TAPESTRY:

WEAVING EXECUTION AND SYNCHRONIZATION MODELS

by

Joshua Landwehr

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical & Computer Engineering

Winter 2013

TAPESTRY:

WEAVING EXECUTION AND SYNCHRONIZATION MODELS

by

Joshua Landwehr

Approved: _____

Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical & Computer Engineering

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Interim Dean of the College of Engineering

Approved: _____

Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

I would like to thank everyone I've known at CAPSL that helped me in some capacity: Robert Pavel, Elkin Garcia, Xiaomi An, Xiaoxuan Meng, Tom St. John, Kelly Livingston, Souad Koliaï, Stéphane Zuckerman, Jean-Philippe Halimi, Handong Ye, Yuhei Hayashi, Ge Gan, Wang Xu, Jeremy Pedersen, Joshua Suetterlein, Juergen Ributzka, Jaime Arteaga, Daniel Orozco, Chen Chen, Yao Wu, Brian Lucas, Mark Pellegrini, Chris Adamopoulus, Sunil Shrestha, Joseph Manzano, my brother Aaron, and Peggy Gao.

With special thanks to Peggy for being my friend, Stéphane for editing and proof reading my thesis, Aaron for editing and proof reading my thesis, and Juergen for buying me lunch and giving me gifts a few times.

And a very special thanks goes to my mother for wanting me to go to graduate school and supporting me thusly. Without her I would not have chosen this experience nor would I be the person I am today.

Thank y'all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With the advent of the many-core era of computing, finding parallelism has become a key battleground to the performance of computer algorithms. Traditional methods focused on providing users with synchronization primitives, standard threading models, and shared memory models. However, it was clear that these models were limited in performance. Thus, many new forms of synchronization and parallel models were designed focusing on the big three types of parallelism: data, task, and dataflow.

Nevertheless, all these models (1) only solve a particular subset of problems, (2) provide limited extendability for addressing new forms of parallelism, (3) and require a new languages with poor fine-grain performance.

As an approach to find a unified solution to these problems, Tapestry is an easily extendable portable compiler-free runtime. It is designed to quickly explore new or traditional synchronization and execution models for multi-core and many-core architectures by separating synchronization and threading models.

The main contributions of this thesis are:

1. Design of the Tapestry runtime and model to explore synchronization features or execution models that supports the mixing of all of the big three types of parallelism.
2. Proposal of extendable threaded dependency model of execution that expands traditional threads, and the creation of optimizations to extend traditional models to support finer-grain parallelism.
3. Implementation of Tapestry runtime library using only C++ 98 for 2 different platforms and 3 different operating systems with better performance and scalability than OpenMP Tasks and Cilk Plus.

Because of these aspects, my preliminary studies with Tapestry have shown that the performance and scalability of the three types of parallelism can be improved

through the use of fine-grain optimizations and careful queue design. And through other studies, I have shown the benefits of dataflow synchronization to reduce operating noise caused by thread context switching. Tapestry has been enormously beneficial in these aspects of study allowing optimizations to be easily applied across all synchronization and threading models showing significant improvements can be made to industry level parallel software such as OpenMP and Intel Cilk Plus.

Chapter 1

INTRODUCTION

Within the last decade, CPU speed improvement has slowed drastically (Figure 1.1) due to major physical barriers in design. These are summarized as the following [1]:

1. Shrinking chip geometry and increasing clock frequency cause transistor leakage current to increase. This leads to excess power consumption and heat.
2. Higher clock speed advantages are negated in part by memory latency since the performance gap between the CPU and memory has been increasing at a rate of 50% each year for decades. The memory access time simply cannot keep pace with increasing clock frequencies.
3. The traditional serial architecture is becoming less efficient as processors became faster due to the Von Neumann bottleneck: a shared bus between data and program memory limits the throughput between CPU and memory compared to memory size.
4. Finally, resistance-capacitance delays in signal transmissions are growing as feature sizes shrink.

The solution to these problems was to divide and conquer by breaking up work and distributing it across many smaller processing elements. Thus, the current trend in computing is increasing core count to add parallelism to tasks. This can be seen in AMD and Nvidia GPUs[2, 3], Intel processors and accelerators[4, 5], and CPUs by Tilera, TI, AMD, ARM, and IBM[6, 7, 8, 9, 10].

