partners in order to set a programming interface between a parallel programming language and the underlying runtime to supporting this API. A user level API allows the user to declare unrestricted DAG parallelism and the underlying runtime executes it. The separation of concerns provided by this API allows runtime research to be conducted on separate platforms with several different objectives.

In the section above, we have revisited our discussion of data-driven futures, data-driven tasks and macro-dataflow scheduling. The future and task constructs are declared by the user through language extensions, which necessitates either a source-to-source translator or a compiler. Those in turn, introduce portability, maintenance and adoption concerns. In the library versus language decision, data-driven tasks adopted the language approach, where Open Community Runtime opted for the library approach. However, the two can come together since OCR can be used to implement higher-level language constructs.

Application Programmer Interface

Open Community Runtime, henceforth abbreviated as OCR, declares macro-dataflow parallelism using the following library calls, which will also cover the concepts utilized:

ocrTaskCreate is used to create a parallel task. This task may have dependences that would be declared via `ocrAddDependence` that the runtime would maintain and may need static data (as in function arguments) that the user should maintain. Since the user may not know the underlying implementation, it is not safe to assume any implicit ordering among tasks or the permanence of the stack variables across stack

invocations. This interface requires the user to pass the function to be executed, the function parameters and how many of them there are, and how many dependences it will eventually declare.

ocrEventCreate This function creates an *event* object, which can be used to declare dependences between tasks. The event construct is a more general version of a future, in that it does not know its producer or the value it will carry. Since events are the dependence abstraction for this model, they are single assignment, as in they can only be satisfied once.

ocrDbCreate This function is used to create a *data-block*. A data-block can be described as a contiguous chunk of memory managed by the runtime. They can be used to satisfy events, and declare data-dependences. This abstraction allows the runtime to maintain the data and provide guarantees. Any data that is not declared to be a data-block is user-managed and if used as shared data in between tasks without knowing the underlying implementation assumptions, would lead to errors. Unlike the data-driven futures discussed earlier, there is no single-assignment property guarantee for data-blocks. Since the dependence aspect is separated into an event and the data it carries, the single assignment property remains on the event, and not the data-block.

ocrEventSatisfy As events can be used to build a dependence graph, this interface informs the runtime that the dependence has been satisfied. If the dependence is a data-dependence, this function declares what data is flowing through this dependence via data-blocks. If the dependence satisfied is not a data-dependence, the event may be satisfied with any object or a sentinel value.

ocrAddDependence is how a task declares that it is a sink of a dependence that is passed as an argument. This serves the same purpose of the `await` clause mentioned in the previous section. The user is required to enlist all the shared data across tasks to be declared as a dependence to guarantee safe access through the synchronization provided by dependences.

ocrScheduleEDT is the user declaring that the listing of the dependences for a particular task is over and now the runtime can take control over it. As the number of dependences are known at creation time and the `ocrAddDependence` can count the number of dependences declared, this interface could have been avoided. In the following versions of OCR, this function is deprecated.

Runtime Library

Any OCR runtime library that implements the functions described above, can be labeled an OCR library, thus there is not '*the*' OCR library. For our explorations, we have used an OCR runtime library implementation that is heavily influenced by Habanero-C [22] that implements the user interface from OCR version 0.7 [34].

A detailed run-down of the runtime programmer's API is beyond the scope of this document, however, we will provide a quick introduction below to lead into the implementation we utilized.

Runtime Programmer Interface

The runtime library is implemented in C, which does not natively support modern constructs like abstract classes, interfaces or inheritance. Therefore we instead have provided a poor man's version for these constructs by providing base structures for

modules that we anticipate the runtime implementer would have to extend, with function pointer tables mimicking a virtual function table. The modules are for data-blocks, events, task pools, workers, executors (abstracting the underlying execution unit), schedulers and *policy-domains* (abstracting a mini-runtime, for hierarchical runtimes).

Habanero-like runtime for OCR v0.7

We implemented an OCR version 0.7 library replicating the scheduler by the data-driven scheduling implementation covered in section 3.2.1. Regarding other modules, like data-blocks, workers, executors and policy-domains, we implemented bare necessities. Data-blocks are implemented as wrappers for contiguous memory on a shared memory machine that does not move or get tracked by the runtime. Workers execute a loop of popping, work-stealing when pop is failing, executing extracted work, just like Habanero workers. Executors are abstracting the underlying cores with an attached PThreads instantiations. Policy models are not utilized as we have not needed explicit hierarchies for our observations.

3.3 Results

3.3.1 Methods and environments

We use Open Community Runtime version 0.7 as a starting point for our parallel runtime implementation. We use N workers for a machine with N execution units, with a total of N double ended last-in/first-out queues (deques) for ready tasks, with one-to-one correspondence between execution units, workers and deques. As introduced in section 2.4, the work-stealing implementation utilized is local task extraction

of the most recently created task in the local doubly-ended queue, once local work is depleted, the default victim selection is random for stealing and extracting the oldest task for stealing from the victim.

We tested our runtime on two different setups:

A Xeon machine This machine has two-sockets of Intel E5-2699 v3 running at 2.30GHz. Each socket is an 18-core machine with an extra hyperthread per core Haswell architecture. The total amount of memory available to the machine is 128GBs. We will not be utilizing the hyperthreads, so the runtime treats this architecture as a flat 36-core machine.

We use Intel Parallel Studio XE 2015 Update 2 Composer Edition for Linux, which features `icc` version 15.0.2 as the C compiler, Intel MKL version 11.2 update 2 as the math library, and Intel TBB version 4.3 update 3 whose scalable memory allocator we link with to replace `malloc` calls.

A XeonPhi machine This machine has a XeonPhi coprocessor as an accelerator that we use in native mode as a standalone machine, rather than a host-card pair. This card features 60+1 inorder x86 cores with 4 hyperthreads per core, each running at 1100Mhz. Total memory available to the machine is 8GB.

We use Intel Parallel Studio XE 2013 Service Pack 1, which features `icc` version 14.0.1 as the C compiler for cross-compiling to the XeonPhi coprocessor, Intel MKL version 11.1 Update 1 as the math library and Intel TBB version 4.2 update 1 whose scalable memory allocator we link to replace `malloc` calls.

3.3.2 An inefficient Fibonacci

We addressed how divide-and-conquer algorithms and languages that express series-parallel task graph match, and competitive performance can be achieved even with this ease of expression. We want to show that divide-and-conquer problems can also be easily expressed through declaring this restricted subset of possible directed acyclic graph and also provide competitive performance.

We will use an inefficient (non dynamic-programming) approach to calculating the Nth element of the Fibonacci sequence. The Fibonacci sequence can be described inductively as follows:

$$fibonacci(n) = \begin{cases} fibonacci(n-1) + fibonacci(n-2) & n \geq 2 \\ 1 & n = 1 \\ 0 & n = 0 \end{cases}$$

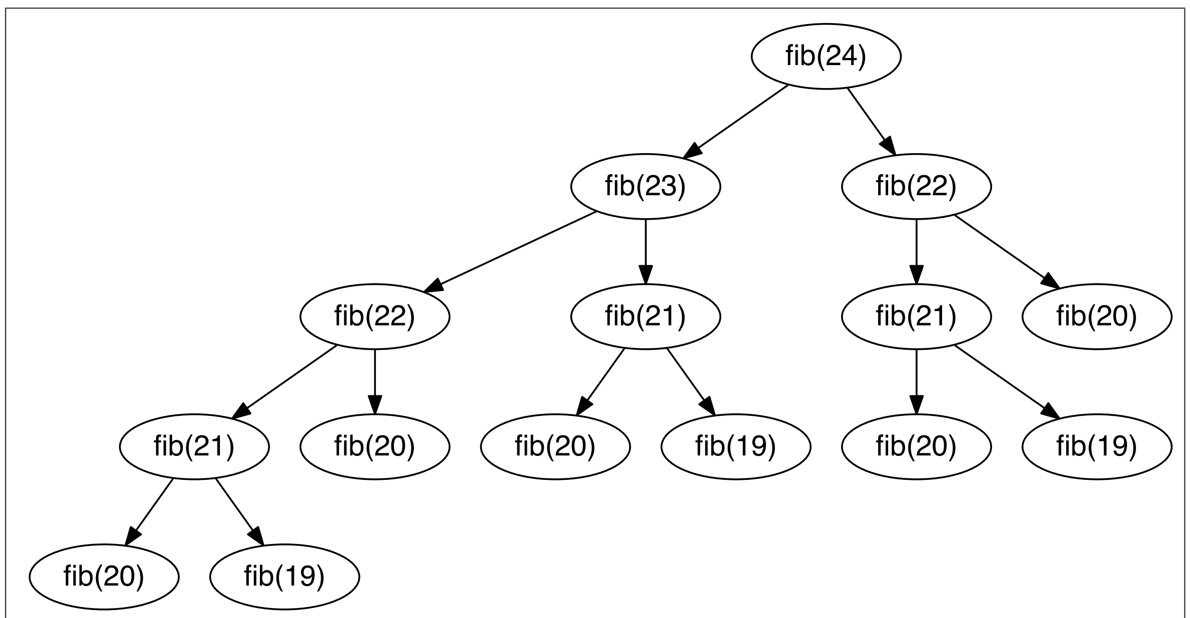


Figure 3.5 : Task graph for calculating $fib(24)$ with cut-off 20

On figure 3.5, we can observe the divide-and-conquer aspect and the almost balanced nature of the algorithm. As we argued before, there also is a dual join edge for every node for synchronization we chose not to depict.

Our implementation of the non dynamic-programming version of Fibonacci features a cut-off value, at which the program stops spawning child tasks and calls a serial version of the non dynamic-programming version of Fibonacci in order to introduce an artificial granularity to leaf computation tasks. This artificial granularity is a means to hide the overhead of task creation, scheduling and work-stealing. Cilk language and runtime fares better at no or very low cutoff values compared to OCR, as Cilk is a language approach which creates continuations and *fast* paths while OCR can not. So with a very low cutoff value OCR fares worse compared to Cilk.

For collecting our OCR results, we provide a text file mapping worker threads to which cores, in order to prevent thread migration or a possible across socket core selection. Therefore for the sake of fairness the Cilk version is run with an explicit restriction of which subset of cores the program can run on using `taskset`.

$\#cores$	1	2	4	8	12	16	18	36
$time(s)$	68.425	35.406	19.149	10.868	7.327	5.498	4.933	2.512
$speedup$		1.933	3.573	6.296	9.339	12.445	13.870	27.240

Figure 3.6 : $fib(50)$ with cut-off 25 results for the Xeon Machine using Intel CilkPlus

$\#cores$	1	2	4	8	12	16	18	36
$time(s)$	68.601	35.182	19.156	10.886	7.363	5.522	4.986	2.511
$speedup$		1.950	3.581	6.302	9.317	12.424	13.758	27.322

Figure 3.7 : $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7

Given a cutoff value, OCR and Cilk performs head to head, as can be seen in

figure 3.6 and figure 3.7. We calculate the 50th number on the Fibonacci sequence and the leaf computation nodes calculate the 25th number. Scaling is not perfectly linear on this 2 socket (a total of 36 core) machine because of the inherent overheads and also the imperfect balance between left and right children of the tree. As we know, the ratio between consecutive Fibonacci numbers converge to the *golden ratio*(~1.618), which means the left child of root task is ~1.618 the size of the right child.

$\#cores$	1	2	4	8	16	32	48	60
$time(s)$	78.646	39.329	19.685	10.073	5.045	2.617	1.756	1.442
$speedup$		2.000	3.995	7.807	15.590	30.049	44.793	54.536

Figure 3.8 : $fib(45)$ with cut-off 25 results for the XeonPhi Machine using Intel CilkPlus

$\#cores$	1	2	4	8	16	32	48	60
$time(s)$	78.668	39.526	19.699	9.906	4.945	2.488	1.669	1.342
$speedup$		1.990	3.994	7.942	15.908	31.614	47.146	58.604

Figure 3.9 : $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7

The results for our XeonPhi machine can be seen on figures 3.8 and 3.9. We observe almost linear scaling for OCR, where Cilk scaling tapers off faster than OCR. However we should note that the environment for the XeonPhi card we utilize does not feature `taskset`, so the Cilk results can be attributed to unpinned threads, where OCR pins individual PThreads to XeonPhi execution engines one-by-one.

3.3.3 Smith-Waterman/Needleman-Wunsch sequence-alignment

This benchmark is an algorithm to align two strings by attributing scores to a match at a given site based on the matches at possible previous sites that are one fewer

in length, hence it is a dynamic algorithm. Building the tabular structure that remembers match scores for substrings of all sizes is of order $O(N * M)$ in space and computation, for strings of size M and N. If the scores cached are positive and 'forgotten' below a certain threshold, it is a *local* sequence alignment and is called the Smith-Waterman algorithm. If scores cached are not forgotten and a global sequence alignment is sought after with the sink node of the task graph's alignment score is the match score, it is called Needleman-Wunsch algorithm. We will be using the Needleman-Wunsch variation for our experiments.

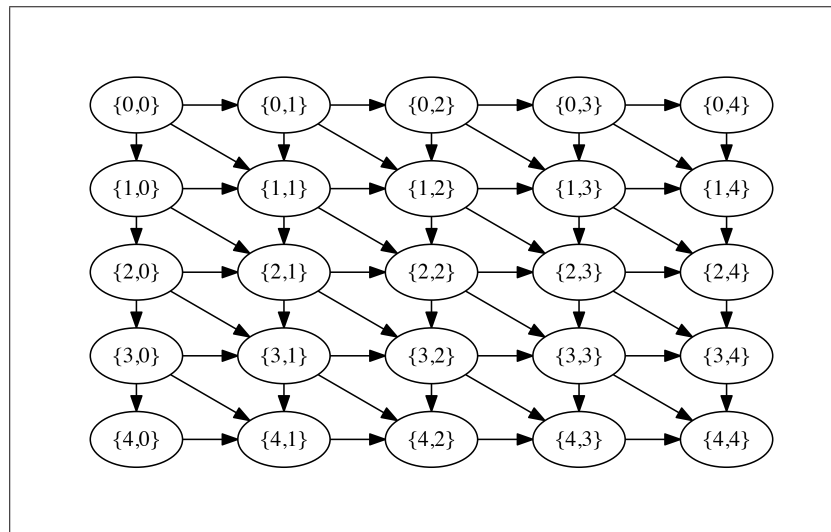


Figure 3.10 : Dependence graph for a 4 by 4 tiled string matching

This benchmark is used in bioinformatics field to match amino-acid sequences of proteins, or nucleotide sequences of sites for local and global alignment to trace evolutionary paths, homology and etc. We see this benchmark frequently in the parallelism, high performance computing literature to showcase non-series-parallel graphs, and are known colloquially as the diamond graph. We can see the dependence graph of a string matching benchmark of 2 strings sized 4, or 2 strings with square

tiles of quarter the size of the original string on figure 3.10. The first row and column is to keep track of the alignment with a gap instead of a nucleotide at the beginning of the first or the second string, hence the need for a fifth row and column.

On figure 3.11, we can see the whole tabular structure maintaining the scores for the alignment of two strings: `TACTG` and `CTAGTCG`. The dependence graph for this computation is superimposed on the tabular structure and is depicted by directional arrows. The score for a node on this tabular structure is calculated by deciding the maximum of three possible predecessor node's values and how much a match from that path would cost. For example the value -3, at the intersection of second row and the third column, is calculated by choosing in between: the diagonal path (the first string's `A` should be matched with the first `C` of the second string, $-1 + -4 = 5$), the horizontal path (matching first string's `A` with a gap, $-2 + -1 = -3$) or the vertical path (matching second string's first `C` with a gap, $-2 + -1 = -3$). The winning path for each node is depicted by a dotted line instead of a solid one. The global match, and hence the alignment, is depicted by red dotted lines tracking the path from the right bottom corner to top left corner.

In order to introduce some granularity to the leaf computation tasks, we tile the strings, so that a task calculates the values of a tile rather than a single entry. We performed a sweep for tile sizes to choose which tile size achieve the highest operations per second, and the results reported below use those tile sizes.

An exception for the results collected on the XeonPhi machine for this benchmark is that the best number operations per second is achieved when neither the application nor the benchmark itself is linked with Intel TBB, and when the libraries including the runtime are linked statically.

For both of the figures above, we observe scaling that is not perfectly linear.

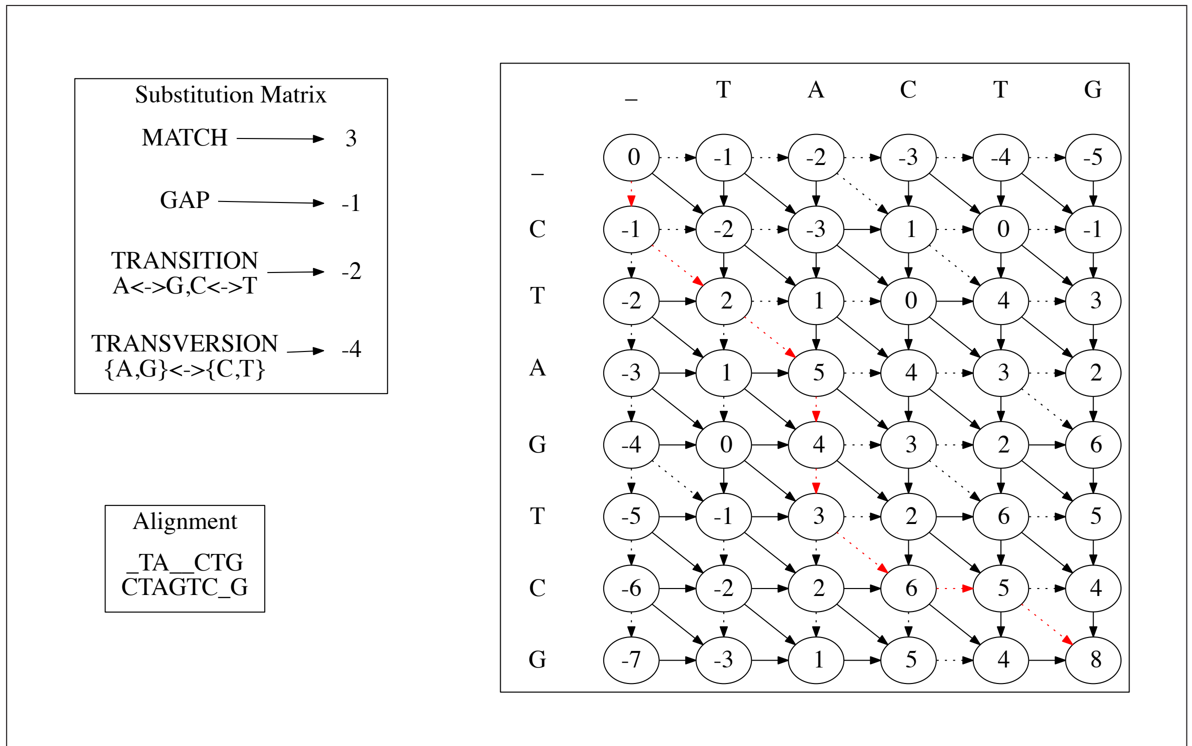


Figure 3.11 : Score graph for a 5 by 7 string matching

$\#cores$	1	2	4	8	12	16	18	36
$time(s)$	39.688	20.354	11.135	6.335	4.318	2.910	2.152	1.523
$speedup$		1.950	3.564	6.265	9.191	13.640	18.440	26.059

Figure 3.12 : Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7

$\#cores$	1	2	4	8	16	32	48	60
$time(s)$	135.327	67.586	34.020	17.127	8.682	4.476	3.225	2.740
$speedup$		2.002	3.978	7.901	15.587	30.236	41.964	49.395

Figure 3.13 : Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7

If we look back at figure 3.10, we can see that the beginning and the end of the computation do not provide sufficient slackness. For a better visualization, one can

skew the graph as in loop-skewing and conclude the parallelism is one the diagonal that can be represented with $f(x, y) \rightarrow x - y + C = 0$. Our macro-dataflow model allows unstructured diagonals to be expressed that does not need static diagonal extractions of parallelism. However, even in our model the unstructured diagonals shrink to sizes smaller than the number of execution units available at the beginning and end of the benchmark preventing linear scaling.

3.3.4 Cholesky decomposition

Given a symmetric, positive definite matrix A, cholesky decomposition calculates a lower triangular matrix L such that $A = LL^T$ and can be considered a special case of LU factorization where the upper triangular matrix is the lower triangular matrix's conjugate transpose. The computational complexity of the calculation is $O(n^3)$ and for a serial, in-place implementation the memory footprint is $O(n^2)$. Our parallel implementation through array-expansion, exposes the iteration-space as the third dimension and gets rid of the antidependences to expose further parallelism, which increases the memory footprint to $O(n^3)$.

The dependence graph of a 5 by 5 blocked cholesky factorization is depicted on figure 3.14 with tasks annotated with the LAPACK routines applied on said tiles. On a given iteration the top-left most tile has a sequential cholesky(`dpotrf`) applied, where that result enables a column of triangular solves(`dtrsm`) below it. A trisolve indexed i feeds data to triangular(`dsyrk`) or square(`dgemm`) matrix multiplications on row or column i. The resulting matrices of these matrix multiplications feeds in to the next iterations domain, depicted as vertical arrows crossing the iteration boundary on the figure.

As it can be seen on the figure, the dependence graph is an unstructured directed

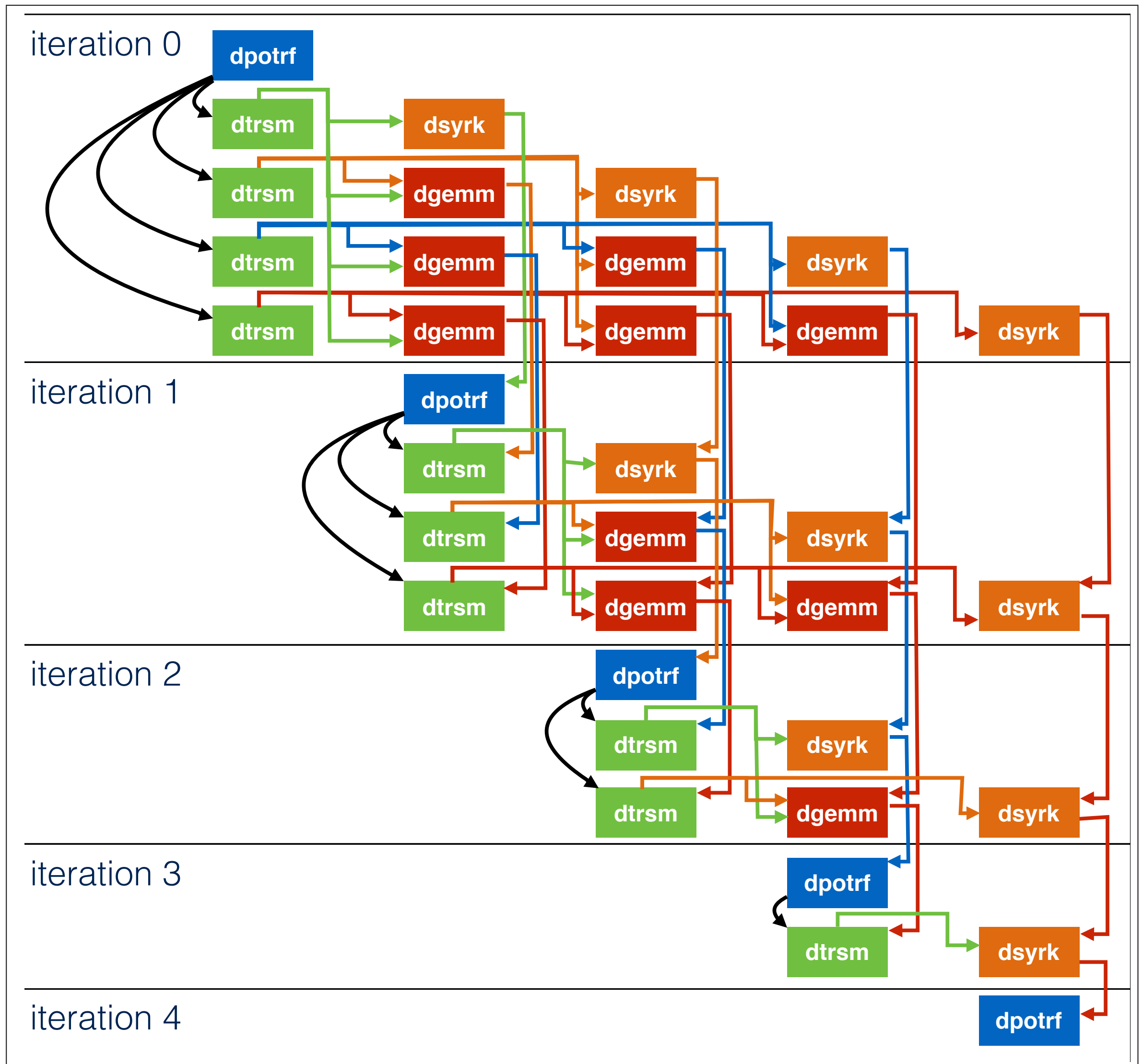


Figure 3.14 : Dependence graph for a 5 by 5 tile cholesky factorization

acyclic graph, and does not remotely resemble a series-parallel computation. Hence it is a motivating example for our macro-dataflow model.

We have implemented the benchmark with tasks serving as a wrapper to serial Intel Math Kernel Library(MKL) calls. Since MKL library calls are destructive writes to their input data, we use events to synchronize these writes on to the same data-

block. Additionally, to provide coarser granularity into the tasks, we use a blocked version of the cholesky decomposition, where the tile size is a user provided runtime parameter. As auto-tuning and providing performance models for different architectures are not within the scope of this work, we do a tile sweep to calculate the tile size that gives the highest floating point operations per second(flops).

$\#cores$	1	2	4	8	12	16	18	36
$time(s)$	16.579	8.409	4.245	2.374	1.625	1.339	1.248	0.665
$speedup$		1.972	3.906	6.984	10.201	12.386	13.287	24.947

Figure 3.15 : Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine

$\#cores$	1	2	4	8	12	16	18	36
$time(s)$	13.870	6.845	3.476	1.943	1.379	1.203	1.065	0.751
$speedup$		2.026	3.991	7.138	10.059	11.526	13.017	18.465

Figure 3.16 : Cholesky decomposition results for a 12K by 12K matrix using Intel MKL for the Xeon machine

Figures 3.15 and 3.16 show that we can surpass a hand-tuned library in floating points per second throughput on our Xeon setup. We should still note we used serial MKL kernels to describe the kernel computations and used our runtime to provide the parallelism where the figure 3.16 results show what a single parallel MKL call to solve cholesky would do. Additionally, we had to do a tile-sweep to find the best throughput providing tile size (192 for this case), where parallel MKL did not need that runtime parameter. Likely, the scaling results for parallel MKL can also be partially attributed to the library using adaptive tiling. We can observe that the base case for OCR is slower than MKLs and a tile size different than 192 providing the best throughput for a single core execution would make our scaling results worse than the 25× shown on the figure.

Additionally, we should remind that we do not utilize hyperthreading and use only 36 cores pinned to individual cores for our OCR executions. The Xeon machine provides 36 additional hyperthreads, which MKL can utilize further to achieve even higher throughput, but we will postpone covering this aspect for the upcoming chapters.

#cores	1	2	4	8	16	32	48	60
time(s)	27.845	13.952	7.039	3.568	1.833	0.972	0.718	0.642
speedup		1.996	3.956	7.804	15.191	28.647	38.781	43.372

Figure 3.17 : Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine

#cores	1	2	4	8	16	32	48	60
time(s)	11.635	11.634	3.932	2.161	1.105	0.581	0.425	0.347
speedup		1.000	2.959	5.384	10.527	20.025	27.394	33.554

Figure 3.18 : Cholesky decomposition results for a 6K by 6K matrix using Intel MKL for the XeonPhi machine

Unlike the result for the Xeon architecture, for the XeonPhi machine our OCR results could not outperform Intel MKL's hand tuned for XeonPhi parallel cholesky decomposition. As we argued above for Xeon, one of the problems can be attributed to our choice of non-adaptive tiling. Though we achieve a respectable scaling for a fixed tile size, Intel MKL can achieve almost twice the throughput for a tile size it tunes.

We observed OCR scaling better on Xeon to catch up and pass MKL even though the base case is OCR is worse. That still is the case our XeonPhi results. However, our base case is 2.5× slower for our best tile size, 96, for maximum throughput on 60 threads. Even when a favorable tile size for a single thread execution is picked the base case is 18.278 seconds, 1.57× slower.

Another observation is the effect of Intel TBB being linked with the multi-threaded MKL implementation. For our OCR variations of the code, the application being linked with TBB do not have much of an effect, however multi-threaded MKL being linked with TBB on XeonPhi provides double the throughput for cholesky. We should also note the suggested link flags for parallel Intel MKL does not necessarily suggest using TBB.

Lastly, as we covered on our Xeon discussion above, MKL can utilize hyperthreads. As we do not have access to `taskset` on our machine we can not make sure how many hyperthreads MKL is using and which subset of cores it utilizes. However, for OCR we still use PThreads pinned to each core without utilizing hyperthreading. Just like Xeon, the XeonPhi results can be further by MKL in throughput by allowing it to use more threads. With OCR, we observe using flat work-stealing on an oversubscribed machine degrades performance as opposed to the single thread per core case.

Chapter 4

Efficient work-stealing in Event-Driven Runtime Systems

4.1 Introduction

It is interesting to observe that nested fork-join parallel models are more popular than macro-dataflow parallel models, despite the factor that the former is more restricted than the latter. One reason nested fork/join models are popular is because of the top to bottom design and implementation choices that go along with the model. We have briefly covered the work-stealing runtimes in section 2.1.3 that are used by these models. Let us elaborate further on this discussion.

Popular work-stealing runtimes for nested fork-join parallelism [35, 25] have utilized lazy task creation [2] in order to avoid the runtime being swamped by eagerly created tasks. Lazy task creation can be interpreted as a sequential-by-default depth-first exploration of the task tree, in which multiprocessor thread scheduling is achieved by taking tasks from the unexplored list in the depth-first traversal. Since this depth-first traversal has a much smaller frontier than an alternative traversal (e.g. breadth-first), the number of tasks available to the scheduler is tightly bound. This approach to thread scheduling of tasks is a flavor of work stealing, since idle threads extract tasks from busy threads' yet unexplored paths.

For example, in Multilisp, one of the earliest implementations of work-stealing schedulers, the stealing heuristic employed is to steal the *oldest task* from the victim,

just like the Cilk [25] implementation that followed. An *oldest* task would be the task that would be the first task to make it to the backtracking list of a depth-first traversal by a worker, and therefore the last one to be utilized for further exploration.

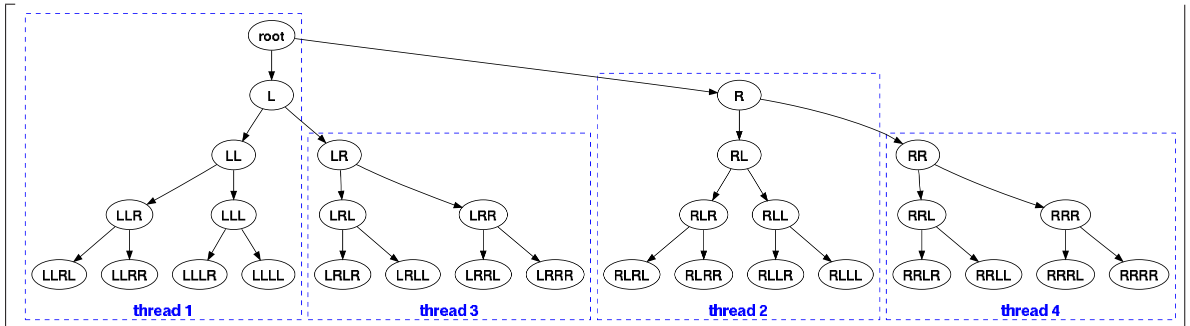


Figure 4.1 : Possible decomposition and mapping of a divide and conquer problem, figure credit [2]

Let us look at a snapshot of a work-stealing runtime with lazy task creation and *oldest task* stealing. We can see in figure 4.1 that $thread_2$ stole $task_R$ from $thread_1$'s first task that is on that thread's depth-first traversal backtrack list. Then $thread_3$ stole a $task_{LR}$ from $thread_1$, which is the new *oldest* task after $task_R$'s extraction. Lastly $thread_4$ stole $task_{RR}$ from $thread_2$ to achieve the depicted decomposition and mapping.

The task-graph on figure 4.1 does not reflect the *join* aspect of the graph. As nested fork/join models require the children to synchronize with the parent, a complete depiction would include a mirroring of the graph on the plane created by the leaf nodes. That graph would have the aforementioned series-parallel graph property. For simplicity, we will depict task graphs only till their leaf nodes and not include their dual join edges.

In a series-parallel task graph, stealing a task that was put aside to be explored provides the source node of another series-parallel graph. By definition series-parallel

graphs are recursive structures, and if a task is put aside to be explored, it can only come from a parallel composition of more than one series-parallel graphs.

For a divide-and-conquer algorithm with a cutoff, like the one depicted in figure 4.1, stealing the oldest task from a thread provides the coarsest grained series-parallel graph that thread has to offer. As work starts dissipating from a single source node, stealing would build a binary reduction tree of splitting and mapping subsets of the task graph, as seen in figure 4.1. Cilk or Mul-T [2] implementations also depend on this property of probabilistic work-stealing.

Figure 4.2 is a pathological case of a series-parallel task graph for stealing oldest tasks, where the graph is recursively *left-skewed* and there is not sufficient parallelism to make up for the runtime overheads introduced. The oldest tasks for any series-parallel that is being explored or stolen has the smallest grain size where the youngest tasks have the coarsest grain. That is why most models perform their best at high parallel *slackness*. *Slackness* is amount of ideal parallelism (in layman's terms the ratio of *width* to *depth* of the task graph).

The increased granularity of steals reduces the number of steal attempts and improves performance as steal operations introduce more runtime overhead, contention and increase idle time.

4.2 Granularity

We use work stealing runtimes for our aforementioned macro-dataflow models for dynamic scheduling and load-balance. However, since task graphs that may be expressed by macro-dataflow models are more general than series-parallel graphs, the heuristics we discussed in section 4.1 for work-stealing runtimes do not necessarily hold anymore.

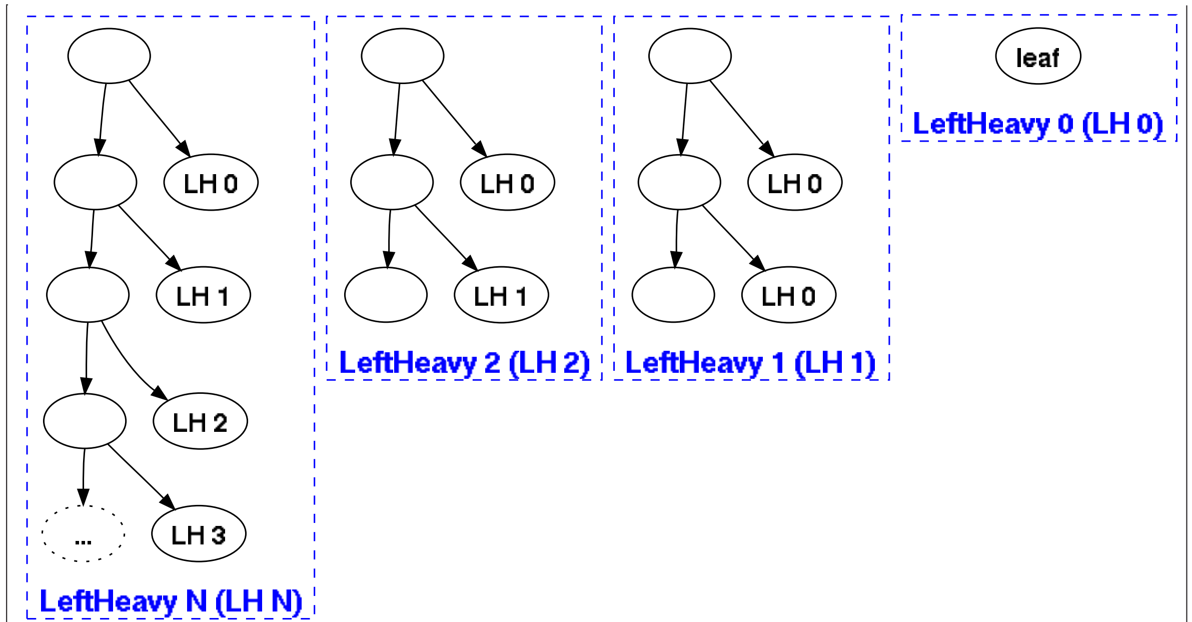


Figure 4.2 : A pathological task graph that is recursively left-skewed

In a series-parallel graph, stealing one task from the victim gives us the source node of another series-parallel graph. If you ignore the join edge symmetry, a series-parallel graph is tree. So a steal returns the root task of a tree of tasks. Since that root task enables all the descendent tasks, if none of them is stolen from the thief, the whole tree is executed on that thief. Therefore stealing one task is analogous to stealing a subtree of a task tree.

In section 4.1, we discussed the motivation for stealing the oldest task for series-parallel graphs with divide-and-conquer algorithms being the best fit. Let us assume that the base case of a divide-and-conquer algorithm has a unit computational cost of c. A thread executing a leaf task, the base case computation, has n ready tasks for execution (or victims for stealing), where the ith youngest task, for $i \geq 1$, costs $c * (2^i - 1)$ cumulatively, since tasks are roots of a task tree of computation. If we observe figure 4.3 as a sample, we see that a thread's depth first traversal of an

available task ($task_A$) executed $task_A$, $task_B$, $task_C$ and $task_D$ and pushed $task_I$, $task_F$ and $task_E$ on to its queue of tasks. The subtrees rooted by $task_E$, $task_F$ and $task_I$ cumulatively cost $c, 3c$ and $7c$ respectively. The total amount of work for these n tasks is $\sum_{i=1}^{n} c * (2^i - 1) = c * (2^{n+1} - 1 - n)$. The *oldest* task, the first in line to be stolen, is of cumulative cost $c * (2^n - 1)$. So stealing the oldest task is roughly stealing *half* the amount of total work, since $c * (2^n - 1) \approx 2^{-1} * c * (2^{n+1} - 1 - n)$ as n grows larger.