Moreover, it is clear as core count increases that future machines will be vastly different than current unified bus systems with shared memory. These new systems will utilize some hierarchy or mesh like network and will not support unified memory access among all processors on a chip due to the complexity. This as evidenced in the

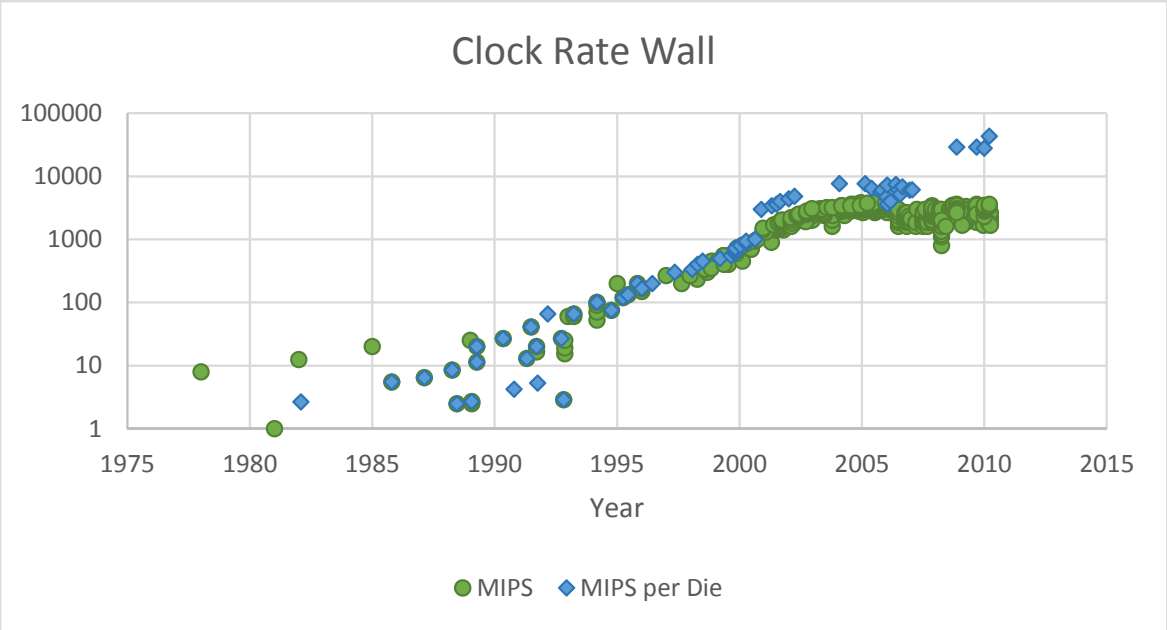


Figure 1.1: Frequency Wall: The MIPS per CPU for Intel CPUs (calculated based on publically available processor data) shows that CPU frequency which is a major contributor to CPU performance has stopped increasing at around 2005, but the MIPS per die are still increasing due to core count increasing and other optimizations such as faster interconnects.

development of Tilera iMesh technology[6], HyperTransport extensions for high node count[11], and the IBM Cyclops64 crossbar interconnect[12].

One such abstract machine that represents possible future machine design is presented in Figure 1.2. This abstract machine contrast heavily with Intel QuickPath Interconnect (QPI)[13] and HyperTransport many-core machines in the fact that it has a hierarchical interconnects whereas current Intel/AMD machines implement a hardware level point to point connection between the cores. These technologies cannot scale due to increasing complexity as core count increase. This is the reason why HyperTransport provides extensions to support core count greater than 32 on boards using special routing logic that does not guarantee unified memory access[11].

In addition, fundamental to dividing work to execute on multiple processors is the parallel execution model for these machines. Traditional models were designed for