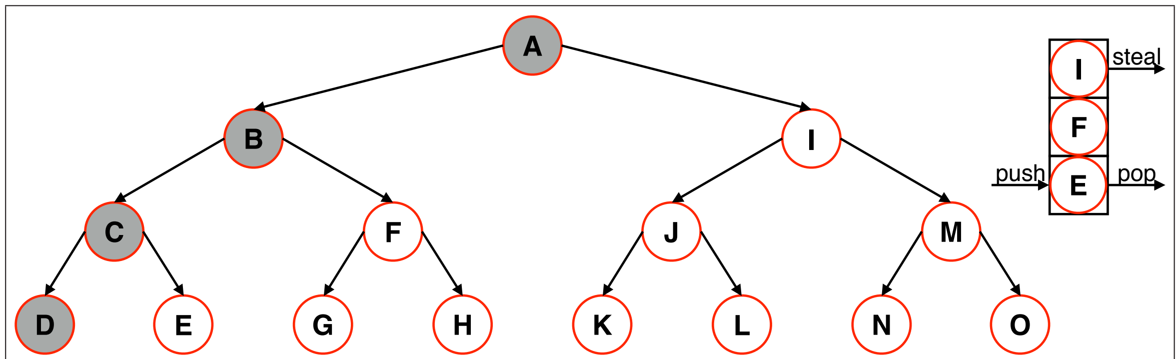


Figure 4.3 : Task graph of a divide-and-conquer application run under *work-first* policy

Our assumption to conclude *steal-half* is a major one, where we know the structure of the dynamic task graph statically by restricting the problem to balanced divide-and-conquer algorithms. How we concluded the *half* is through assuming unit task sizes, and the number of tasks a stolen task would enable as a root (source node). If we want to generalize, we may require the user to annotate tasks with how many descendants they may enable. For example, in figure 4.2, we have given an example of an 'unbalanced' series-parallel graph. However, if the user knows how many descendants every task has when the task is created, and annotates the task accordingly, the runtime still can pursue a *steal-half* heuristic by a bin packing approximation.

In contrast, stealing a single task from an arbitrary DAG may not result in getting a root of a tree of tasks. Pathologically, a stolen task may not *dominate* * any of its descendant tasks. Let us observe figure 4.4. If a $task_A$ is stolen with successors $task_B$ and $task_C$, and if $task_B$ and $task_C$ each depends also on $task_D$ and $task_E$ respectively where $task_D$ and $task_E$ are not $task_A$'s ancestors, based on a particular schedule $task_D$ and $task_E$ may not yet be available. In that case $task_A$ would not lead to any new descendant computations at all, and would eventually lead to another steal attempt.

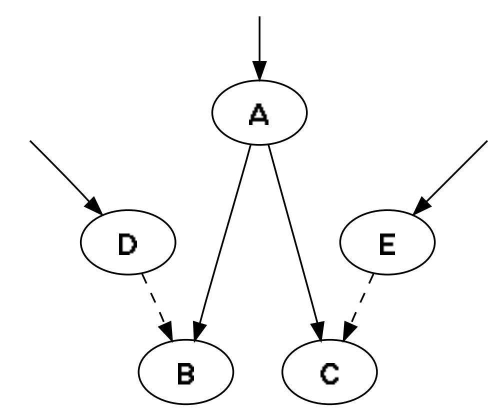


Figure 4.4 : Snapshot of a stolen $task_A$ and its immediate successors $task_B$ and $task_C$

If we wish to translate the implicit *steal half the work* policy for nested fork/join work-stealing models to macro-dataflow work-stealing in order to minimize steal attempts, we need to take further information into account as the property does not hold anymore and we need an explicit steal-half as in [36, 37]. For graph algorithms, granularity optimization to achieve half through batching can be employed [38].

*A $node_n$ *dominates* a $node_m$, if all the paths from the source of the graph to $node_m$ passes through $node_n$.

Firstly, we would need annotations, just like unbalanced graphs would have needed for nested fork/join models. For arbitrary DAG task graphs, one can still annotate or calculate the number of dominated descendants for a task. However, the number of descendants a task may lead to is schedule dependent. A task can lead to descendant tasks only if it satisfies their respectively last unsatisfied dependence, so dominance relation is a function of a runtime schedule. In contrast, for nested fork/join models, since tasks are roots of trees of tasks, the values can be computed bottom up or can be determined statically by counting the tasks they dominate.

Since we can not calculate half the tasks on a dynamically unfolding task-graph with schedule dependent number of descendants, we restrict our heuristic to static assumptions, just like nested fork/join models. We explore two extremes of the spectrum on the number of descendant tasks, one pessimistic and one optimistic. A pessimistic heuristic assumes a task can only lead to a number of tasks it statically dominates. On the contrary, an optimistic heuristic assumes all the descendants will have all their other dependences satisfied by the schedule and a task can lead to all its descendants.

We also explore further simplifications of the model, where the number of descendants are not calculated or annotated and every task is treated uniformly pessimistically, as they do not lead to any work which implicitly treats every task as a leaf.

4.3 Successor task heuristics

Stealing half the tasks have been proposed to address granularity in other models, but the goal is to steal half the amount of *work* [39]. On generic directed acyclic task graphs, we can not statically decide the number of successors a task has, since

a descendant task on a join node may have predecessors that may not be executed by the time the join node is visited. Predecessor-successor relation becomes schedule dependent on join nodes, as we discussed above on figure 4.4. In order to address the granularity concerns arising from adopting non series-parallel task graphs, we propose two heuristics on counting the number of successor tasks for a given task to be used as annotations to guide granularity aware work-stealing.

4.3.1 Pessimistic descendance

Given tasks $task_A$ and $task_B$, where $task_B$ is a predecessor to $task_A$ and also dominates $task_A$ on the task graph, we label $task_A$ a pessimistically descendant task of $task_B$. A node $task_B$ dominates a node $task_A$, if $task_B$ is on every path from a root task to $task_A$. If a task is annotated for granularity with the cardinality of its set of dominator tasks, it has the number of tasks it leads to in the worst possible schedule: a schedule where no schedule dependent descendance is materialized. Hence we label this heuristic as pessimistic.

These annotations can be calculated for a task in linear time by the programmer ahead of time since the all legal task graphs are reducible, assuming that the structure of the graph can be known statically. An iterative reverse postorder traversal on the task graph until convergence in constant number of steps would give the dominance sets, and therefore that set's size.

Using this heuristic on series-parallel graphs does not lead to inaccurate descendant counts. As we argued before, on series-parallel task graphs join edges just for synchronization purposes may be eliminated which would leave a tree of tasks. The special case of directed acyclic graphs, trees, have the dominance tree which is identical to themselves. Hence the pessimistic descendance annotations for trees reflect the

correct number of descendants. If a *steal half the work* heuristic for a work-stealing runtime supporting macro-dataflow is implemented using these annotations, problems that have a series-parallel task graph (well-balanced divide and conquer algorithms) would perform as if they were expressed and executed in a nested fork/join model using a default work-stealing approach. This abides by the motto: 'first do no harm'.

We argue above that the motivation for macro-dataflow parallelism is the prevention of overconstricting arbitrary task graphs to a structured graph. Therefore one expects the less series-parallel a task graph is, the better fit it is for macro-dataflow models. However, the less structured the graphs are, the fewer dominance relations they have. The lack of dominance relation between tasks leads to pessimistic descendance providing much smaller numbers for granularity annotations of tasks and may regress to almost every task treated as a leaf task that may not lead to any other computation. As this is the default behaviour, using pessimistic descendance has a smaller impact on graphs that are further away from series-parallel graphs. For those graphs, performance may be hampered by not being able to utilize granularity aware scheduling because of how unaggressive this heuristic is. We will see how pessimistic descendance relation regresses to steal half the number of tasks from a ready work queue on the discussion of our case studies below.

4.3.2 Optimistic descendance

Contrary to pessimistic descendance, optimistic descendance heuristic assumes that every task enables all its descendant tasks, hence the name optimistic, as it assumes a particular schedule that allows this assumption. Though this assumption may be an invalid by being overly optimistic, where no such schedule may exist.

These annotations can also be calculated in linear time by the programmer ahead

of time with the same caveat that the graph can be known statically. A traversal of the topologically sorted task graph in reverse where tasks transitively accumulate the descendant counts of their predecessors provides the values for annotation in linear time.

As pessimistic descendance, this heuristic also converges to the correct descendance counts for problems with series-parallel task graphs. The leaf nodes update the deepest inner nodes of the tree with leaf counts, and the inner nodes recursively propagate all the way to the root, annotating it with the total number of nodes on the task graph. So just like pessimistic descendance, one should observe performance as if these programs are expressed in a nested fork-join model with steal oldest policy with random victim selection.

We should note that the possible abundance of join nodes on task graphs, which is our motivation for macro dataflow models, introduces complexities to this heuristic as well. A node with more than one predecessor, where those predecessors do not have a dominance relation, is counted more than once, since that node can be optimistically enabled by each of those predecessors. No matter how many predecessor a task has, it is only executed once, making these optimistic annotations more accurate for some nodes over others based on a particular schedule. On our discussion of the cholesky decomposition case study below, we observe that even though the complexity of the algorithm is $O(n^3)$, the root task is annotated with values of $O(4^n)$ rather than $O(n^3)$, as it counts possible descendant tasks many times over through separate paths.

4.4 Task queues

In section 4.2, we talked about how the *age* of a task is reflective of its granularity, using the example on figure 4.3. As double-ended last-in first-out queues, deques, are

the default data structure for work stealing, one can observe that there is an implicit prioritization of the most granular tasks for stealing and the least granular tasks for local thread's execution. We argue in the previous sections that these assumptions do not hold for arbitrary task graphs. Below we address how the choice of data structures impacts granularity-aware work stealing.

4.4.1 Deques

Prior work-stealing algorithms on series-parallel programming models utilize deques, as they prioritize most granular tasks for stealing which leads less frequent steals and also to the simplification of the data structure which in turn leads to less contention between the victim and the thief.

We mentioned in section 4.3 that if one chooses to employ *steal half of the total work from victim* policy for arbitrary task graphs, they have to estimate(or annotate) costs for tasks, since the number of successors one task may lead to (and therefore its cumulative cost) is schedule dependent. Given that tasks are annotated with the number of tasks they may lead, one can calculate an *approximate* set of half the available work. A deque of tasks in this case is an unprioritized set of tasks with granularity estimates. We still can employ separate ends for local (pop) and remote (steal) extraction to ease contention and the implementation for synchronization.

We emphasized *approximately* above, since an accurate half (of possibly inaccurate granularity estimates) of the total work is an NP-complete problem to solve because it can be reduced to the partitioning problem. Solving that problem is not likely to be amortized by fewer steals, so we employ a greedy heuristic to calculate an approximate half by iterating through the steal end of the deque and steal tasks whose total cost (of all the cumulative costs) passes the half of the estimated work on a deque. The

estimated work on a deque can easily be a state of the data structure by adding the estimated cost of every pushed task and subtracting the cost of every extracted task.

Both descendance heuristics we mentioned above give the accurate number for granularity annotations, if the input problem has a series-parallel task graph. If we use the heuristic for stealing approximately half the work by accumulating victim tasks' costs using deques as described, we converge to the default work stealing algorithm of stealing a single task from the steal end, still abiding by: 'first do no harm'.

4.4.2 Prioritized Data Structures

The ordering provided by deques is a ranking from youngest to oldest task for local extraction and the other way around for remote extraction. Though the age of the task on a ready task queue has granularity implications on series-parallel task graphs, we covered above that this coupling (and therefore implicit granularity ordering) does not hold for more general task graphs.

In section 4.4.1, we utilized deques to coarsen the granularity of stealing using annotations and steal approximately half the work heuristics to reduce the number of steals and improve scalibity for general task graphs. Despite addressing the coarsening of the set of stolen tasks, we did not address the ranking of tasks for granularity, as deques achieve implicitly in series-parallel task graphs. Annotations for cumulative number of descendant tasks serve as a natural priority for granularity ranking, which we already anticipate to be provided. If we explicitly rank tasks increasing by their descendant task count annotations, we achieve ranking in estimated granularity of tasks.

The ranking of tasks percolates the least granular tasks for the local thread to execute and more granular tasks to be stolen, where the granularity ordering may

be arbitrary based solely on push order on deques. Additionally, the traversal to steal cumulatively half terminates quicker by exploring coarsest grain tasks first and therefore reduce contention on task queues.

Another benefit to store explicitly ordered tasks by granularity is that it already pays the sorting cost of employing the Karmarkar-Karp algorithm to better partition the set of tasks into two approximate sets in cost than the aforementioned greedy algorithm.

Total versus partial orders

Though we argue for the intrinsic values of prioritization above, we need to address the feasibility and the amortizability of the associated cost. The data structures for task queues are chosen to be contiguous, like circular buffers, in order to increase the locality of reference for tasks and also to avoid frequent memory allocations necessary for pointer based data structures.

When we employ data structures with a total order that are contiguous, insertion to a sorted list costs $O(n)$ which may be too costly to incur per task and also increases the contention on the underlying queue. If we keep the ranking as state of a random set of entries, extracting the minimum cost and updating ranks cost $O(n)$.

In order to trade accuracy for performance, we also propose in place partial order data structures that allow cheaper extract minimum, insertion and extract a set of likely maximum till half the cost. Binary heaps provide solution for these interfaces accordingly: extracting the minimum is of constant cost with $O(logn)$ maintenance cost, insertion is of $O(logn)$ maintenance cost. Stealing is achieved by traversing the heap leaf-first in a breadth-first fashion so extracting half the work from the costlier end is of worst case $O(n)$ cost with no maintenance cost.

Let us observe the precision we lose for the performance we gain. We can interpret a heap data structure as a recursive partial orders through parent-child relationships. This recursive nature allows us to cover half the number of sorted elements at any given iteration of insertion and therefore lead to $O(logn)$ cost rather than $O(n)$, where no precision is lost. However, when trying to steal the costliest half, we can not anymore guarantee a ranking of the absolute costliest m tasks if m tasks are to be stolen. We expect the costliest of tasks to cluster on the leaf nodes and steal leaves as if they are ordered, which also eliminates the need for re-fixing the heap property. Though on any sub-heap of a heap, the costliest element of that heap is a leaf node, it is possible to have pathological cases like we showcase in figure 4.5 where the order of the leaf nodes may not be to our expectation.

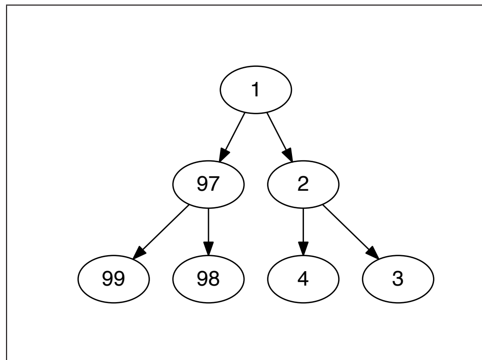


Figure 4.5 : A pathological heap for leaf-first breadth-first stealing

4.5 Experiments

4.5.1 Methods and environments

Our experimental setups are just as we described in section 3.3.1 and below are the differences for our results in this particular chapter.

As we did on section 3.3.1, we use Open Community Runtime version 0.7 is parallel runtime implementation. The base case algorithm is analogous to the default work-stealing algorithm for macro-dataflow programs discussed in the previous chapter. For a machine with N execution units, we create N workers with a total of N double ended first-in/last-out queues for ready tasks with one-to-one correspondence between execution units, workers and deques. The default local work extraction is popping the youngest task from the local queue. Once local work is depleted, the default victim selection is random, and once a victim is selected the remote work extraction (stealing) is getting the oldest task on that queue.

We proposed different heuristics above to ameliorate the default choices. Firstly, we provide a locked queue implementation to provide a base case for our workpile implementations in contrast to the lock-free deque implementation utilized by the default implementation. Though it is possible to have lock-free implementation of prioritized data structures, we chose to exclude their implementations as we postulate that the need for frequent memory allocations and compare-and-swap operations will not be amortized with the extra concurrency introduced on our experimental setups.

Our prioritized work queue implementations are specializations of the aforementioned locked queue. A total-ordered queue is a locked queue where the invariant is tasked are total-ordered with respect to their annotations of expected granularity estimates. A partially-ordered queue is a minimum binary heap on a circular queue whose invariant is the heap order being maintained.

4.5.2 An inefficient Fibonacci

As we have argued the need for better heuristics to fit work-stealing and macro-dataflow, we want to observe what are the impacts on a series-parallel benchmark

expressed in macro-dataflow. We will be using an inefficient way of computing an element of the Fibonacci sequence, rather than the linear dynamic programming solution for the sake of having an almost balanced series-parallel task graphed benchmark.

A task of height, h, leads to $fib(h+2)$ number of leaf nodes, as can be seen on figure 3.5. If we were to take into account the intermediate empty spawn edges and empty join synchronization edges, a spawn node of height, h, would still lead to $fib(h+2)$ leaf nodes. This sub-tree of spawn tasks would lead to $fib(h+2)-2$ *in* nodes excluding the root of the sub-tree, since this sub-tree is a binary tree. Since there is also a dual join edge tree for synchronization, there is an additional $fib(h+2)-1$ as this time the dual root has to be counted too. So the annotation for a pessimistic tree would have been $3 \times fib(h+2)-3$ rather than the actual $fib(h+2)$ leaf tasks with actual computation. For optimistic descendance we would have to consider all the join edges on the dominance frontier of the sub-tree, which would add an additional $max_h - h$ amount of empty synchronization nodes.

#cores	1	2	4	8	12	18	36
min lock-free time(s)	68.657	35.339	19.154	10.886	7.361	4.979	2.482
min locked time(s)	68.631	35.218	19.214	10.903	7.359	4.973	2.504
median lock-free steals	0	116696	100188	35913	77515	84284	109050
median locked steals	0	216437	104171	47349	60647	74136	65074

Figure 4.6 : $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 lock-free vs locked ready task queues

When we showcase task graphs of benchmarks throughout this work, we abstained from showing the dual join edges of the task graph that does not carry any work but

is there only to propagate the synchronization and reduce it to a single *done* node. That is why, we argued that series-parallel graph having benchmarks have a task graph that are trees. In that case, optimistic and pessimistic descendance relations converge to the actual descendance relations. We will be displaying the results for this case, where the descendance counts reflect the actual descendance relations by annotating the tasks with how many leaf task nodes with actual computation they dominate.

Figure 4.6 shows that the locked ready task queue utilization has a 1% slow-down effect on the maximal core case and competitive on all other configurations. We observe the number of steals attempts diminish which can be attributed to the contention introduced by the lock, as a failed steal attempt would take longer.