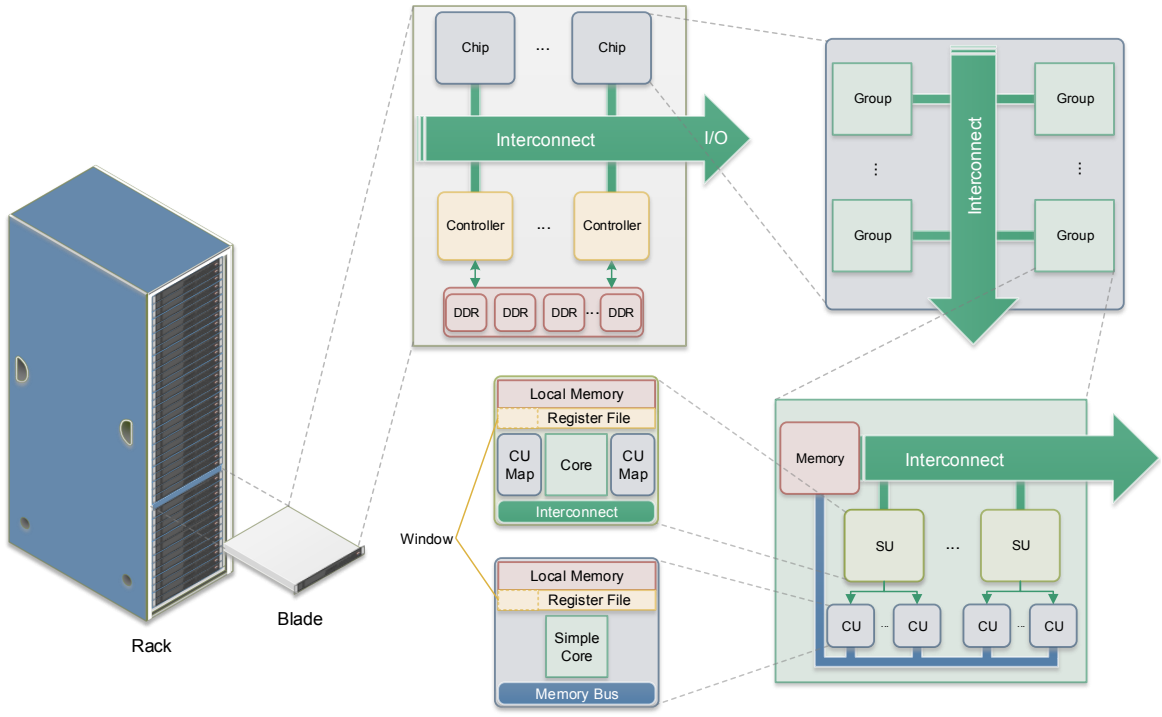


Figure 1.2: Future Abstract Machine: Future machines will most likely contain blades/nodes that are connected with some sort of switching fabric such as Infiniband. However, within a blade a central interconnect will only be connected to main memory with any number of controllers and special synchronizing units (SU) to sidestep the Von Neumann Bottleneck. These SUs are full featured processors with branching etc. Their task is to distribute work to the simpler computational cores (CUs) and synchronize with other SUs. These CUs most likely will not feature branch prediction.

serial execution environments with shared memories that had context switching. Synchronization between threads occurred via mutual exclusion on shared memory regions or with barriers. Naturally, these models could be easily extended to support systems with multiple cores and a unified memory. Nonetheless, these models required explicit synchronization by the programmer on specific memory regions and were designed with the notion of a unified shared interconnect. It is clear that these models cannot work as memory access becomes non-uniform and not available to all processors. In addition, these models do not provide a way for threads to directly synchronize instead of synchronizing on memory areas. Directly synchronizing on threads is key to providing performance and power reduction on systems via smart scheduling.

To solve these problems, this thesis introduces a C++ runtime known as Tapestry designed to explore new execution models. From Tapestry's current feature set, a model of execution is derived. This model extends traditional threads to support many-core unified bus systems with traditional memory and new systems without unified buses or unified memory access guarantees between all cores. Thus, this model of execution and framework attempts to provide a way to utilize future machines effectively without compromising performance on currently available systems.

For Tapestry to achieve this, the execution of threads is determined by the availability of input arguments to those threads. Those inputs can come from other threads and are explicitly known to the scheduler. Threads thus have inter-thread dependencies which can be used to schedule threads effectively without guarantees of uniformed memory access. In addition, for inter-core communication without shared memory, these dependencies allow for explicit point to point communication of data through arguments without the need for a separate message passing framework. Finally, the model supports fork/join parallelism which is the standard used in shared memory systems. Hence, Tapestry's model can easily support future machines without compromising current machine performance.

This is why the model is an expression of the theories of dataflow, and the framework and execution model provides many of the features of dataflow by adding them to the threading model and implicitly tying them to the threads themselves. They provide information to scheduler and allow for point to point communication.

Moreover, Tapestry aims to be a modular compiler free interface for creating and running threads with many synchronization features. It also aims to fix the shortcomings of previous models such as EARTH[14] by providing a truly portable runtime that can be executed across all types of architectures with many features. These features can be used to evaluate various execution models across various architectures.

In addition, Tapestry differentiates itself from other models of computing which implicitly handle data movement using frames or allocated memory by allowing dataflow features to be expressed as part of a thread. This means the dataflow features are not

just part of the threading model and can be handled by the underlying runtime allowing for all types of optimizations to apply dynamically. Threads that use these features become inherently tied to their data; hence, these threads are geared for fine-grain data-centric execution. Data and the computing thread are tied together in data-centric approach. Whereas, in traditional models, threads or computations are central to the execution and data are fed into those computation threads by the programmer's design as opposed, to the runtime doing this. This data-centric execution can be varied at runtime allowing for dynamic, static, or hybrid executions unlike previous runtimes. Moreover, these threads can be moved near their data and executed as opposed to moving the data to them.

1.1 Features

The Tapestry models provides a number of key advantages over the existing model of threading found in conventional operating systems for shared memory systems (POSIX Threads etc). Firstly, that standard model of threading's way to handle dependencies has a number of thorny issues solved by explicit dependencies between threads. The standard model only allows dependencies to be handled at the programmer level using either some form of mutual exclusion or by manually scheduling threads to avoid conflicts. Dependencies met with mutual exclusion require some form of preemptive scheduling either voluntary or real-time; otherwise, a deadlock can form when resources run out. Additionally, mutual exclusion of memory is not guaranteed in the future for many-core chips because keeping memory in sync between cores can be costly especially with thousands of cores. Moreover, using mutual exclusion or manually scheduling the threads would conflict with the schedule determined by the operating system or runtime.

Explicit dependencies do not have these issues. Dependency matching can be handled by the scheduler using some established protocol to send results to requested thread's input. The scheduler can use information about the arguments to determine how to adapt during runtime. For instance, it could favor locality of dependencies to

avoid long latencies. Additionally, the model does not need a preemptive scheduler to handle dependencies because only threads that have their dependencies met can run. Finally, the dependency matching occurs in the scheduler. Therefore, the user does not interfere with the scheduler. This allows the scheduler to find a more optimal schedule.

Allowing dependencies to be handled without manually scheduling simplifies the work of the programmer. The programmer does not need to handle dependency matching using explicit mutual exclusive regions nor do they need to manually handle the scheduling themselves. The programmer just needs to indicate a thread should run before another using a member function.

1.2 Challenges

A number of challenges exist by placing the dependencies in the scheduler. This adds additional information to the scheduling and is usually handled by having two queues, one for tasks waiting for dependencies to be met and another for tasks ready to execute. The two simplest ways to check the waiting queue is to do so only when a dependency is met or after a certain amount of time has elapsed. Both have varying advantages. However, it is more optimal to only touch memory when you need to. Additionally satisfying dependencies as they come allows for the scheduler to maximize parallelism by keeping the ready queue as busy as possible. But, waiting to fill tasks may give the scheduler more information about how to distribute work. Either way, when a task is ready it is moved from the waiting queue to the ready queue.

Implementing the dependency matching is another key challenge. It needs to be fast. A simple approach is to use a counter and compare it against the total number of dependencies known. This can be easily done quickly with intrinsics on current processors. However, future processors may not guarantee that in-memory atomics will be available across chips or cores. Potentially these could be available within local memory which will allow threads to utilize atomics within a certain small block of processors which in turn forces dependencies to be local. Possibly there will be some

implementation within hardware to allow dependency matching. In either case, the matching needs to be fast and perhaps have access across the whole chip.

1.3 Problem Statement

With the advent of the many-core era of computing approaching, programming computers is becoming increasingly challenging. These challenges are caused by the lack of parallelism found in the Von Neumann model of computing. The current trend of multi-core computing focuses on adding threads to this model and changes the fundamentals of programming forever. Threads in the context of current multi-core systems are designed only for task level parallelism and are only suitable for coarse grain parallelism. They even lack thread-level dependency support as described in this chapter. Threads themselves are dependency free, but the memory they access can be shared. This adds complexity to programs. Additional problems arise as threads make their move to many-core machines. Some such problems include finding enough parallelism, power efficiency of threaded solutions, speedup of algorithms, and lack of shared memory. Furthermore, no framework or runtime exists as a transitory step to move from simple multi-core systems to more complex systems with many more cores. Key to finding parallelism in these many-core systems is taking tasks and data and breaking them into finer and finer parts so that it can be run concurrently.

The objective of this thesis is to explore along these ideas, but more specifically to:

1. Facilitate an integrated solution to many of these problems.
2. Provide a solution/software to move from a shared memory multi-core system to a many-core system.
3. Expand the current idea of a thread and parallelism to include a more data-centric fine-grain approach.
4. Facilitate new ideas on thread-level dependencies.
5. Expand ideas on locality and reuse in many-core machines.
6. Explore optimizations for current synchronization and execution models.

1.4 Contributions

Runtime systems are important for both many-core systems and current generation multi-core systems. For current systems they allow expression of new programming models on top of existing operating systems. For future systems they allow for the exploration of new execution models on other systems through simulation and the evaluation of their programming model. However, current runtime systems are quite rigid in their design. They couple the features of the programming model with that of the runtime system and tie their designs tightly to a specific hardware. This is detrimental to portability and the evaluation of programming models and execution systems. Furthermore, runtime systems have a number of problems when used to evaluate specific programming execution model features. In general, programming models use a source to source compiler to translate simpler code into very complicated runtime syntax. If the programming changes, code changes must occur in both the runtime and compiler. Not only that, the programming model requires extensions to popular languages such as C or the use of pragmas.

The main contribution this thesis makes is an open source compiler free modularized runtime to allow exploration of various programming models:

- Design of the Tapestry runtime and model to explore synchronization features or execution models that supports the mixing of all of the big three types of parallelism.
  - Separation of threading and synchronization model from the operating system or runtime design. I propose a model for runtime design where the synchronization model is independently designed from the lower level design. The synchronization model can use independent code or runtime/OS level synchronization to provide synchronization features. The synchronization model can also run on top of various runtimes or operating systems.
  - Compiler free voluntary preemptive scheduling. I propose new ideas on how to handle voluntary-preemptive scheduling at the runtime level without the use of a compiler that saves stack information. This is independent of the architecture and does not use low level assembly or the application binary interface (ABI).

  - Stackable scheduling. I propose a new scheduling concept that allows new schedulers to be run on top of old schedulers. This allows the stack information and scheduling information of old schedules be saved while a new scheduler runs on top. The new schedule may favor other things than the old and run temporarily. The old schedule can resume after the new is finished.