	#cores	1	2	4	8	12	18	36
default steal last no descendance time(s)	min	68.631	35.218	19.214	10.903	7.359	4.973	2.504
	median	68.655	35.468	19.279	10.935	7.365	4.978	2.524
steal half descendance time(s)	min	68.619	35.395	19.180	10.875	7.362	4.970	2.519
	median	68.655	35.628	19.236	10.917	7.366	4.978	2.539

Figure 4.7 : $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 partially sorted descendance heuristic impact on execution time

We argued that for a series-parallel task graphed benchmark our heuristics would converge to the default work-stealing algorithm. Figure 4.7 shows the slowdown to be around 1% in execution time for following our heuristics which can be attributed to the overheads introduced.

Besides outliers that can be attributed to the randomness of the steal algorithm, figure 4.8 shows that while the execution time remains competitive the number of

	#cores	2	4	8	12	18	36
default steal last no descendance time(s)	min	164598	34466	37398	24488	28377	44748
	median	216437	104171	47349	60647	74136	65074
steal half descendance time(s)	min	6907	38924	13697	26219	56321	31416
	median	194245	82089	43195	62390	66040	57057

Figure 4.8 : $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 partially sorted descendance heuristic impact on steal attempts

steals drops while our proposed heuristics are used though the underlying locked queue implementation remains. Our steal half algorithm utilizing descendance relation allows us to steal more accurate halves than stealing the oldest element that is roughly half the amount of work enqueued.

	#cores	1	2	4	8	12	18	36
default steal last no descendance time(s)	min	68.710	35.538	19.174	10.905	7.370	4.979	2.520
	median	68.759	35.659	19.208	10.928	7.373	4.987	2.541
steal half descendance time(s)	min	68.720	35.420	19.191	10.894	7.366	4.976	2.522
	median	68.765	35.687	19.229	10.929	7.370	4.985	2.545

Figure 4.9 : $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 sorted descendance heuristic impact on execution time

Comparing figures 4.7 and 4.8 versus figures 4.9 and 4.10, we can see the extra overhead introduced by maintaining a total order of granularity for a more precise steal half algorithm is not amortized and the execution times gets 0.5% worse. On a sorted task queue of granularities stealing only the oldest task provides fewer number

	#cores	2	4	8	12	18	36
default steal last no descendance time(s)	min	98721	20260	36500	38488	60455	24110
	median	194452	80070	47988	65721	73411	51031
steal half descendance time(s)	min	109079	5599	30834	37670	29768	38191
	median	203530	72110	40632	65824	62432	61162

Figure 4.10 : $fib(50)$ with cut-off 25 results for the Xeon Machine using OCR v0.7 sorted descendance heuristic impact on steal attempts

of steal attempts compared to a partial order, however for the steal half the work case partially ordered queues have fewer steals.

#cores	1	2	4	8	16	32	48	60
min lock-free time(s)	78.965	39.515	19.764	9.889	4.988	2.498	1.673	1.342
min locked time(s)	78.963	39.512	19.762	9.890	4.988	2.498	1.672	1.341
median lock-free steals	0	8856	12282	7948	9925	9648	13794	19202
median locked steals	0	8692	34426	6022	5563	10105	15811	18800

Figure 4.11 : $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 lock-free vs locked ready task queues

XeonPhi architecture features a simpler in-order core that does not seem to benefit more from the lock-free queue implementation, as it is seen on figure 4.11. As we argued before, we anticipate the default of stealing last and explicitly stealing half heuristic should be comparable for this benchmark. Figures 4.12 and 4.13 for partially sorted and figures 4.14 and 4.15 for totally ordered queues show this property to hold for execution time. For the maximal core case, we observe the number of steals

	#cores	1	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	78.963	39.512	19.762	9.890	4.988	2.498	1.672	1.341
	median	78.971	39.519	19.765	9.938	4.993	2.500	1.678	1.348
steal half descendance time(s)	min	78.971	39.516	19.763	9.889	4.987	2.497	1.670	1.343
	median	78.979	39.519	19.766	10.034	4.995	2.501	1.674	1.348

Figure 4.12 : $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 partially sorted descendance heuristic impact on execution time

	#cores	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	5389	6709	4755	4494	8622	10633	13176
	median	8692	34426	6022	5563	10105	15811	18800
steal half descendance time(s)	min	5656	6773	4584	6588	7756	10045	18378
	median	9341	14649	6967	7845	10852	11611	20394

Figure 4.13 : $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 partially sorted descendance heuristic impact on steal attempts

	#cores	1	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	79.093	39.577	19.787	10.049	4.960	2.502	1.674	1.343
	median	79.099	39.586	19.792	10.065	4.963	2.504	1.678	1.349
steal half descendance time(s)	min	79.096	39.574	19.793	9.904	5.000	2.501	1.673	1.343
	median	79.102	39.576	20.313	10.054	5.005	2.503	1.678	1.350

Figure 4.14 : $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 sorted descendance heuristic impact on execution time

	#cores	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	6968	9378	4097	5284	6799	12746	12534
	median	9573	16604	6004	6935	10986	14107	19120
steal half descendance time(s)	min	6765	7168	4772	4654	7832	9857	17863
	median	10121	12089	7221	7644	9569	14622	20741

Figure 4.15 : $fib(45)$ with cut-off 25 results for the XeonPhi Machine using OCR v0.7 sorted descendance heuristic impact on steal attempts

increases slightly for the annotated steal half the work cases. One explanation for this phenomenon is that stealing half by stealing multiple tasks may suffer from imbalances worse than stealing the oldest task that is roughly the half. We steal tasks till we pass the threshold of half the work, which may be much coarser than half the work if the task passing the half threshold is coarse. This may lead to over-stealing and ping-ponging to achieve load balance.

4.5.3 Smith-Waterman/Needleman-Wunsch sequence-alignment

In section 4.3, we talked about how pessimistic descendance can underestimate and optimistic descendance can overestimate descendance relations. If we look at the dependence graph at figure 3.10, we can see that except the outer most layer (the first row and the first column) all pessimistic descendance counts are zero, since all those tasks can be visited by distinctly multiple paths to the root task. In order to simplify, and also observe the default unannotated behaviour, we treated all tasks as leaf tasks for pessimistic descendance. So a steal-half heuristic uses a locked-queue where half the oldest tasks are extracted. In contrast, on figure 4.16 we can see the optimistic descendant counts overestimating the descendant tasks, by counting join

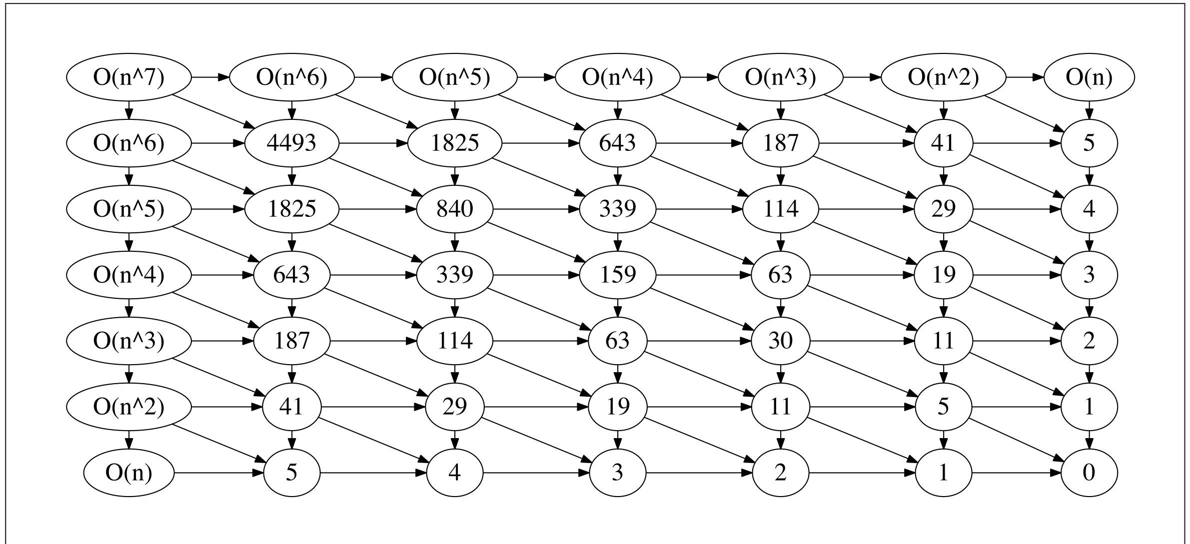


Figure 4.16 : Optimistic descendance count graph for sequence alignment

edges for every path over and over again. For example, the task annotated with a descendant count of 5 on the diagonal, can lead to 3 immediate tasks, where 2 of those tasks have descendant counts of 1 each, for the sink task node they enable. However the sink task node is count on 3 separate occasions which lead to the count of 5 rather than a 3. The recurrence relation to populate this table is not trivial function. The bottom row is linear as all of them enable a task down a chain. The row above is quadratic as the difference in two sequence is linear, and likely the row above is cubic as the row below is quadratic. The values are symmetric across the diagonal $x + y = k$, so the functions domain are just half the bottom half triangle. The first four functions are $f(x) = x$, $f(x) = x^2 + 3x + 1$, $2/3x^3 + 5x^2 + 40/3 + 11$ and $1/3x^4 + 14/3x^3 + 77/3x^2 + 196/3x + 63$, and the functions above can be calculated inductively from those or just fit to a curve of their ordinal. Building this table by counts proved a much simpler exercise, so our benchmark calculates these counts and annotates the tasks accordingly for optimistic descendance.

#cores	1	2	4	8	12	18	36
min lock-free time(s)	39.688	20.354	11.135	6.335	4.318	2.910	1.523
min locked time(s)	39.694	20.418	11.096	6.356	4.320	2.906	1.520
median lock-free steals	0	172135	844919	1063628	964971	796581	1084365
median locked steals	0	248104	1048177	1009193	814662	721328	658999

Figure 4.17 : Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 lock-free vs locked ready task queues

Figure 4.17 shows the locked queues do not incur longer execution times and with higher number of threads the number of steals diminish for the Xeon machine.

	#cores	1	2	4	8	12	18	36
default steal last no descendance time(s)	min	39.694	20.418	11.096	6.356	4.320	2.906	1.520
	median	39.700	20.443	11.198	6.381	4.364	2.937	1.531
steal half pessimistic descendance time(s)	min	39.688	20.392	11.131	6.334	4.318	2.903	1.514
	median	39.695	20.451	11.179	6.384	4.349	2.946	1.539
steal half optimistic descendance time(s)	min	39.692	20.345	11.081	6.369	4.322	2.910	1.519
	median	39.701	20.499	11.169	6.394	4.353	2.938	1.533

Figure 4.18 : Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 partially sorted descendance heuristic impact on execution time

Taking the locked queues with the steal oldest task as a base case, we see on figure 4.18 that steal half the work heuristic do not provide benefits for either descen-

dance heuristic and introduced overheads are in the milliseconds scale for the median case where the minimum execution time can be better for the pessimistic case.

	#cores	2	4	8	12	18	36
default steal last no descendance time(s)	min	112229	369973	445747	426085	346991	526196
	median	248104	1048177	1009193	814662	721328	658999
steal half pessimistic descendance time(s)	min	38598	799094	1137233	544637	344246	519281
	median	161289	1102944	1511462	773097	739668	584397
steal half optimistic descendance time(s)	min	14497	404105	583300	613631	372815	473406
	median	144937	1087902	936369	942265	698295	665971

Figure 4.19 : Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 partially sorted descendance heuristic impact on steal attempts

Stealing half the work under pessimistic descendance heuristic leads to fewer number of steals with maximal core utilization for the median case where optimistic descendance leads to more steal attempts, though optimistic descendance fares better under fewer cores utilized.

Though maintaining a sorted queue introduces overheads and contention, figure 4.20 shows that the execution times are a couple percentage points better across the board in comparison to a partially order queue case. However, stealing half the work does not provide benefits as it did not for the partially ordered case.

Comparing figure 4.20 and 4.21, we observe that the number of steal attempts have increased dramatically though the execution time went down. The contention, overheads and the load imbalance introduced help the program run faster though it

	#cores	1	2	4	8	12	18	36
default steal last no descendance time(s)	min	39.685	20.374	11.085	6.267	4.277	2.897	1.496
	median	39.698	20.499	11.113	6.324	4.280	2.904	1.499
steal half pessimistic descendance time(s)	min	39.685	20.436	11.060	6.316	4.278	2.894	1.493
	median	39.697	20.488	11.093	6.341	4.279	2.901	1.503
steal half optimistic descendance time(s)	min	39.695	20.336	11.082	6.313	4.279	2.894	1.496
	median	39.701	20.495	11.101	6.334	4.282	2.896	1.505

Figure 4.20 : Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 sorted descendance heuristic impact on execution time

	#cores	2	4	8	12	18	36
default steal last no descendance time(s)	min	2404488	865973	1372983	1612446	1201229	1228981
	median	2479098	1388011	1740426	2021978	1334203	1429434
steal half pessimistic descendance time(s)	min	2369563	517300	1128669	1253308	964977	1127862
	median	2399557	1303704	1630406	1706037	1344766	1274677
steal half optimistic descendance time(s)	min	2327468	598690	1126037	1371637	1195740	1008394
	median	2426383	1324075	1579062	1532670	1364124	1294009

Figure 4.21 : Matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7 sorted descendance heuristic impact on steal attempts

is busier stealing. As this application is memory bound, the bandwidth reduction achieved through the load imbalance helps achieve better throughput.

#cores	1	2	4	8	16	32	48	60
min lock-free time(s)	134.766	67.669	34.743	17.197	8.677	4.513	3.209	2.768
min locked time(s)	135.039	67.645	34.043	17.196	8.627	4.516	3.223	2.768
median lock-free steals	0	510941	573992	561979	350153	473096	578762	603606
median locked steals	0	515302	701050	354854	357391	360883	467350	367362

Figure 4.22 : Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 lock-free vs locked ready task queues

As it was the case for fibonacci on XeonPhi, for string alignment using a locked queue had no impact on execution time. The number of steals are noticeably fewer for the maximal core case for the locked queue implementation.

	#cores	1	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	135.039	67.645	34.043	17.196	8.627	4.516	3.223	2.768
	median	135.449	67.784	34.176	17.377	8.790	4.614	3.248	2.814
steal half pessimistic descendance time(s)	min	135.084	67.635	33.987	17.263	8.634	4.512	3.237	2.756
	median	135.370	67.821	34.185	17.336	8.812	4.531	3.265	2.782
steal half optimistic descendance time(s)	min	134.791	67.478	34.052	17.117	8.674	4.520	3.222	2.733
	median	135.168	67.729	34.203	17.276	8.823	4.590	3.277	2.773

Figure 4.23 : Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 partially sorted descendance heuristic impact on execution time

Utilizing steal half heuristics had a miniscule advantage for execution time for the

maximal core case for partially sorted queues, as can be seen on figure 4.23.

	#cores	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	214428	434656	160895	258134	290729	410202	313375
	median	515302	701050	354854	357391	360883	467350	367362
steal half pessimistic descendance time(s)	min	187241	142356	119132	258707	293072	383022	410298
	median	683617	867473	269778	434857	373860	441834	480433
steal half optimistic descendance time(s)	min	195845	478352	330130	184977	263363	372835	445425
	median	710379	801994	391780	289037	365076	406908	483459

Figure 4.24 : Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 partially sorted descendance heuristic impact on steal attempts

As it is for the Xeon results case, this miniscule improvement on execution time can be attributed to the load imbalance introduced by contention as we see number of steal attempts increase when all the cores are used.

We can see the impact of more steal attempts leading to better execution times further on figures 4.25 and 4.26 presenting the cases for sorted, more contentious queues.

4.5.4 Cholesky decomposition

Just like we discussed for sequence alignment, cholesky factorization also has a graph where pessimistic dominance relations are almost as informative as treating each task as a leaf. Accordingly, we will also use the pessimistic descendance counts of 1 for all tasks, and resort to the default behaviour. For optimistic descendance counts, as

	#cores	1	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	135.055	67.641	33.890	17.000	8.585	4.470	3.204	2.716
	median	135.365	67.806	33.944	17.018	8.653	4.489	3.227	2.758
steal half pessimistic descendance time(s)	min	134.872	67.635	33.873	17.000	8.595	4.468	3.194	2.722
	median	135.410	67.742	33.995	17.162	8.614	4.486	3.233	2.749
steal half optimistic descendance time(s)	min	135.238	67.772	34.702	16.995	8.587	4.469	3.201	2.718
	median	135.444	67.824	34.861	17.029	8.595	4.486	3.227	2.752

Figure 4.25 : Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 sorted descendance heuristic impact on execution time

	#cores	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	164929	213178	371432	394147	501369	580709	621771
	median	527960	451255	559418	620387	641609	653999	734910
steal half pessimistic descendance time(s)	min	202469	238869	403338	397109	570022	492892	543412
	median	516578	533688	534673	616280	667771	673093	660325
steal half optimistic descendance time(s)	min	418276	232263	445089	444523	532175	549408	499212
	median	479753	284024	548128	669186	658722	713847	635061

Figure 4.26 : Matching strings of size 67.5K with tile size 432 for the XeonPhi machine using OCR v0.7 sorted descendance heuristic impact on steal attempts

expected the numbers are overly optimistic and though the computation is of $O(n^3)$, the root task (the very first `dpotrf`) has the annotation to the order of $O(2^{2n+1})$. Let

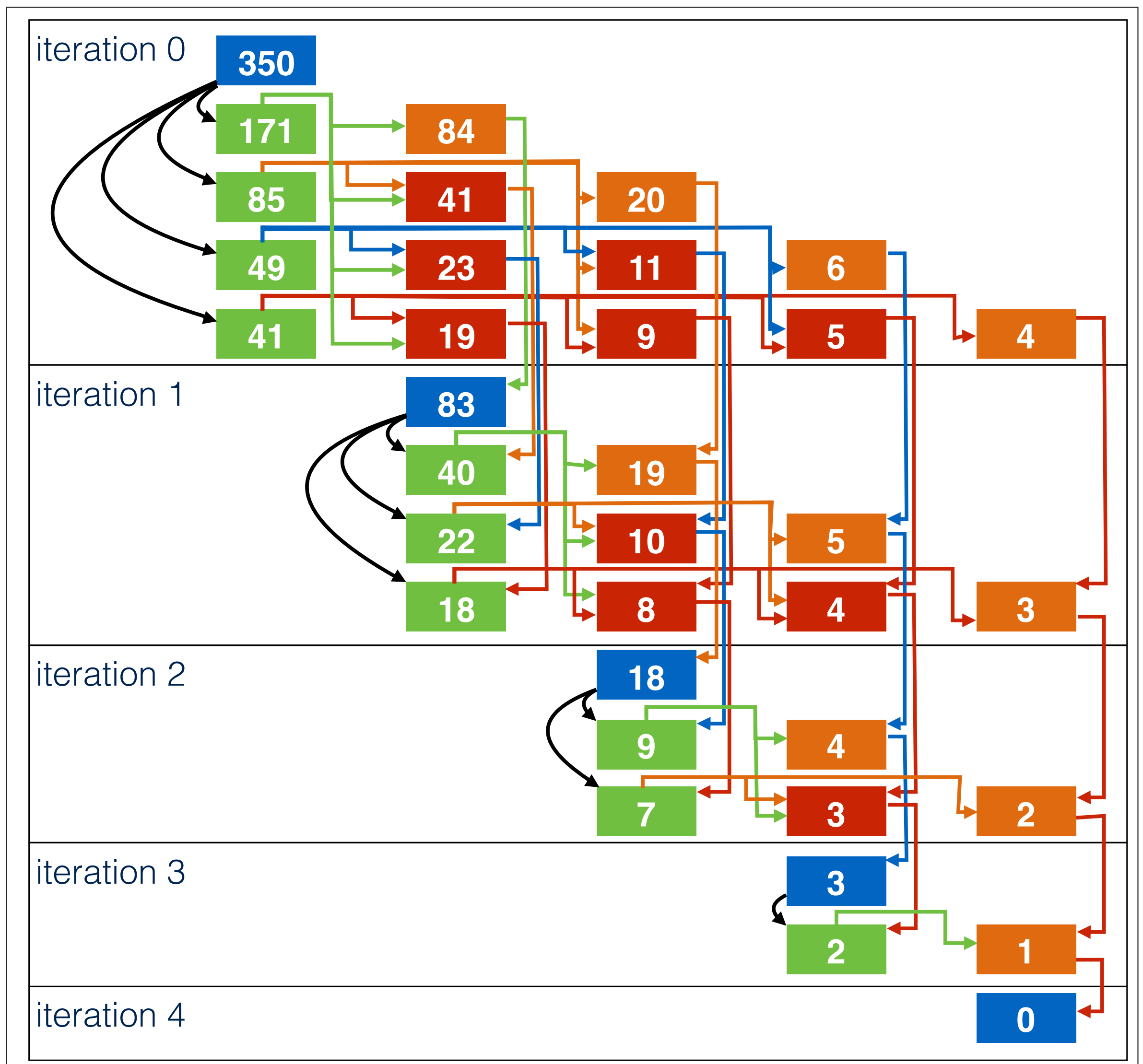


Figure 4.27 : Optimistic descendance counts for cholesky factorization

us observe the optimistic descendance counts for the example trace on figure 4.27. We can start seeing the effect of the optimism exaggerating the counts at iteration 2, where every `dsyrk` and `dgemm` are claiming credit for all the tasks downstream, where it is their join which would lead to those tasks executions.

The recurrence relations can be solved to return a polynomial, which we feed

as the annotations. Let us first start with noting the part of the function that is common to calculating any class of task, g, to be $(19 \times (2^{2k+1}) - 9k^2 + 3k + 43)/27$, where k is difference of the range of the iteration space(5 for the example) and the current iteration number. For example, the descendance counts for `dpotrf` tasks are $g(k-1) - k - 2$. If we label the aforementioned function g_{dpotrf}, then $g_{dpotrf}(1) = 0, g_{dpotrf}(2) = 3, g_{dpotrf}(3) = 18, g_{dpotrf}(4) = 83$, and $g_{dpotrf}(5) = 350$. Tasks of the `dtrsm` kind have the function $g_{dtrsm}(k,i) = g(k-1-i) \times 2^i - k - 2$, where k still is used as the difference of the range of the iteration space and the current iteration number and i is the row index on the column of `dtrsm`, where $i = 0$ for the `dpotrf` tile of the same iteration. So $g_{dtrsm}(5,1) = 171, g_{dtrsm}(5,2) = 85, g_{dtrsm}(5,3) = 49$, and $g_{dtrsm}(5,4) = 41$. Once we have the formulas for `dpotrf` and `dtrsm`, `dsyrk` and `dgemm` functions can be calculated from them. As we initially discussed, these matrix multiple update variations are the computations that introduce the loop-carried dependences. So a `dgemm`(or `dsyrk`) starts a chain that ends in a non matrix multiply task. For example, we can see the whole iteration 1 numbers on the graph are 1 fewer than their corresponding iteration 0 ancestors. Likely, if we look at the task labeled 6 at the iteration 0, it is a chain that feeds into the `dpotrf` of iteration 3, accumulating 1 on every iteration. For the sake of completeness, the function for `dsyrk` is $g_{dsyrk}(k,j) = g(k-1-j) - k - 2 + 2j$, where k is as mentioned above and j is the row/column number where $j = 0$ for the `dpotrf` tile of the iteration, so $g_{dsyrk}(4,1) = 19, g_{dsyrk}(4,2) = 5$ and $g_{dsyrk}(4,3) = 3$. Lastly, $g_{dgemm}(k,i,j) = g(k-1-i) \times 2^{i-j} - k - 2 + 2j$, where k is the same as the functions above, and i and j are the row and the column indexes where $i = 0$ and $j = 0$ for the `dpotrf` tile of the iteration. So $g_{dgemm}(4,2,1) = 10, g_{dgemm}(4,3,1) = 8$ and $g_{dgemm}(4,3,2) = 4$.

Results

#cores	1	2	4	8	12	16	18	36
min lock-free time(s)	16.475	8.139	4.193	2.349	1.612	1.353	1.260	0.670
min locked time(s)	16.443	8.216	4.210	2.352	1.620	1.350	1.276	0.695
median lock-free steals	0	5545867	1872148	868718	1339106	1657395	1730409	2171493
median locked steals	0	6283501	1671696	867156	1094485	1195346	1213038	1212280

Figure 4.28 : Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine lock-free vs locked ready task queues

Let us observe the effect on the proposed heuristics on the Xeon machine. We use the tile size 192 by 192 doubles on a 12288 by 12888 matrix, which provides the best results in FLOPS. First, we compare the default lock-free deque implementation with the locked deque we use for this section on figure 4.28. The locked deque implementation used for ready work queues causes a 3% drop on throughput for the 36 core case and for most cases it is still competitive within that noise range. As the queues are locked, even with the same policies for scheduling, the number of steals are drastically reduced compared to the lock-free case, since pop and steal can not be run concurrently anymore. We will be using the locked queues implementation as our base case for the heuristics results to be addressed below.

Figure 4.29 shows the results for partially ordered ready tasks for optimistic and pessimistic descendance counts and for steal half the available work policy. Using a

	#cores	1	2	4	8	12	16	18	36
default steal last no descendance time(s)	min	16.443	8.217	4.210	2.352	1.620	1.350	1.276	0.695
	median	16.551	8.250	4.231	2.373	1.633	1.360	1.284	0.711
steal half pessimistic descendance time(s)	min	16.339	8.195	4.216	2.359	1.613	1.348	1.279	0.671
	median	16.543	8.282	4.244	2.380	1.632	1.361	1.290	0.683
steal half optimistic descendance time(s)	min	16.345	7.981	4.243	2.371	1.625	1.356	1.261	0.663
	median	16.556	8.150	4.311	2.402	1.647	1.370	1.278	0.674

Figure 4.29 : Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine partially sorted descendance heuristic impact on execution time

steal half the amount of ready work according to pessimistic or optimistic descendance relations lead to 3% or 5% improvement in throughput respectively for the 36 core case for minimum execution times of 10 runs. For the median execution times the improvement is 4% and 5% respectively. These results are better than the default lock-free case which we have shown in the previous chapter to have outperformed MKL, and make up for the overhead and contention introduced by the locked deques.

The impact on the number of steals can be seen on figure 4.30. With the exception of the 2 core case, stealing half the estimated work optimistically or pessimistically lead to fewer number of steals and better load balance.

Comparing figures 4.31 and 4.32 versus figures 4.29 and 4.30, tells us that for the fewer number of cores the overhead and contention introduced by maintaining a contiguous ordered queue is amortized. However scaling is not as steep as the

	#cores	2	4	8	12	16	18	36
default steal last no descendance time(s)	min	5879889	1574198	814140	1047042	1154373	1175952	1023233
	median	6283501	1671696	867156	1094485	1195346	1213038	1212280
steal half pessimistic descendance time(s)	min	5817302	1545589	786202	1018884	1122402	1124866	986641
	median	10665503	1661004	815038	1058976	1148046	1164886	1078297
steal half optimistic descendance time(s)	min	5858771	1542175	764717	1032906	1126424	1157081	1004314
	median	15505355	1665103	797596	1058638	1179674	1192685	1134369

Figure 4.30 : Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine partially sorted descendance heuristic impact on steal attempts

	#cores	1	2	4	8	12	16	18	36
default steal last no descendance time(s)	min	15.271	7.959	4.266	2.423	1.657	1.367	1.303	0.708
	median	15.379	8.052	4.308	2.439	1.677	1.395	1.316	0.713
steal half pessimistic descendance time(s)	min	15.247	7.784	4.134	2.356	1.600	1.336	1.268	0.703
	median	15.390	7.867	4.164	2.366	1.623	1.349	1.276	0.710
steal half optimistic descendance time(s)	min	15.270	8.009	4.260	2.415	1.654	1.376	1.302	0.705
	median	15.393	8.057	4.295	2.441	1.673	1.391	1.310	0.722

Figure 4.31 : Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine sorted descendance heuristic impact on execution time

	#cores	2	4	8	12	16	18	36
default steal last no descendance time(s)	min	5938836	1554192	757351	1029531	1147752	1152238	1064868
	median	15779814	1727265	780512	1090875	1186915	1199266	1105933
steal half pessimistic descendance time(s)	min	5622065	1496201	776626	1030586	1080877	1144362	1052693
	median	15282867	1709527	806670	1055041	1118891	1182699	1139030
steal half optimistic descendance time(s)	min	5555446	1545787	759326	1027088	1087020	1126756	1073924
	median	6056469	1631443	804290	1044307	1132836	1181772	1172316

Figure 4.32 : Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine sorted descendance heuristic impact on steal attempts

partially ordered case and the 36 core case for the sorted queue results show a lower throughput compared to the partially order case. The overheads and contentions introduced challenge the scaling sought by the heuristics we proposed.

On XeonPhi, our base case results from the previous chapter is presented in figure 3.17. Let us see the impact of our proposals above by changing one parameter at a time. First, let us compare the impact of a locked deque implementation we use that supports the heuristics we described. We start with presenting the values for the tile size providing the highest throughput, 96 by 96 doubles.

Figure 4.33 shows that the lock contention expected from 60 workers stealing from one another did not materialize. On the contrary, the minimum execution time improved by 0.5%. Additionally, the number of steals also decreased for higher number of cores. One outlier is the two core case, where the minimum is fewer than

#cores	1	2	4	8	16	32	48	60
min lock-free time(s)	27.845	13.952	7.039	3.568	1.833	0.972	0.718	0.642
min locked time(s)	27.878	13.963	7.027	3.569	1.835	0.980	0.723	0.639
median lock-free steals	0	2141999	229449	357942	561748	464851	798042	925303
median locked steals	0	5223875	257613	371591	459488	370755	614567	748908

Figure 4.33 : Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine lock-free vs locked ready task queues

the lock-free case but 6 out of 10 trials ended up stealing 5 million times, making the median much higher. However, we should note it is possible for the fewer number of steals and faster execution time to be caused by contention. The workers could be spinning rather than overloading the cache coherency and main memory bandwidth. For our granularity work, the locked queue is used as the base line to compare the impact of the heuristics.

Figures 4.38, 4.39, 4.40, 4.41 show the impact of the heuristics on a smaller tile size of 64 by 64 doubles. As we expect parallelism to increase in the future in order to fit power budgets, to facilitate strong scaling one has to support smaller tile sizes. We see that stealing half the estimated work has a bigger impact on smaller tile sizes compared to the tile size we used that provided the maximum throughput. However, as in other cases, sorted contiguous ready work queues do not amortize the overheads and contentions introduces for maximal core utilization.

	#cores	1	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	27.878	13.963	7.027	3.569	1.835	0.980	0.723	0.639
	median	27.892	13.980	7.099	3.610	1.846	0.985	0.733	0.661
steal half pessimistic descendance time(s)	min	27.872	14.033	7.125	3.597	1.836	0.977	0.710	0.622
	median	27.882	14.044	7.176	3.628	1.851	0.983	0.719	0.629
steal half optimistic descendance time(s)	min	27.859	14.179	7.028	3.562	1.829	0.974	0.722	0.631
	median	27.870	14.199	7.042	3.586	1.846	0.983	0.729	0.648

Figure 4.34 : Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine partially sorted descendance heuristic impact on execution time

	#cores	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	2048945	242353	360395	437370	364267	506046	732313
	median	5223875	257613	371591	459488	370755	614567	748908
steal half pessimistic descendance time(s)	min	1965356	212124	345750	423646	347285	534442	687727
	median	1996673	248181	360618	443965	351748	570376	704295
steal half optimistic descendance time(s)	min	1979577	215427	351142	427654	354911	561870	709329
	median	5056339	243876	364379	447438	362438	603603	725036

Figure 4.35 : Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine partially sorted descendance heuristic impact on steal attempts

	#cores	1	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	30.690	15.983	7.418	3.715	1.876	0.986	0.713	0.617
	median	30.767	16.058	7.437	3.729	1.886	0.989	0.718	0.634
steal half pessimistic descendance time(s)	min	30.682	15.165	7.524	3.767	1.904	1.020	0.746	0.663
	median	30.786	15.192	7.551	3.784	1.917	1.026	0.756	0.682
steal half optimistic descendance time(s)	min	30.742	15.139	7.445	3.702	1.884	0.987	0.715	0.624
	median	30.780	15.181	7.475	3.722	1.895	0.994	0.723	0.628

Figure 4.36 : Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine sorted descendance heuristic impact on execution time

	#cores	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	1785920	216064	362403	451408	352923	495241	695828
	median	1810773	252212	370426	453404	355821	578489	703770
steal half pessimistic descendance time(s)	min	2028367	232253	345794	450156	360563	496183	705870
	median	5043777	244381	364736	461513	368264	584401	745836
steal half optimistic descendance time(s)	min	1956378	231960	348784	410939	345974	472524	680292
	median	4643201	242595	361844	441988	352903	564978	692718

Figure 4.37 : Cholesky decomposition results for a 6K by 6K matrix with 96 by 96 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine sorted descendance heuristic impact on steal attempts

	#cores	1	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	35.238	17.833	9.060	4.546	2.303	1.229	0.902	0.826
	median	35.253	17.871	9.123	4.587	2.338	1.239	0.927	0.835
steal half pessimistic descendance time(s)	min	35.244	17.929	9.022	4.563	2.306	1.211	0.883	0.757
	median	35.252	18.025	9.071	4.577	2.317	1.224	0.889	0.772
steal half optimistic descendance time(s)	min	35.245	18.159	8.939	4.499	2.286	1.212	0.897	0.796
	median	35.257	18.258	8.965	4.524	2.314	1.232	0.904	0.815

Figure 4.38 : Cholesky decomposition results for a 6K by 6K matrix with 64 by 64 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine partially sorted descendance heuristic impact on execution time

	#cores	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	2345957	239281	405546	491642	432739	697022	868260
	median	5694922	261125	417818	515249	443344	729196	884956
steal half pessimistic descendance time(s)	min	2255943	241122	403568	476623	395313	590154	771575
	median	3985901	259792	408756	500862	399478	662676	788819
steal half optimistic descendance time(s)	min	2270256	239354	389497	476122	412845	576533	829182
	median	3895726	262234	409494	510881	421011	695410	838279

Figure 4.39 : Cholesky decomposition results for a 6K by 6K matrix with 64 by 64 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine partially sorted descendance heuristic impact on steal attempts

	#cores	1	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	42.311	19.997	9.659	4.760	2.391	1.251	0.902	0.801
	median	42.326	20.033	9.999	4.786	2.422	1.264	0.911	0.821
steal half pessimistic descendance time(s)	min	42.301	20.270	10.050	5.000	2.512	1.325	0.969	0.838
	median	42.341	20.303	10.066	5.015	2.524	1.338	0.980	0.846
steal half optimistic descendance time(s)	min	42.307	19.966	9.657	4.767	2.414	1.269	0.905	0.784
	median	42.330	20.015	9.779	4.838	2.452	1.284	0.920	0.831

Figure 4.40 : Cholesky decomposition results for a 6K by 6K matrix with 64 by 64 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine sorted descendance heuristic impact on execution time

	#cores	2	4	8	16	32	48	60
default steal last no descendance time(s)	min	2347000	244532	391480	487895	398495	614862	836222
	median	5670729	263937	423531	496045	402649	664766	853802
steal half pessimistic descendance time(s)	min	2299815	254199	395518	482656	405872	620655	786072
	median	2321755	279799	413318	502440	412174	675328	796520
steal half optimistic descendance time(s)	min	2236455	239951	385190	470013	391511	616382	780760
	median	2258287	254851	414864	495533	400599	647527	825821

Figure 4.41 : Cholesky decomposition results for a 6K by 6K matrix with 64 by 64 tiles using OCR v0.7 with Intel MKL for the XeonPhi machine sorted descendance heuristic impact on steal attempts

Chapter 5

Locality-aware Scheduling in Event-Driven Runtime Systems

5.1 Introduction

In previous chapters, we discussed why macro-dataflow parallel programming models support more general task-graphs than nested fork/join models. The support for more general task-graphs required us to address the granularity aspect of the underlying runtime work-stealing algorithm, which we covered in chapter 4.

Work-stealing algorithms for series-parallel graphs do not only have inherent granularity benefits, but also locality benefits. Let us recap the discussion on the granularity benefits of restricting task-graphs to series-parallel graphs and stealing the *oldest* task first, from section 4.1. A thread exploring the task graph traverses the data structure in a depth-first fashion and enables unexplored paths to be stolen by idle threads. Series-parallel task graphs are declaring *control* dependences between tasks because of their imperative nature. Since data dependences have to have been satisfied for a child task at creation time, the control dependence graph is also a superimposed data-dependence graph. So as tasks get further decomposed deeper on the tree, these tasks' input data are also getting decomposed, and therefore the memory footprint is anticipated to get smaller. As data footprint gets smaller, the data is likelier to fit in the closest hierarchy in memory. As a thread traverses the task graph in depth-first fashion, and leaving a last-in first-out (*youngest* to *oldest*) trail

of unexplored paths, the thread is executing tasks that are *closest* on the task graph topology. If the data decomposition *closeness* matches the task decomposition, this algorithm has tight locality bounds with a least-recently-used eviction policy. Additionally, since stolen tasks are also sources of series-parallel graphs (i.e. roots of task-trees), the properties hold for stolen sub-trees of tasks recursively.

We also talked about divide-and-conquer algorithms being a good fit for series-parallel models. A thread with a private cache using divide-and-conquer recursion (considering it is executing the non-stolen part of its task graph *serially*) can also be classified as a *cache-oblivious* algorithm [40].

Locality benefits of executing tasks that are closer in an execution graph on the same processor for work-stealing runtimes have been covered in [41] and in [42]. Blumofe, in [42] shows that the subset of computation graphs that are series-parallel (nested fork/join) incur better locality. A detailed theoretical discussion on the locality properties of series-parallel task graphs can be found on [43, 44, 45, 46].

In our discussions in chapter 4, we argued stealing *oldest* task has granularity benefits and it is expected on average to have the tasks available for stealing from a thread are implicitly ordered from coarse to fine grain. The same property can be extended to locality. The newest tasks are likelier to consume data that is close to their sibling task's data that have left the cache favorably dirty for the newest tasks. If the cache holds more data and uses least recently used eviction policies, the cache is likelier to hold data from closer levels of their pedigree than further ones.