- Proposal of extendable threaded dependency model of execution that expands traditional threads, and the creation of optimizations to extend traditional models to support finer-grain parallelism:

  - I propose a locality aware optimization that uses a lock-free and synchronization-free local queue to improve performance of fork/join algorithms with abundant work on multi-core systems.
  - I propose a locality aware optimization that directly executes threads in local stack space to improve performance of fork/join algorithms with abundant work on multi-core systems.
  - I propose a locality aware optimization that directly executes threads in local stack space and skips the runtime to improve performance of fork/join algorithms with abundant work multi-core systems.
  - I propose a locality aware optimization for fork/join that skips the scheduler on join operations and directly executes work.

- Proposal and implementation of a new high throughput input-restricted deque that is lock free and cache-aware with better scalability than locked queues and Intel Cilk Plus' queue.

- Proposal and implementation of a new methodology for compiler free work stealing with support for blocking operations that is cache-aware.

- First third party runtime to be ported to the C66X.

- Implementation of the compiler free Tapestry runtime using only C++ 98 for 2 different platforms and 3 different operating systems that is an alternative to Cilk/Cilk Plus or OpenMP Tasks with better performance and scalability:

  - A compiler free Open Source alternative to Intel Cilk Plus with comparable speed and performance that is faster than MIT Cilk without its limitations (legacy libraries can be parallelized). My runtime supports all the features of Cilk divide and conquer parallelism, but is compiler free and up to 15 times faster on x86-64. Furthermore, it has performance up to 3x faster than Intel Cilk Plus on x86-64 for less than 8 cores. In addition it has better scalability and performance for 48 cores. Finally, it can parallelize legacy code unlike Cilk.

  - New optimizations that bypass the scheduler for Cilk/ Cilk Plus divide and conquer algorithms (what Cilk is designed for) and increase fine grain performance of divide and conquer algorithms by up to 3x.

- Created C++ Thread Dependencies that have the features of codelets and EARTH with optimizations:

  - The proposal, implementation, and evaluation of data driven threads. The idea is tying work units to threads which allows the scheduling of work to be done dynamically or statically by the scheduler without the need for a compiler or programmer to handle this.
  - C++ methodology/implementation to determine thread dependency count and type based on the function signature statically. I propose a technique that uses templates, function pointers, member function pointers, and function signature information including pointers to allow for representation of arguments as dependencies or as classical arguments for C++ programs.
  - A portable compiler free implementation of codelets that support all the features of [15]. The runtime supports all the features of the execution model described by [15] and more.
  - Dynamically creating EARTH-like thread continuations to allow for code optimizations. The runtime supports continuations like EARTH. However, the runtime expands the idea to allow for threads to create them on the fly.
  - Allowing multiple continuations to be spawned from the same thread which introduces new optimizations.
  - A simple and easy way to change locality, parallelism, and load balancing by utilizing the idea of data driven threads in conjunction with the scheduler.

Using the runtime, I evaluate the idea of data-centric threads. Current thread models require the programmer to explicitly handle data movement and requires them to handle task management within a thread. This stems from the fact that threads can have a number of deficiencies in their design for expressing massive parallelism: First and foremost, if thread creation time is too large, threads cannot be used to divide work, but rather work must be divided up within the thread. This means the programmer will handle the movement of data and the scheduling of work. This is complicated in current multi-core systems and will be even more complicated as core counts increase, chip divisions become more complex, and non-uniformed memory access are taken into account. Second, preemptive scheduling causes OS noise and this

is compounded as thread count increases. In current cache based systems, as thread count increases so does cache thrashing. Furthermore, if the thread count exceeds the current system memory size, paging (retrieving memory from secondary storage) also increases. These can exponentially increase the time for algorithms with many threads to finish. Further, memory size per core on future many-core systems must likely will be very small and a preemptive schedule does not place a bound on the number of threads in memory at once. This is a huge problem. Finally, current system threads have no way to express dependencies at the thread-level. Dependencies must be handled within the thread manually by the programmer using locks or some other form of memory. Tying dependencies to the thread would give the runtime or scheduler more information to be able to better schedule threads.

This thesis proposes coupling threads, dependencies, and data to alleviate the former problems with thread design. Specifically it introduces the idea that threads are tasks of work and not just units of computation. These ideas expand on thread-level dependencies and are wholly unique.

1.5 Summary

The rest of this thesis covers all the all aspects of Tapestry from the execution model to the runtime:

- Chapter 2 provides relevant background information to help understand the model.
- Chapter 3 explains the execution model derived from the Tapestry runtime.
- Chapter 4 presents an overview of the Tapestry runtime.
- Chapter 5 provides a detailed explanation of the features of the runtime.
- Chapter 6 explains details on the lower level runtime.
- Chapter 7 contains benchmarks that compare the runtime to other runtimes.
- Chapter 8 describes all work that is related to the current runtime.
- Chapter 9 concludes the work and talks about potential future directions for research.