As macro-dataflow models utilize data dependence as a first-level construct, application programmers can declare computations that are unstructured DAGs, which are a superset of nested series-parallel graphs. Optimizing locality is more challenging in the more general context of DAG parallelism, relative to fork-join parallelism [47].

This follows from the fact that is also covered in section 4.2. We discussed for figure 4.4 that we can not statically deduce the availability of predecessors for a task, since in our model there can be more than one successor per task. A task can only become ready when all of its dependences are satisfied; the order these dependences may be satisfied is schedule dependent and can only be deduced at runtime. This impacts locality, because now the *closeness* in the task graph is also schedule dependent. Additionally the distance to a single predecessor, a constant, is sufficient for a locality metric for series-parallel graphs, where the distance metric is n-dimensional for a task with n predecessor tasks.

We will explore data-structures and policies adopted by work-stealing runtimes and propose ameliorations for better locality results for event-driven runtimes using work-stealing.

5.2 Task queues

Task queues, in conjunction with policies for local work extraction and work stealing, impose an implicit ordering as we discussed. We will observe the implications of data structure choices for task queues for locality optimizations.

5.2.1 Deques

As discussed before, double ended last-in first-out queues (deques) have been the data structure of choice [25, 35, 48] when it comes to implementing task queues for work stealing runtimes.

We will be re-using the figure 5.1 to make the locality discussion clearer for work-stealing using deques. Here we observe a divide-and-conquer benchmark's task graph unfolding, and the state of the worker (and therefore the deque) doing the unfolding.

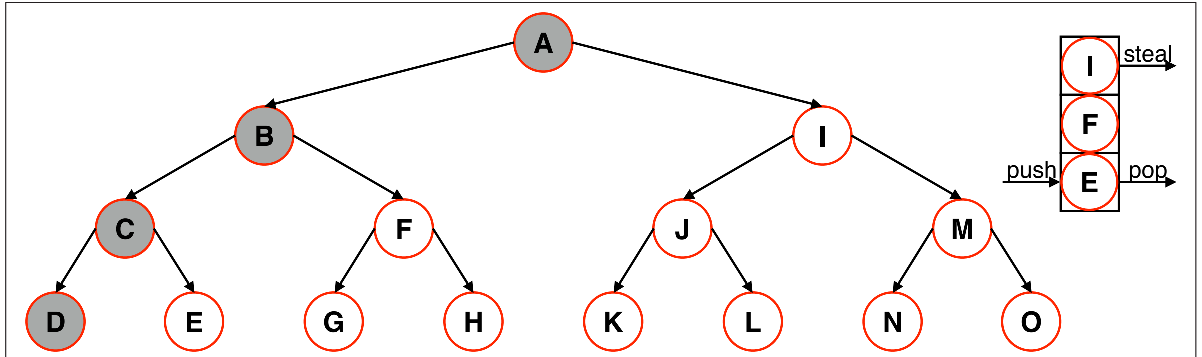


Figure 5.1 : Task graph of a divide-and-conquer application run under *work-first* policy

On this state, we expect $task_E$ to have the most data locality as $task_D$ and $task_C$'s executions may have brought its data to the closest memory hierarchy. The data brought in for $task_B$ is likelier to be evicted than $task_D$ or $task_E$'s data, so $task_E$ is a better candidate for locality than $task_F$. One can inductively build the same argument for all the explored paths' root tasks. Additionally, this tree could have been a stolen task subtree, and therefore the properties also hold for stolen tasks.

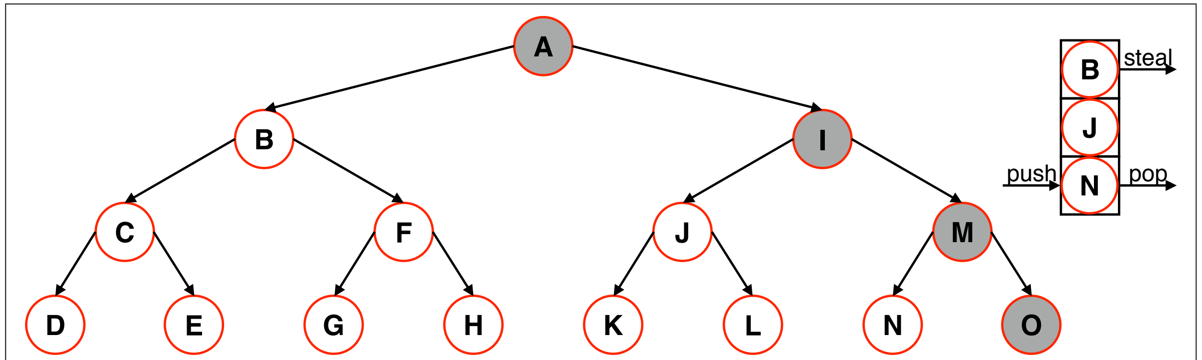


Figure 5.2 : Task graph of a divide-and-conquer application run under *help-first* policy

We can also observe the impact of a *help-first* policy on figure 5.2, which is the mirror image of the *work-first* picture with the same properties. It is the depth-first traversal, not the left-most child exploration that matters. Help-first policy is

likelier to behave like a breadth-first traversal when tasks have higher fan-out and the task-graph is not recursively-decomposed.

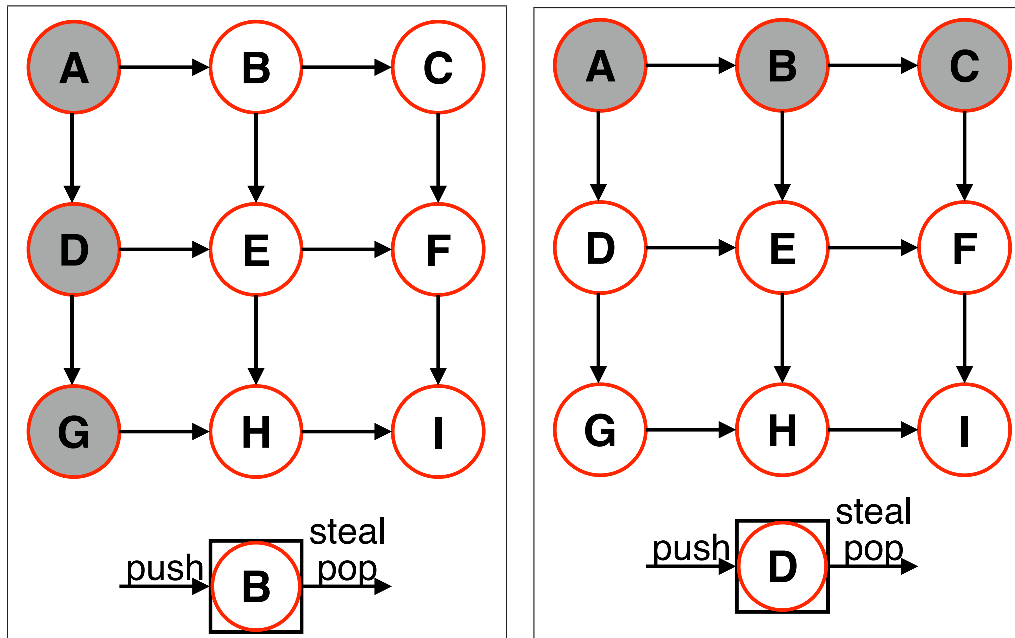


Figure 5.3 : Non series-parallel task graphs

Now, let us observe a non series-parallel task graph as in figures 5.3, left for *work-first* and right for *help-first* policies. On the left side of the figure, we can see the scheduler picked $task_A$, then $task_D$ and lastly $task_G$ to be executed. The data that is freshest to be consumed on the memory is $task_G$'s, then $task_D$'s and lastly $task_A$'s. To take advantage of better locality, one might want to schedule $task_H$, but because of this particular runtime schedule, that task is not yet available to be executed and the least local task, $task_B$, will be chosen. This is not the only possible runtime schedule, as can be observed from the graph on the right for figure 5.3.

We can conclude that a deque does not provide the same implicit locality or granularity orderings because of the schedule dependent nature of the task frontier

and the not taking multiple paths converging being taken into account. Since we do not want to constrain our runtime to divide-and-conquer like programs, we need task queues besides deques for when the assumptions that make deques favorable do not hold anymore.

5.2.2 Explicitly prioritized data-structures

In section 5.2.1, we argued that deques do not implicitly order tasks for locality and granularity for non-series-parallel graphs. Therefore, we postulate that we need task graphs where the ordering, or classifying, of tasks are more explicit and schedule dependent since locality metrics are schedule dependent for DAG parallelism.

One possible locality measure for tasks, is the dynamically calculated cost of bringing the data to consumer task. As our models have explicit producer/consumer relationships through dependences, and since shared data objects among tasks are also explicitly expressed, we can keep track of these dynamically at the runtime system. Therefore we can estimate where the data, that a task depends on, are and how much it would cost to bring all that data in at scheduling time for that task. If we employ a priority queue, instead of a deque, and use the cost of data movement per task as the priority, we will have a task queue that has a most local to least local ordering of tasks. These costs are computed when a task becomes ready and remains constant as long as it stays on the same queue. This simplifies the implementation at the cost of accuracy, but saves us from the cost we would incur from simulating caches.

One major assumption here is knowing where dependences have been satisfied and where the data that satisfies the dependence is located. Since dependences are monotonic structures that can be satisfied only once, we can deduce where the dependences have been provided. However, what matters is knowing where the data

that satisfied the dependence lives. For simplicity, a one-to-one correspondence can be assumed initially, as if the 'producer' of the data is also the 'creator' or 'allocator' (or 'over-writer' if the dependence is a storage dependence). On a cache-architecture, if a data is consumed in two separate contexts, the data can be replicated to live in more than one place, making the placement tracking more complicated. For simplicity, we will initially focus on the intrinsic data transfer effects of locality and assume that each worker thread runs on a core with an unbounded eviction-free cache. This ignores the impact of capacity and conflict misses. Additionally, since caches are not of infinite size, data eventually gets evicted which should also be taken into account.

This locality ordering for a task per thread is 'dynamic', as it is delayed until creation time. However, another policy decision to make is to decide if the costs (and therefore priorities) should also change dynamically. One option is to simplify the design at the cost of inaccuracy and not update the cost of a task based on data eviction and replication, the other is to simulate a cache in software to update costs accordingly at runtime. Initially, we opt for not updating cost of a task once it is enqueued.

Another possibility we are exploring is prioritizing the tasks on the critical path and reducing the risk of exposing the task graph in a fashion that would introduce bottlenecks. Figure 5.3 features two example of exposures of the task graph that are not favorable to parallelism. However, we should note that the runtime exposes the task graph dynamically during runtime, so it has no way of knowing where the critical path lies. We would require the programmer to hint or annotate tasks accordingly to utilize this feature.

5.2.3 Recursive data structures

Though deques are linear structures, as we discuss on section 5.2.1, when we are working on a recursive decomposition task graph, the tasks on the deques are roots of task trees that grow exponentially. Hence, when a task is stolen roughly half the amount of computation is actually stolen are actually stolen, since the left over work is $\sum_{i=1}^{depth-1} = 2^{depth} - 1 - depth$ and the stolen work is $2^{depth} - 1$, as we argued in chapter 4.

Let us recap some of the points we made on how to employ granularity optimizations. Once we work with non-series-parallel graphs, tasks have descendants rather than children and those descendants may have many predecessors. So stealing a task may not lead to a sub-tree of tasks to be stolen. We introduced descendance heuristics in the previous chapter to help steal half of the estimated work on a ready queue. For example, computations with intricate dependence structures lead to pessimistic descendance to estimate no descendant tasks as we have shown in the previous chapter. That implies that to steal half the work, we may need to steal half of the ready tasks on the task queue.

We postulate that we may alleviate the granularity problem that non-series-parallel task graphs may introduce, with recursive data structures for task queues without losing the locality prioritization. Recursive data structures would make stealing half of the structure easy to implement and would require less synchronization to consider. Moreover, if the relevant property for the data structure (e.g. priorities) also hold recursively, we benefit from stealing a set of tasks which can be utilized as task queue by the thief without additional manipulation.

If we employ a priority-queue of tasks for better locality, coarse granular stealing should work in tandem with this ordering. For example, if a task queue is implemented

as a binary heap, the task that would incur the best locality would be at the root for the owner thread to exploit. The likely least local tasks would be at leaf level, for the thieves to exploit. On a balanced heap, the number of leaves is almost equal to all the non-leaf nodes. However stealing all the leaves may increase contention (new tasks are pushed at the leaf level and then heap property is fixed) and the heap may not necessarily just have the worst of the worst tasks with respect to locality. Stealing a child of the root node would give almost half the number of tasks that are likely to have the distribution of locality of the original heap has, which maybe favorable if the thief is topologically adjacent to the victim. A self-balancing binary search tree would have the task that would incur the best locality as the left-most leaf, where the right child of the root node would have the worst of the worst locality incurring tasks, that is almost half the task queue. All these choices should be made in the context of the steal policy heuristics chosen for the runtime. We will elaborate further on these on section 5.4.2. Binary heap implementation for priority queues provides a partial order to the tasks and this lack of precision can be used to achieve better performance [49].

5.3 Task scheduling heuristics

Series-parallel programming models that constrain the task graphs expressed to trees, have a clear parent-child relationship between the tasks. They may employ eagerly executing a child task and leaving the continuation pending; this is aforementioned as work-first work-stealing policy and is a left-most depth-first traversal of the task graph. Alternatively, child tasks may be left pending as the continuation is being executed; this is the right-most depth-first traversal of the task graph, as formerly discussed, help-first strategy.

In our work, we explore the effect of non-parent/child predecessor/successor relationships and the challenge of scheduling tasks with these more general relationships. In our previous work [21], we did not provide an implementation to utilize work-stealing on non-series-parallel graphs. The implementation for work-stealing support for tasks with multiple predecessors is analogous to *currying*. Every predecessor of a tasks perform a *partial function application* till all the dependences are met. The last predecessor changes the descendant task to a zero arity function, making it *ready*. We treat this as if the last predecessor created a child task. Since we use a help-first policy on OCR, the descendant task gets pushed into the local ready task queue. The task graph frontier for execution is schedule-dependent because this currying is schedule dependent.

This simplification provides an implicit scheduling heuristic, namely *most recent satisfied dependence first*. When a task satisfies a dependence, for that dependence it walks through all the awaiting tasks synchronization frontiers. If all other dependences are satisfied for a task, that task is enabled and scheduled to be executed. Hence the last satisfied dependence is the enabling one, it leads to the enabled task to be enqueued on the same worker. Though this may help with simplifying the implementation and synchronization concerns, it may not necessarily provide the best performance, as this choice may not provide minimal data retrieval cost, and therefore locality performance. For example, a task with n dependences may have its first $n-1$ dependences met at $thread_i$, where the last one is met at $thread_j$, where $i \neq j$. If this task is executed on $thread_j$, all the necessary data may be replicated on $thread_j$, may incur possible cache misses and may evict data that is local to tasks waiting to be executed on $thread_j$.

5.3.1 Alternative push policies

Let us formulate the *push* heuristic of a locality-aware work-stealing runtime as a function p. This function p takes the places of a task's predecessors as arguments and returns a place for the successor task to be pushed. As discussed above, for tree like task graphs p is a unary function, that is the identity function, $\forall_{pl \in places}, p(pl) = pl$. Simply put, child tasks get placed in their parent's queue.

Our initial implementation for work-stealing support, chooses the function p to be a n-ary function for tasks with n dependences, where it returns the ith element, $\forall_{pl_1, pl_2, \cdots, pl_n \in places}, p(pl_1, pl_2, \cdots, pl_n) = pl_i$, where i^{th} thread is where the last dependence is satisfied on this particular schedule. We postulate that a push heuristic should not be a function where $n-1$ arguments are ignored.

We employ a group of *push* heuristics that are more cost aware, based on varying definitions of cost that we will elaborate further in this chapter. One simple heuristic is to run the task on the thread with the largest number of input arguments for the task. The hypothesis is temporal locality of data to be consumed, will increase if a task is run where most of its predecessors have run. This function can be further refined to take into account the dynamicity of work-stealing and the underlying memory hierarchy of the machine. Tasks that ran at a thread may have had predecessors that were stolen to be run at another thread, which results in their produced data to be copied to the thief thread. Further, if we are working on a cache-based machine, it is possible for the data to exist in more than one cache at once. Therefore we may further refine our *push* heuristic to take replication and eviction of data into account.

Cache-based machines are hierarchical in nature and so is the cost of bringing data in is hierarchical and uniform within a hierarchy. However the cost may be non-uniform, as on a distributed memory machine. The cost of bringing data may

be dependent on how *far* are machines on the network topology. So when scheduling task with n dependents, rather than counting predecessor threads, we may utilize calculating a *centroid* thread, that minimizes the *variance* on the set of predecessor threads.

5.4 Task stealing heuristics

The differentiator of work-stealing runtimes from the rest, as the name implies, is utilizing the idle workers to do the load balancing by letting them steal tasks from busy workers. There are two-tiers of heuristics to stealing; first, victim selection (from where) and secondly, which tasks to extract from the selected victim.

5.4.1 Victim selection

A common implementation choice for a victim selection heuristic is the random victim selection [42], under the observation that a uniform random distribution reduces the average number of steal attempts to find work and is used to also to prove the theoretical bounds of work stealing. Though random victim selection has proven guarantees for load balance, it fails to address locality concerns. On a machine with a deep memory hierarchy, it may be favorable to prioritize the most local tier of workers to be the first set of victims. Then, the workers that are further in the memory hierarchy can be traversed as potential victims with lower priority.

Hierarchical traversal

As we suggested before, cache-based machines are mostly hierarchical, e.g. multiple threads sharing a cache, on a multi-cache socket, on a multi-socket machine. We can take this into account on our traversal to replicate the locality inherent to this

hierarchy by traversing bottom-up for better locality as covered in [50]. Following the aforementioned sample machine, initially, a thread would try to steal from the threads it shares a cache with, then it would traverse threads it only shares a socket with and then explore threads on other sockets.

5.4.2 Task extraction

Once a victim is chosen, the second policy aspect to pin down is to decide which tasks to steal. We have argued for granularity optimizations for DAG parallelism on chapter 4, in the absence of locality concerns. Taking locality optimizations into account does not invalidate the case for granularity. So for now, let us assume that we are employing the steal-half policy that we have been advocating.

Given that we are pursuing locality optimizations by classifying and ordering tasks in ready task queues, which changes task queue choices, which in turn changes how a steal-half heuristic would work. We have offered the initial discussion on section 5.2.3.

Secondly, another policy to consider is *which* half to steal. Granularity optimizations offered before did not differentiate between subsets of tasks, as long as they are half the size of the queue. With locality optimizations, we have an opportunity to decide *which* half.

Let us initially introduce the two extremes of *which half to extract* policies:

altruistic stealing

Locality optimizations are pursued through ordering (partially or totally) ready tasks by how much data retrieval cost they would incur. By default and historical convention, in order to reduce the synchronization cost on the task queue, stealing is done from the alternate end of the queue, rather than the end the owner thread uses. So

when a thief extracts half the work from the other end of an ordered queue, they are likely to get the tasks with high data retrieval costs to the victim, hence the name *altruistic*. This policy is achieved by implicitly, just by using ordering for locality and by stealing with a coarse grain.