Chapter 2

BACKGROUND

The model of execution and framework that will be introduced in this thesis known as Tapestry has features that are the evolution of many concepts of the past. Its history can be traced to the multi-faceted idea of dataflow and its fusion with the Hybrid Von Neumann machines of the late 1980s. As the Tapestry framework is an expression of the ideas in dataflow, its future has become entrenched in dataflow's ideals. Thus, it is important to understand dataflow's history to see why Tapestry's future is rich.

2.1 Dataflow

Dataflow was a departure from many of the ideals of Von Neumann architectures (unified bus assumption with a single processor). In dataflow architectures there is no program counter, and the execution of instructions is solely determined based on the availability of input arguments to those instructions. These principles came about as reaction to the shifting trend toward multitasking in the late 1960s. Dataflow's features were primarily designed to provide abundant parallelism for multitasking. Despite providing many new strengths, dataflow had a number of weaknesses.

2.1.1 Historical Perspective

The theory of dataflow language execution stemmed from the need for parallelism which was fueled by the trend of multitasking in the late 1960s [16]. This need, coupled with the desire to represent programs in a simplified manner laid the bases for dataflow's features [17]. Dataflow in general is a computer architecture model that is painted as heavily contrasting to the popular and now ubiquitous Von Neumann

architecture model. The three main features of dataflow are: 1) no program counter or global store, 2) the execution of operations is driven only by the availability of input arguments, and 3) programs are represented by graphs.

The first model of dataflow was known as static dataflow and included only pipelined parallelism, but overtime the exchange of ideas fortified a model with maximum parallelism further progressing it into what is known as dynamic dataflow. The model began its infancy in the late 1960s and grew from the independent, but seminal works of Karp and Miller [18], Rodriguez[19], and Adams[20]. However, the idea of data-driven computations was first introduced in the original version of Karp and Miller's paper in 1964 according to Jack Dennis [21].

In 1972, the first version of the data flow language was introduced by Jack Dennis [22]. This language brought together the features of many older languages and schemas that used similar ideas, but was wholly unique in expression of Jack Dennis' ideas for parallelism. A subsequent static dataflow architecture model blossomed in 1975 to execute such a language. As a result, this solidified the static model as fundamental to the theory of executing of data flow languages [23]. That same year, Arvind introduced a new way to interpret dataflow languages which was crucial to the development of dynamic dataflow [24]. The 1980s saw further progression of these ideas. Finally, in the mid 80s and early 1990s some dataflow machines were produced [25].

But due to the poor performance of dataflow machines in comparison with Von Neumann ones, only the remnants have lived on in current architectures. However, with the development of massive many-core machines with thousands of cores, the idea has been resurrected and fused with traditional Von Neumann architectures.

2.1.2 Features

Although dataflow's past is checkered, it still is important to understand the unique benefits of parallelism it provides through its features. Dataflow contrasts heavily with control flow. Dataflow focuses on the movement of data through a program and the relations of that data to different components of a program. In general in

dataflow, multiple components can be active at once taking in data or producing it. In contrast, control flow focuses on which component of a program will be executed at a certain time. Order is important in control flow programs. Whereas, in dataflow components execute once they are ready and in no particular order.

Dataflow's concept relies heavily on graphs to represent its features. A dataflow graph is a direct graph with vertices called actors and edges called arcs. Arcs leaving a node are called the output arc of that node. Arcs coming into that node are known as input arcs to that node. At the end and beginning of a program graph there will be arcs that are not connected to any actors. These are the input and output arcs of the program graph.

The most fundamental component of dataflow is the actor. The actor is a single computation which takes in data and produces new data. Actors are similar to instructions in the Von Neumann architecture. However, actors only have a partial order determined by their dependencies. An actor can be thought of as the most basic unit of computation and thus cannot be any smaller. Any number of actors can be executed at once as long their dependencies are met. Dependencies are created by connecting outputs of one actor to the inputs of another. Data moves across these connection arcs and is known as tokens.

Dataflow is a natural, simple, and a generic way to represent a program. Programs are naturally represented by their dependencies and are not written in a specific language, but represented by a dataflow graph. This graph is generic to all the various programs across any dataflow machine and has seven basic actors. These are the:

- N-ary function - Computes a function with N inputs.
- Unary function - Computes a function with 1 input.
- T-gate and F-gate - The input token moves to the output once a T or F token is present.
- Deterministic Merge - Depending which T or F token is present on the control line, the token on the T input or F input is correspondingly moved to the output.
- Nondeterministic Merge - Moves inputs non-deterministically to the output.

- Split node - Duplicates input tokens to the output.
- Switch node - Moves the input token either to a T output or F output depending on what token is on the control line.

Dataflow programs can be composed of any number of these actors. To execute a program, these actors need rules. The rules are simple: as long as an actor instance has all its inputs and no tokens on its output it can execute. The actor is an enabled state when this is true. Once the actor instance executes and produces a new output token, it is said to have fired. These rules are the fundamental two general classes of dataflow: static and dynamic. In static, two tokens cannot occupy the same arc at the same time. Thus, when firing in static dataflow, the output arcs of a token must be empty so they have space for the new tokens. In dynamic dataflow, multiple tokens can be present per arc and colors are used to differentiate. Each color represents a different execution of the same instructions. Tokens waiting must have the same color to allow an actor to fire.

There are special firing rules for merge actors. If there is any input and no output on an nondeterministic merge actor, it can fire. Similarly for the deterministic one, the actor may fire if there is a boolean token on the switch line and the corresponding input line has a token. These special rules allow dataflow to support conditionals and loops.

2.1.3 Static Dataflow

Static dataflow follows all the rules mentioned above: one token per arc and one instance of the same actor firing, but it has a simple and easy way to interpret a dataflow graph through its rules about graph instancing. These rules however limit its potential parallelism because a section of the graph that is executed repeatedly such as in recursion or with a loop execution cannot execute in parallel.

In the static dataflow interpretation, only one instance of a graph ever exists, meaning that each node of a graph is unique and cannot have multiple copies of the same node running. Because of this, recursion and function calls are challenging to

implement. In general on most recursion, if the same instance of a graph is used for the recursive call it can result in a deadlock. However, this is not the case if the algorithm is implemented with tail recursion. Function calls also have the same problem where if two calls occur simultaneously it can result in a deadlock.

The limit on instancing of actors (one instance per actor can only be running) also reduces the amount of parallelism available for loops, recursion, or wherever the graph needs to be reused. Because parallelism is inherently found in dataflow, it is essential for these areas to execute fast. Two different solutions have been proposed to limit these problems:

- Pipelining the execution of the actors.
- Duplicating the graph.

For pipelining, the graph of code needs to be organized so that branches are the same size. In addition, it must be cycle free. Data should be available and ready so that the iterations may execute one after another as quickly as possible. This is called pipelined parallelism. It is the responsibility of the architecture or compiler of the dataflow language to guarantee a mapping that will result in this. Otherwise, there will be stalls or bubbles in the program execution which will slow down its total execution time [26]. Another key aspect essential to fast executions, is to make sure that graphs are constructed in a pipelined fashion to guarantee the most parallelism possible [27]. For duplicating, subsequent iterations or recursive calls to a graph would be duplicated. However, this will work well only if the number of iterations can be determined at compile time.

These solutions put much emphasis on the compiler and rely heavily on the compiler optimizing code. Therefore, building an efficient and good compiler is major challenge of static dataflow.

Even with pipelining, static dataflow still has the major disadvantage of lacking the ability to reuse the same graphs. Hence, recursive calls cannot be implemented without duplication on graphs of known size.

2.1.4 Dynamic Dataflow

The dynamic dataflow interpretation of graphs builds off the fundamentals of the static interpretation, but allows for more parallelism. The idea stems from U-Interpreter interpretation of graphs (Arvind [24]) by associating each token with a tag or color. Dynamic dataflow creates multiple instances of a graph by allowing each actor to have many instances. The tokens are linked to each instance of the actor. These ideas provide the basis and fundamentals to create recursion and function calls. The implementation details are left to the designer. However, in general on a function call, a new token is created with a tag or color. Within that function all tokens will have that color until they are returned.

Because there are multiple instances of nodes, this means maximum parallelism can occur in loops due to the fact the same node in a loop can execute at the same time as another iteration without waiting for another instance to finish. This assumes dependencies do not exist across iterations of a loop. Similarly, recursive calls and multiple function calls can occur in parallel.

Nevertheless, in actual hardware token matching is difficult to implement because the scheme can be costly and too complex. Each token has to be matched to other tokens. This requires storing memory for every token. Furthermore, if matching is implemented with an associative search, this would require each token to be checked against all others.

2.1.5 Semi-Dynamic Dataflow

Semi-Dynamic dataflow is the convergence between the features of static and dynamic dataflow to address the shortcomings of both. Various implementations exist[28, 29, 30], but in these models the dominant idea is to use static dataflow within a function and dynamic dataflow to schedule functions. This allows for recursive level parallelism, but not loop level.

2.1.6 Problems with Dataflow

Dataflow provides very fine-grain execution, but synchronizing is not free[31]. As the number of actors synchronizing increases, so does the cost. The cost can be high for such fine-grain synchronization. Most instructions take two pieces of data, and that means there is on average two synchronizations per instruction which can be quite costly.

Another complex issue with dataflow machines is their design for parallelism. Sequential code must execute in order using dependencies. Each instruction is dependent on the other instruction in the code, which causes them to be issued into the execution unit one at a time as they wait for their dependencies to be met. If the processing pipeline is N length and each takes one cycle, then each instruction will have to wait a minimum of N cycles before executing. This means the processor utilization is 1/N and the execution will take around N times as long compared to a conventional Von Neumann machine.

Lastly, dataflow machines do not exploit locality of data or locality of reference effectively. In conventional machines, data is short lived. t is consumed then discarded. So, conventional machines will exploit this by having a few set of registers that are quite fast because they are stored close to the processor unlike conventional memory. When the program is executed, most data is stored and used within these registers because its life is short term. Furthermore, caches are added to aid in the reuse of larger sets of data and enhance code with temporal (accesses to the same location will occur close together) or spatial locality (accesses to locations in memory are close). This known as locality of reference.