There are multiple pitfalls to this approach. Firstly, let us assume that the victim selection policy used is hierarchical. So an idle thread may randomly try to steal from a thread that it may be sharing a cache or a socket. What was deemed costly for the victim, will also be costly for the thief, because of their mutual proximity. One might argue that the idle worker is a better place to incur cache misses than a busy worker. However, when the idle worker and the busy worker are close-by, this increases the chances of capacity misses in the lowest common ancestor memory and burden the busy worker. In this case parallelism for load balance is favored over locality.

Another problem is the possible nature of the tasks that are deemed costly to execute. It is possible the reason that a task is costly for one thread, is the same reason it is costly for all other threads. If a task's input is scattered across the machine, the task is not local to any thread. So when these kinds of tasks are stolen by a thief, it is likelier to be re-stolen and re-stolen, as it is costly for every thread. This will not lead to starvation as every task will eventually have to get executed, but it may lead to significant delays. It is not unlikely for these tasks' delay in execution to constrain the width of the parallelism available, since they are likely to have high fan-ins or be tasks on the critical path. This may lead to less parallelism and increased contention.

selfish stealing

Ideally, every thread would benefit from executing what is local to them. So the natural extension to this idea is that thieves would benefit from stealing tasks that are most local to them.

However an implementation of this idea is non-trivial. Firstly, for the ideal case, every thread would need to know what is local and not-local to every other thread, to maintain locality orderings for the thieves. This would lead to quadratic increase in task queues to maintain, which would introduce significant overhead and invalidate the decentralization property that allows work-stealing runtimes to scale.

One solution is for the thieves to calculate on-the-fly what to steal from victims by calculating how much every task would cost to them. This would complicate proper synchronization of the task queues and introduce contention, the higher the granularity of stealing is.

Just like altruistic stealing covered above, *selfish* stealing may suffer from *topology obliviousness.* A thief, $thread_A$, stealing what is best for itself from $thread_B$, is analogous to $thread_B$ stealing what is worst from $thread_A$, if these threads are sharing a cache. One thread in both cases gets the best for both, when the other gets the worst for both subsets of tasks. Additionally, the close-by threads may ping-pong the same subsets back and forth, just like altruistic stealing might, for the same reason.

For both these algorithms, we can argue the policy for what is being stolen, is not an orthogonal decision to the victim selection policy. We are exploring victim-selection (or topology) aware policies that compromise between these two extremes. One possible compromise policy is using these extremes when the victim and thief are far apart. When the victim and thief are close to one another, a neither selfish nor

altruistic policy may be employed that approximately splits (clusters) a partial order of locality, into multiple partial orders that are preferably acyclic. This way both threads may get a partially-ordered set of tasks, that reflects their mutual locality properties without favoring one over the other.

5.5 Results

We explore the impact of the heuristics covered above by observing their effect on caches for today's machines. The XeonPhi setup we used in the chapter above does not support native hardware performance monitor observation through the performance library, PAPI. As a result, we will focus on the the Xeon machine for the results presented in this chapter.

The OCR setup is the same as the previous chapters, where there is a one to one correspondence among cores, workers and ready task queues. We report results for the maximum number of cores, since our goal is to increase locality without hampering parallelism.

The benchmarks studied in this chapter are string alignment and cholesky decomposition, since the fibonacci benchmark does not offer any opportunities for locality optimization.

In the results below, we abbreviate hierarchical random work-stealing victim selection policy, which attempts to steal randomly from its local socket and only steals from neighboring sockets when the local socket is depleted, to *hier*. The default socket oblivious random victim selection policy is abbreviated to *flat*. Stealing half the work policy using a pessimistic descendance heuristic is labeled *p.steal half*, and the optimistic dependence case is labeled *o.steal half*. A lock free deque is the default ready task queue implementation with no ordering constraints. A locked deque, as

we discussed in the previous chapter is a deque that is locked and has no ordering constraints by default. For the cases of optimistic and pessimistic descendance relations, a locked deque is a partially ordered set of tasks where the ordering constraint is the granularity of the tasks. For totally ordered granularity relations, we use the *sdeque* label standing for a sorted deque.

We propose imposing an order on locality relations on this chapter and argue for a case for priority queues of locality costs as ready task queues. These queues are represented with a *pqueue* below, standing for priority queues. The implementation choice for the priority queues is a binary min-heap, so the order is partial, by which we mean that the lowest cost task is known as it is the root of the min-heap and the rest of the tree follows the min-heap property. For a total ordering of priority queues, we use the label $s.p.queue$ referring to sorted priority queues.

5.5.1 Sequence Alignment

In figure 5.4, we establish a baseline by observing the cache effects of the results for the heuristics we covered in the previous chapter. The rows contain performance data for different scheduling choices — lock-free vs. locked deque vs. sorted deque, flat vs. hierarchical stealing, and steal-last vs. pessimistic-steal-half vs. optimistic-steal-half. We performed 10 measurements for each scheduling choice, and report the minimum and median values of these 10 runs in each case.

We can observe the impact of introducing hierarchical random victim selection work stealing on the number of L3 cache miss reductions on alternating rows. For unsorted deques, we observe L3 misses reduced by at least 17% where sorted deques benefit from 0% to 13% reduction in L3 misses for the median values.

The L2 miss ratio appears to drop when sorted queues are utilized, however that

		L2 data misses	L2 data access	miss ratio	L3 total misses	steals	time
lockfree deque flat, steal last	min	3961740	4146312	0.889	209030	631289	1.536
	median	4040297	4209262	0.963	247837	690540	1.564
lockfree deque hier, steal last	min	4007210	4124778	0.898	183466	895368	1.505
	median	4012403	4204810	0.956	206250	1027575	1.528
locked deque flat, steal last	min	3989099	4172655	0.949	199792	439645	1.521
	median	4065830	4234069	0.955	239945	596349	1.537
locked deque hier, steal last	min	4028405	4197683	0.933	164433	457668	1.513
	median	4078118	4221260	0.966	197755	576095	1.525
locked deque flat, p.steal half	min	4004833	4187854	0.924	233280	470282	1.518
	median	4062672	4206718	0.963	254890	580783	1.528
locked deque hier, p.steal half	min	4016425	4170637	0.904	169152	496685	1.507
	median	4084773	4240647	0.963	209467	589514	1.531
locked deque flat, o.steal half	min	4003390	4158241	0.926	220509	479642	1.518
	median	4078677	4218641	0.966	254098	629529	1.538
locked deque hier, o.steal half	min	3961523	4455386	0.836	179469	469462	1.514
	median	4027014	4499770	0.895	209721	590770	1.527
locked sdeque flat, steal last	min	3973913	4419083	0.859	200345	1203363	1.498
	median	4005206	4496290	0.888	227931	1453144	1.506
locked sdeque hier, steal last	min	3978392	4483611	0.859	178696	1295590	1.493
	median	4020066	4549422	0.886	198602	1500658	1.501
locked sdeque flat, p.steal half	min	3931867	4418742	0.870	198756	1210280	1.494
	median	3985705	4545424	0.876	220885	1343976	1.502
locked sdeque hier, p.steal half	min	3973680	4473707	0.869	179236	1237609	1.496
	median	4027519	4546635	0.884	203924	1428324	1.501
locked sdeque flat, o.steal half	min	3969911	4454106	0.873	190414	1207591	1.493
	median	3995193	4545009	0.878	209667	1323835	1.501
locked sdeque hier, o.steal half	min	3969510	4475363	0.850	183276	817220	1.494
	median	3988439	4586385	0.876	209287	1327407	1.499

Figure 5.4 : Locality implications of former heuristics on matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7

is because the maintenance of the sorted queue introduces more accesses without introducing more misses, which implies the ratio is smaller not because the dividend is smaller but the divisor is bigger.

We observe the impact of ordering tasks with ready queues based on where we their data dependence have been satisfied and how many cycles it would take to acquire those data in figure 5.5. Using hierarchical victim selection provides less benefit in this case, where the reduction in L3 misses are dropped to the range of -1% to 8%.

As in the previous figure, maintenance of partial orders seem to drop the miss ratio because the accesses increase in numbers. However, for the minimum of 10 runs cases, the L2 misses drop 0% to 2% where for the median results the L2 misses drop 1% to 2% for unsorted and 0% for the sorted ready queues. These results lead to 2% reduction for the median values in execution time when compared to the corresponding rows in figure 5.4 for partially order queues and no impact on execution time for sorted queues.

Figure 5.6 showcases the impact of pushing tasks to workers which the runtime decides to be the most local based on our aforementioned heuristics. The base case for this chart is figure 5.4 where the tasks are always pushed to the worker that enabled the task, meaning the satisfier of the last data dependence. Looking at the alternating multirows of the figure, we see that hierarchical victim selection for work-stealing leads to 4% to 10% reduction in L3 misses. The reduction in execution times in comparison to figure 5.4 for corresponding rows are between 1% and 2% for unsorted queues and 0% for sorted queues. Likely, the reductions in L2 misses compared to the corresponding base cases are between 1% and 2% for unsorted queues and 0% for sorted queues.

Figure 5.7 brings all heuristics together, so all previous results constitute base

		L2 data misses	L2 data access	miss ratio	L3 total misses	steals	time
locked p.queue flat, steal last	min	3965157	4454415	0.839	166542	359932	1.497
	median	4017174	4542283	0.884	198118	468172	1.504
locked p.queue hier, steal last	min	3938413	4507467	0.815	172085	378375	1.497
	median	3998391	4553583	0.876	188456	516626	1.503
locked p.queue flat, p.steal half	min	3983990	4544617	0.844	174217	355238	1.492
	median	4017416	4595117	0.876	205265	462273	1.503
locked p.queue hier, p.steal half	min	3980735	4489431	0.823	149785	369341	1.496
	median	4006032	4564679	0.878	190688	446628	1.507
locked p.queue flat, o.steal half	min	3957944	4455062	0.852	175971	382596	1.497
	median	4024404	4602258	0.878	204388	421792	1.506
locked p.queue hier, o.steal half	min	3961632	4492619	0.840	163716	373093	1.498
	median	4001604	4564037	0.873	193512	418829	1.505
locked s.p.queue flat, steal last	min	3976570	4441373	0.801	160734	343259	1.496
	median	4021907	4545132	0.881	190285	514355	1.504
locked s.p.queue hier, steal last	min	4006683	4500475	0.871	167119	417964	1.500
	median	4027992	4585915	0.881	193695	511459	1.504
locked s.p.queue flat, p.steal half	min	3931415	4428195	0.871	139284	332428	1.496
	median	3994173	4491047	0.888	194168	413279	1.505
locked s.p.queue hier, p.steal half	min	3980761	4529721	0.831	144066	359555	1.498
	median	4004123	4587243	0.874	191278	467018	1.504
locked s.p.queue flat, o.steal half	min	3953450	4449080	0.825	163010	354653	1.497
	median	3976404	4529913	0.882	193357	440884	1.506
locked s.p.queue hier, o.steal half	min	3994240	4494735	0.845	177097	387211	1.494
	median	4029481	4563155	0.885	191203	501298	1.501
locked s.p.queue flat, steal selfish	min	3963260	4486559	0.843	159392	363973	1.492
	median	3988234	4549839	0.876	187054	447960	1.502
locked s.p.queue hier, steal selfish	min	3976541	4456562	0.841	163387	375656	1.497
	median	4041027	4544906	0.890	185159	411942	1.500

Figure 5.5 : Locality implications of former heuristics using locality priorities on matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7

		L2 data misses	L2 data access	miss ratio	L3 total misses	steals	time
locked deque flat, steal last	min	3997770	4550288	0.872	214308	293191	1.496
	median	4032251	4585604	0.879	232257	310383	1.506
locked deque hier, steal last	min	3988617	4431366	0.799	190595	367534	1.498
	median	4027629	4489296	0.901	222678	417285	1.509
locked deque flat, p.steal half	min	3981744	4410640	0.854	199884	295292	1.496
	median	4046180	4563015	0.891	230229	345537	1.506
locked deque hier, p.steal half	min	4004453	4467461	0.838	196891	376274	1.501
	median	4035035	4553008	0.890	219524	419710	1.507
locked deque flat, o.steal half	min	3991518	4483301	0.844	194065	274180	1.496
	median	4034749	4536417	0.886	230589	301566	1.503
locked deque hier, o.steal half	min	3970344	4502150	0.846	182953	334501	1.500
	median	4021132	4567093	0.880	214258	375279	1.505
locked s.deque flat, steal last	min	4017469	4503902	0.781	198343	279381	1.495
	median	4067165	4586374	0.888	231799	330596	1.500
locked s.deque hier, steal last	min	3984854	4486738	0.877	163585	406665	1.495
	median	4042608	4546363	0.884	212981	422243	1.503
locked s.deque flat, p.steal half	min	3984251	4500767	0.862	198284	279775	1.494
	median	4036458	4611317	0.875	225320	307802	1.501
locked s.deque hier, p.steal half	min	3987264	4492472	0.815	191023	368049	1.495
	median	4034855	4551000	0.889	213751	401752	1.501
locked s.deque flat, o.steal half	min	3991462	4485367	0.851	212645	273095	1.494
	median	4027071	4585708	0.880	232130	311268	1.503
locked s.deque hier, o.steal half	min	3984717	4475191	0.837	179611	366688	1.498
	median	4037279	4505785	0.892	207987	421077	1.505

Figure 5.6 : Locality implications of former heuristics using non-local locality aware pushing on matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7

		L2 data misses	L2 data access	miss ratio	L3 total misses	steals	time
locked p.queue	min	4002473	4486906	0.826	188182	280978	1.497
flat, steal last	median	4032585	4544679	0.888	223822	341701	1.501
locked p.queue	min	3994843	4530513	0.825	181474	355502	1.499
hier, steal last	median	4046991	4571646	0.884	215473	420436	1.503
locked p.queue	min	3980370	4543591	0.829	200048	281201	1.514
flat, p.steal half	median	4041297	4557836	0.883	228236	319772	1.524
locked p.queue	min	3995541	4502688	0.814	176514	364902	1.501
hier, p.steal half	median	4031577	4570562	0.883	207313	414442	1.513
locked p.queue	min	3984171	4426682	0.832	192981	304022	1.497
flat, o.steal half	median	4036413	4537555	0.886	230228	335928	1.509
locked p.queue	min	3975385	4534347	0.870	174173	366916	1.495
hier, o.steal half	median	4031788	4596895	0.878	209522	414605	1.504
locked s.p.queue	min	4012209	4570087	0.850	200909	317256	1.495
flat, steal last	median	4048095	4594060	0.878	234579	338139	1.505
locked s.p.queue	min	3999196	4449433	0.832	193558	383307	1.500
hier, steal last	median	4032714	4530555	0.888	211706	413048	1.505
locked s.p.queue	min	4014261	4501533	0.845	189279	299224	1.495
flat, p.steal half	median	4060964	4600986	0.882	241164	338929	1.507
locked s.p.queue	min	3995269	4511145	0.826	208381	376913	1.504
hier, p.steal half	median	4031154	4572741	0.879	222634	398187	1.509
locked s.p.queue	min	4007668	4524055	0.827	189048	286846	1.497
flat, o.steal half	median	4051543	4610721	0.875	227177	325166	1.503
locked s.p.queue	min	3999160	4532694	0.835	183052	360799	1.500
hier, o.steal half	median	4033249	4574738	0.881	209958	400934	1.505
locked s.p.queue	min	4008827	4475872	0.818	200326	297538	1.501
flat, steal selfish	median	4046080	4542383	0.887	226201	340212	1.504
locked s.p.queue	min	4013419	4569530	0.829	187951	393139	1.500
hier, steal selfish	median	4042182	4603067	0.876	213074	445185	1.510

Figure 5.7 : Locality implications of former heuristics using locality priorities and non-local locality aware pushing on matching strings of size 135K with tile size 576 for the Xeon machine using OCR v0.7

lines. First, using hierarchical victim selection for work-stealing reduces L3 misses on median values 4% to 10%, as can be seen from the alternating multirows of this figure. With respect to figure 5.4, these heuristics lead to 0% to 1% reduction in the median values for L2 misses for unsorted queues and may cause 0% to 1% increase for the median values for L2 misses for sorted queues. Median execution times are reduced by 1% to 2% for unsorted queues when we compare this figure to figure 5.4. There is no discernible difference in median L2 misses or median execution times compared to figure 5.5 or figure 5.6.

The minimum result for the median values for L2 misses are achieved by using priority queues using data retrieval cost as the priority metric, with flat victim selection for work-stealing and stealing half the work according to pessimistic descendance. The minimum result for execution time is achieved by using sorted priority queues, with hierarchical victim selection for work-stealing and stealing half the work according to optimistic descendance.

5.5.2 Cholesky decomposition

Figure 5.8 establishes our base line for this benchmark by observing the cache effects for heuristics used in the previous chapter. If we compare alternating multirow in this figure, we observe an 8% to 11% reduction in the median values for L3 misses by using hierarchical victim selection.

We introduce the prioritized queues into the heuristic mix on figure 5.9. First, we observe a 0% to 6% reduction in the median values for L3 misses when hierarchical victim selection is employed for these results. When we compare figure 5.9 to figure 5.8, we see a -2% to 2% reduction for the median values for L2 misses. We also see a -3% to 3% reduction in median execution times for all cases besides the

		L2 data misses	L2 data access	miss ratio	L3 total misses	steals	time
lockfree deque flat, steal last	min	56479198	77357889	0.730	4476847	1867811	0.672
	median	65982683	87581564	0.754	5355228	2085100	0.687
lockfree deque hier, steal last	min	62515538	83291122	0.740	4204448	1832666	0.655
	median	65199230	86093914	0.754	4896860	2291411	0.676
locked deque flat, steal last	min	64664460	85706262	0.751	5311622	960043	0.684
	median	67055022	88742011	0.758	6376251	985031	0.707
locked deque hier, steal last	min	61456100	82986600	0.741	4640087	1104829	0.672
	median	66262543	86936130	0.756	5504041	1164188	0.684
locked deque flat, p.steal half	min	64169480	84811670	0.748	4271766	1005384	0.658
	median	64892687	85786820	0.763	4543411	1060247	0.665
locked deque hier, p.steal half	min	62710538	83514545	0.738	3848072	1139751	0.650
	median	64245948	84630021	0.759	4028503	1180511	0.659
locked deque flat, o.steal half	min	57268550	78528255	0.729	3701032	947945	0.656
	median	64510314	84896422	0.757	4676029	1036999	0.664
locked deque hier, o.steal half	min	61841685	83100889	0.735	3963713	1070298	0.657
	median	64442056	85215310	0.756	4370572	1106993	0.661
locked sdeque flat, steal last	min	63839807	86725741	0.736	5928481	982453	0.696
	median	67348680	88781621	0.760	6323505	1052263	0.705
locked sdeque hier, steal last	min	62740318	84731648	0.737	4706805	1044886	0.664
	median	65845420	86490305	0.759	5746650	1091283	0.684
locked sdeque flat, p.steal half	min	67400255	89395654	0.744	4208819	1025338	0.683
	median	69420265	90821130	0.765	4645269	1104725	0.691
locked sdeque hier, p.steal half	min	66918863	88678071	0.750	4202282	1086424	0.681
	median	68332968	89800004	0.761	4291617	1118648	0.686
locked sdeque flat, o.steal half	min	66141018	87365929	0.749	5817665	987715	0.697
	median	67581116	88989562	0.762	6357072	1025802	0.702
locked sdeque hier, o.steal half	min	64067359	83927905	0.726	4512483	1097382	0.672
	median	66281600	86631783	0.760	5799576	1139373	0.687

Figure 5.8 : Locality implications of former heuristics on Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine

		L2 data misses	L2 data access	miss ratio	L3 total misses	steals	time
locked p.queue flat, steal last	min	61780914	84125634	0.734	4457889	1091619	0.702
	median	65385843	87132894	0.753	5834939	1147944	1.018
locked p.queue hier, steal last	min	58339157	77625722	0.741	4864810	1212497	0.689
	median	65911044	86764557	0.755	5473246	1325933	1.133
locked p.queue flat, p.steal half	min	64802523	85702459	0.747	3915679	928730	0.670
	median	65964816	86989146	0.758	4608105	970456	0.674
locked p.queue hier, p.steal half	min	63559531	83903928	0.742	4149305	1063418	0.663
	median	66026099	86678104	0.758	4443322	1118832	0.670
locked p.queue flat, o.steal half	min	64534933	84917161	0.747	4323762	1039577	0.676
	median	65862872	87335070	0.756	5274937	1096355	0.690
locked p.queue hier, o.steal half	min	64821775	84374477	0.746	4109133	1227114	0.673
	median	65465694	86192564	0.760	5058932	1291287	0.679
locked s.p.queue flat, steal last	min	64690033	86172170	0.746	4572610	1156449	0.673
	median	66465338	87960118	0.755	4817474	1226355	0.683
locked s.p.queue hier, steal last	min	64822367	85076811	0.745	4115348	1284869	0.674
	median	66165280	87643961	0.756	4413690	1423245	0.683
locked s.p.queue flat, p.steal half	min	65061382	87223948	0.743	4161802	953756	0.666
	median	67284006	88167307	0.758	4502535	1060116	0.675
locked s.p.queue hier, p.steal half	min	65557707	85644051	0.755	4156552	1135667	0.667
	median	66528068	87272294	0.759	4321792	1183976	0.671
locked s.p.queue flat, o.steal half	min	65326322	85962536	0.755	4636417	1010767	0.673
	median	66668207	87792269	0.760	4918847	1066228	0.683
locked s.p.queue hier, o.steal half	min	64337075	84335051	0.746	4096403	1208625	0.667
	median	65677105	86928403	0.763	4926063	1327932	0.685
locked s.p.queue flat, steal selfish	min	59620955	80260647	0.730	4011046	1094238	0.684
	median	64880247	85740727	0.753	4792676	1173645	1.230
locked s.p.queue hier, steal selfish	min	51230324	69022713	0.722	3783638	1258315	0.674
	median	63603090	85434380	0.743	4595344	1333303	0.689

Figure 5.9 : Locality implications of former heuristics using locality priorities on Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine

first 2 multirows. The overhead and contention introduced by priority maintenance could not be amortized by stealing a single task, which has a 43% and 65% slowdown impact on execution time.

Figure 5.10 showcase the introduction of locality-aware pushing, where the base case is figure 5.8. Before comparing the two, the impact of hierarchical victim selection is a reduction in median values of L3 misses between 3% and 11% for all but the sorted queue, last steal case which introduces a 4% overhead. When we compare figure 5.10 and figure 5.8, we can conclude a -5% to 1% decrease in L2 miss median values and a 1% to 11% increase in execution times. Pushing to non-local locked queues introduces contention which is not amortized in these cases. We should note that the first two multirows covering the unsorted queue with stealing the last task did not reach completion and overflowed the ready task queues.

Figure 5.11 showcases all heuristics employed at once, and hence all previous figures are base lines for this figure. Initially, we observe an 11% to -7% reduction in L3 miss median values. When we compare figure 5.11 to figure 5.8, we see a 4% to 24% reduction in median values for L2 misses. The reduction in median L2 misses are between 3% to 23% when we compare figure 5.11 and figure 5.9, and lastly the differences are between 7% to 20% when figure 5.11 and figure 5.10 are compared.

As we hinted in previous graphs, stealing the last element for unsorted queues produce much worse results where the amortization does not occur. For the execution time results we discuss we are ignoring these cases. For unsorted queues the median execution times are slowed down between 27% to 49% when compared to figure 5.8, 25% to 45% when compared to figure 5.9, and 22% to 35% when compared to figure 5.10. The results where sorted queues and hence more precise locality priorities are used, fare much better. We observe an increase in execution time between 1%

		L2 data misses	L2 data access	miss ratio	L3 total misses	steals	time
locked deque flat, steal last	min						
	median						
locked deque hier, steal last	min						
	median						
locked deque flat, p.steal half	min	60499043	81908518	0.739	4028421	990277	0.683
	median	68621228	91214465	0.751	4327623	1053234	0.689
locked deque hier, p.steal half	min	61540277	83558472	0.724	3711949	1105963	0.680
	median	65064538	88078108	0.736	4017041	1240928	0.690
locked deque flat, o.steal half	min	61674559	82564150	0.732	6231571	1038679	0.720
	median	65339118	88192868	0.739	6771704	1137472	0.741
locked deque hier, o.steal half	min	62761863	84963990	0.725	5403835	1355301	0.716
	median	64154253	87726598	0.734	6144285	1380391	0.726
locked s.deque flat, steal last	min	67265768	97489140	0.687	7800891	1047056	0.768
	median	69060654	99060484	0.696	8018829	1077304	0.770
locked s.deque hier, steal last	min	68762892	98572881	0.695	7783951	1331277	0.756
	median	70384734	99459960	0.703	8364294	1378597	0.762
locked s.deque flat, p.steal half	min	67660979	91064636	0.736	4013780	978200	0.700
	median	70700554	95623657	0.742	4348411	1109258	0.704
locked s.deque hier, p.steal half	min	66082026	90693087	0.729	3931258	1147960	0.690
	median	68052118	92345223	0.736	4208880	1182714	0.698
locked s.deque flat, o.steal half	min	67054153	95792523	0.675	7322286	1032163	0.766
	median	69119164	99381007	0.693	8077716	1095612	0.774
locked s.deque hier, o.steal half	min	64714544	94158242	0.687	6803789	1264831	0.738
	median	68698480	95653589	0.719	7150612	1343203	0.753

Figure 5.10 : Locality implications of former heuristics using non-local locality aware pushing on Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine

to 4% when compared to figure 5.8, 4% to 5% when compared to figure 5.9 and -1% to 8% when compared to figure 5.10. For the unsorted queues cases(excluding the aforementioned initial two), parallelism is hampered, and the execution is slowed for the sake of locality. The reduction for the median case of L2 misses per second of execution are between 36% to 38% percent. When we look at the sorted priority queue cases, we observe that we reduce cache misses without a huge impact on execution time. For these cases the reductions of median L2 miss values per second of execution are between 7% and 23%.

		L2 data misses	L2 data access	miss ratio	L3 total misses	steals	time
locked p.queue flat, steal last	min	48338243	73111757	0.661	4375783	1212715	0.895
	median	58980442	85308167	0.686	5260438	1383337	1.908
locked p.queue hier, steal last	min	41888290	65675431	0.563	4334807	1663443	0.726
	median	50699313	76310404	0.668	4665925	1778911	1.605
locked p.queue flat, p.steal half	min	48689987	75001214	0.630	3976237	994376	0.710
	median	57236951	84844094	0.669	4548195	1077574	0.915
locked p.queue hier, p.steal half	min	38854286	67513027	0.576	4379593	1150606	0.707
	median	53919222	83479488	0.646	4880698	1203049	0.842
locked p.queue flat, o.steal half	min	44953263	70159547	0.641	5000941	1185600	0.761
	median	58160124	84083301	0.695	5348350	1364218	0.961
locked p.queue hier, o.steal half	min	37081056	65927499	0.562	4062721	1569797	0.728
	median	59203198	86078283	0.681	5082874	1693114	0.986
locked s.p.queue flat, steal last	min	43969569	74347124	0.591	4652049	1228849	0.700
	median	60682519	86979898	0.699	5324051	1364701	0.714
locked s.p.queue hier, steal last	min	44981234	76265314	0.590	4484963	1734012	0.698
	median	57825403	85588617	0.676	5199558	1862500	0.714
locked s.p.queue flat, p.steal half	min	46320490	73171112	0.612	4479574	1059834	0.696
	median	57539464	85230140	0.677	4726341	1103728	0.713
locked s.p.queue hier, p.steal half	min	40801792	72909157	0.560	4442008	1247589	0.697
	median	54900298	84206950	0.655	4848995	1358807	0.710
locked s.p.queue flat, o.steal half	min	51957987	79609772	0.640	4798314	1213034	0.694
	median	59846968	85125958	0.696	5501606	1338692	0.710
locked s.p.queue hier, o.steal half	min	53273475	79982030	0.666	4941258	1540159	0.703
	median	63858407	88980000	0.717	5320292	1642414	0.715
locked s.p.queue flat, steal selfish	min	38276318	54529787	0.702	4658843	1145322	2.161
	median	64518432	85696056	0.742	5290567	1207965	2.785
locked s.p.queue hier, steal selfish	min	59516698	78886420	0.719	4979967	1378642	2.209
	median	66170853	89969345	0.740	5467607	1404574	3.975

Figure 5.11 : Locality implications of former heuristics using locality priorities and non-local locality aware pushing on Cholesky decomposition results for a 12K by 12K matrix with 192 by 192 tiles using OCR v0.7 with Intel MKL for the Xeon machine

Chapter 6

Conclusions & Future Work

6.1 Conclusion

We should remind we stated:

> *Macro-dataflow parallelism and event-driven runtime systems offer programmability and performance benefits for applications with complex dependence structures. These runtime systems can also be extended to address new granularity and locality concerns for programs with dependence structures that are more general than fork-join parallelism.*

We propose macro-dataflow models as an intuitive way to program in order to address exposing parallelism by declaring dependences between tasks explicitly. This unconstrained parallelism does not have the tight bounds that can be achieved by models that support constrained subsets. In our former work, we built our model on top of work-sharing systems which may suffer from contention because of its centralized approach.

In this work, we built our macro-dataflow model on top of a work-stealing runtime which implies decentralized scheduling and load-balancing. First, we show our macro-dataflow approach can declare programs both with simple dependence structures without a performance penalty, and also declare complicated dependence structures and surpass hand-coded parallel libraries in execution time tuned to the specific application on chapter 3.

We observe that the underlying assumptions of work-stealing runtimes on the nature of a program's task graph do not necessarily apply to complex dependence graphs. On chapter 4, we address the granularity challenges by employing heuristics on the schedule-dependent descendance relations. We show reductions on number of steals to showcase better load balance and possible reduction in bandwidth for simple micro-benchmarks and also reduction in execution time for cholesky decomposition.

Lastly on chapter 5, we propose heuristics to the default work-stealing runtime to address locality concerns that arises with employing complex dependence structures. For a simple benchmark, we show that L3 cache misses can be reduced by employing a hierarchical work-stealing algorithm leading to a reduction in execution time. For a more complex dependence graph like cholesky decompostion, using all heuristics proposed on chapter 4 and chapter 5 we show that up to 22% reduction of total L2 cache misses with only a 4% increase in execution time. We argue that these optimizations would translate to energy savings that are becoming more and more important as the energy budget is becoming a viable constraint.

6.2 Future Work

6.2.1 Less contentious task queues

We use a locked ready queue implementation to showcase the impact of our heuristics and we observe reduction in throughput in some cases. Some of this can be attributed to workers spinning on a queue to extract or inject tasks to a queue rather than doing useful work. We intend to employ a finer grain locked version or multiple queues to reduce this effect. Additionally, steal half queues [37] and priority queues [51] have been implemented in a lock-free fashion and we are exploring their amalgamation and

the tradeoff between contention and amortized queue manipulation cost.

6.2.2 Better cache simulation

Currently, our approach is to annotate futures with caches they originate from in the runtime and do not take into account their replication and eviction. By extending our runtime with a simple cache simulator, we expect to achieve better locality results.

6.2.3 Energy modeling

We looked into possible simulators to estimate energy use but for the ones that can be collected from the user space were crude and we did not have root access on the machines we collected our results. The XeonPhi architecture does not even provide access to a set of hardware performance monitors that may provide a crude estimation. Our last chapter can be extended to address energy usage as an objective function so we can tune heuristics to optimize for energy through minimizing data-movement/cache-misses per second.

6.2.4 Topology-aware stealing

For hierarchical work-stealing, the thief is oblivious to the victim's place. This may be extended to address different stealing heuristics based on the relationship the thief and the victim share. Stealing the worse task of a worker on the same core may not lead to a better execution than passing stealing those tasks and increasing their change to be stolen by a more favorable execution unit for those tasks.

6.2.5 Distributed memory support

We believe the contributions of this work can be observed better in an environment where steal attempts and data movement are much costlier and saving these metrics have a higher impact on execution time. Saving cache misses and steal attempts did not result in drastic changes to execution times on our results but on a distributed memory architecture the costs of the heuristic introduced would be better amortized and bandwidth reduced could be better tracked. We initiated this work by introducing futures to distributed Habanero-C[22].

Bibliography

[1] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[2] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr, "Lazy task creation: a technique for increasing the granularity of parallel programs," in *1990 ACM Conference on LISP and Functional Programming*, (New York, New York, USA), pp. 185–197, ACM Request Permissions, May 1990.

[3] S. Taşırlar, "Scheduling Macro-DataFlow Programs on Task-Parallel Runtime Systems," Master's thesis, Rice University, Feb. 2011.

[4] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, May 2006.

[5] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2009.

[6] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 3.0 ed., May 2008.

[7] 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*, SPAA '98, (New York, NY,

USA), pp. 119–129, ACM, 1998.

[8] W. Ackerman, “Data flow languages,” *Computer*, vol. 15, pp. 15 – 25, Feb. 1982.

[9] J. P. Grossman, J. S. Kuskin, J. A. Bank, M. Theobald, R. O. Dror, D. J. Ierardi, R. H. Larson, U. B. Schafer, B. Towles, C. Young, and D. E. Shaw, “Hardware support for fine-grained event-driven computation in anton 2,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’13, (New York, NY, USA), pp. 549–560, ACM, 2013.

[10] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar, “Concurrent collections,” *Sci. Program.*, vol. 18, pp. 203–217, August 2010.

[11] D. Gelernter and N. Carriero, “Coordination languages and their significance,” *Commun. ACM*, vol. 35, pp. 97–107, February 1992.

[12] D. Gelernter, “Generative communication in linda,” *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 80–112, January 1985.

[13] K. Agrawal, C. Leiserson, and J. Sukha, “Executing task graphs using work-stealing,” in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, April 2010.

[14] K. Wheeler, R. Murphy, and D. Thain, “Qthreads: An api for programming with millions of lightweight threads,” in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, April 2008.

[15] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler techniques for massively scalable implicit task parallelism," 2014.

[16] D. Orozco, "Tideflow: A parallel execution model for high performance computing programs," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 211–211, Oct 2011.

[17] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 24–33, ACM, 1991.

[18] B. Liskov and L. Shrira, "Promises: linguistic support for efficient asynchronous procedure calls in distributed systems," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, (New York, NY, USA), pp. 260–267, ACM, 1988.

[19] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," *SIGART Bull.*, pp. 55–59, August 1977.

[20] G. E. Blelloch and M. Reid-Miller, "Pipelining with futures," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, (New York, NY, USA), pp. 249–259, ACM, 1997.

[21] S. Taşırlar and V. Sarkar, "Data-Driven Tasks and Their Implementation," in *2011 International Conference on Parallel Processing*, IEEE Computer Society, Sept. 2011.

[22] S. Chatterjee, S. Taşırlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with mpi,"

Parallel and Distributed Processing Symposium, International, vol. 0, pp. 712–725, 2013.

[23] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: The new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, (New York, NY, USA), pp. 51–61, ACM, 2011.

[24] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 277–288, ACM, 2008.

[25] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, (New York, NY, USA), pp. 212–223, ACM, 1998.

[26] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–12, 2009.

[27] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems," *SIGPLAN Not.*, vol. 45, pp. 341–342, January 2010.

[28] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *Proceedings of the 22Nd*

International Conference on Languages and Compilers for Parallel Computing, LCPC'09, (Berlin, Heidelberg), pp. 172–187, Springer-Verlag, 2010.

[29] "The arm cortex-a9 processors." `http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf`. Accessed: 2014-08-28.

[30] "Intel 64 and ia-32 architectures optimization reference manual." `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-%ia-32-architectures-optimization-manual.pdf`. Accessed: 2014-08-28.

[31] M. G. Ricken, *A Framework for Testing Concurrent Programs.* PhD thesis, Rice University, 2007.

[32] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, (New York, NY, USA), pp. 1–12, ACM, 1979.

[33] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, pp. 598–632, October 1989.

[34] "Ocr github repository." `https://github.com/01org/ocr`. Accessed: 2014-08-28.

[35] J. Reinders, *Intel threading building blocks.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., first ed., 2007.

[36] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance*

Computing Networking, Storage and Analysis, SC '09, (New York, NY, USA), pp. 53:1–53:11, ACM, 2009.

[37] D. Hendler and N. Shavit, "Non-blocking steal-half work queues," in *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, (New York, NY, USA), pp. 280–289, ACM, 2002.

[38] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pp. 536–545, Sept 2008.

[39] M. Wimmer, D. Cederman, J. L. Träff, and P. Tsigas, "Work-stealing with configurable scheduling strategies," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 315–316, ACM, 2013.

[40] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, (Washington, DC, USA), pp. 285–, IEEE Computer Society, 1999.

[41] R. H. Halstead, Jr., "Implementation of multilisp: Lisp on a multiprocessor," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, (New York, NY, USA), pp. 9–17, ACM, 1984.

[42] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999.

[43] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, (New York, NY, USA), pp. 1–12, ACM, 2000.

[44] G. E. Blelloch and P. B. Gibbons, "Effectively sharing a cache among threads," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, (New York, NY, USA), pp. 235–244, ACM, 2004.

[45] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: A quantitative limit study," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, (New York, NY, USA), pp. 315–325, ACM, 2013.

[46] J. Lifflander, S. Krishnamoorthy, and K. S., "Optimizing data locality for fork/join programs using constrained work stealing," 2014.

[47] T. Gautier, J. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 1299–1308, May 2013.

[48] U. A. Acar, A. Chargueraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 219–228, ACM, 2013.

[49] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers.," tech. rep., Department of Computer Science, The University of Texas at Austin, 2011.

[50] J.-N. Quintin and F. Wagner, "Hierarchical work-stealing," in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, (Berlin, Heidelberg), pp. 217–229, Springer-Verlag, 2010.

[51] I. Lotan and N. Shavit, "Skiplist-based concurrent priority queues," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, IPDPS '00, (Washington, DC, USA), pp. 263–, IEEE Computer Society, 2000.