In dataflow locality of data occurs between creation of a token and its use by another actor. The locality is defined by the time between the arrival of the first token and the final token to execute an actor. In empirical studies it has been shown this time is small in most cases. Therefore, dataflow would benefit from a way to do token matching using registers close to the processor and compilers. Yet, this is hard to exploit because the non-determinate order of tokens can increase their average

lifetimes. This makes it hard for the compiler to analyze and predict the lifetime of the tokens.

2.2 Hybrid Von Neumann/ Dataflow Machines

Hybrid Von Neumann machines add dataflow properties to threads. The two methodologies of doing such include combining dataflow actors into a thread or adding dataflow synchronization into the thread model. The first methodology groups dataflow actors into a more conventional Von Neumann ideology allowing synchronization to occur on the edge of groups. The actors within the group are ordered sequentially. The second methodology adds signaling to threads of a Von Neumann machine so they can synchronize much like a groups of dataflow actors in the first methodology. Both these methodologies reduce the synchronization of regular dataflow by a factor equal to the average number of instructions per thread. Also this reduces space used for these signaling mechanisms. This can be substantial in reducing synchronization costs. Furthermore, this idea increases locality of dataflow programs allowing for registers to be used within threads much like they are in Von Neumann machines.

2.2.1 The Super Actor Machine

The Super Actor Machine (SAM) was a hybrid Von Neumann/ Dataflow execution model and abstract machine. Programs for the model were designed to be written in a high level language and then translated into a dataflow graph. Actors were grouped together into super actors using an algorithm to minimize the arcs across the boundaries of actors using a translator. The actors were then translated into threads to be loaded on the machine. The machine was very much a dataflow machine and used an innovative feature called a register cache to reduce stalls in the dataflow scheduling unit.

2.3 EARTH Model

The Efficient Architecture for Running THreads (EARTH) [14] model of multithreading as the successor of SAM falls within the class of Hybrid Von Neumann /

Dataflow models. The model was designed for off the shelf components and uses a higher level language called Threaded C to translate code into the EARTH language specifications. Threads can be spawned in the EARTH model using the *invoke* keyword on a function call. These threads' frames contain signaling information to support some dataflow properties. These properties are token-like invocation of threads. The main difference with other hybrid models and EARTH is that EARTH can link function frames to other threads using a grouping called a threaded procedure. This means multiple threads under a threaded procedure will share the same function frame.

When a grouping is *invoked*, the initial function in the frame will always run. It can do two unique things: spawn new functions to run in parallel using *invoke* and setup other functions that is part of its threaded procedure frame to run after a certain amount synchronizations/signals occur from other threads. However, the running thread still needs to spawn threads to signal the frame to run the new threads. The signal information and data to be filled must be passed explicitly to other threads by the programmer and needs to be called by that thread unless a compiler is used to mitigate the programmer's responsibility for making a parallel application. Once all signals are met, the dependent function can run. Signaling can be reinitialized and allows for reuse of memory.

2.4 Codelet Model

The Codelet Model of execution [15] extends upon the basic design of EARTH. The main difference here is it allows for availability of a resource to be part of the signaling information for a thread.

2.4.1 Inspiration

Codelets draw their ideas from dataflow, EARTH, and previous work in parallelism. The execution model emphasizes a generic representation of code with dependencies via graphs much like in dataflow. These event driven graphs are heavily inspired by the execution of a generic dataflow language and borrow much terminology from

there. The model also proposes the use of various features from EARTH with software pipelining techniques leveraged from dataflow. However, the techniques described in the execution model document[15] focus only on software pipelining of instructions in classical architectures. Lastly, the concept of codelet as a list of instructions has many roots in traditional computer programs.

The Model for Codelets borrows actors, firing rules, and dependency arcs from dataflow. The codelet graph can be thought of as a subset of a dataflow graph, but semantically redefines actors and tokens. Furthermore, the model's firing rules are more explicit than dataflow, and are much more realistic because they take into account resource constraints for the availability of firing. Finally, arcs for dependencies between codelets also comes from dataflow.

Similarly, codelets borrow the idea of threaded procedures and split-phase continuations from the EARTH model of execution [14]. In addition, the implementation of the model would benefit from the features of dataflow pipelining combined with classical software pipelining. Under the codelet model, threaded procedures may be split further into codelets. This is similar to threaded procedures containing threads in EARTH. Each threaded procedure can be thought of as a task which allows for task level parallelism. Each task can run in parallel and be invoked. Further, support for suspending and continuing threads later once data is available will come in the form of split-phase continuations. Such continuations use much less space than traditional thread suspensions. In addition, the model implementation would benefit from software pipelining to increase parallelism. In static dataflow, pipelining means keep data flowing between codelets with minimal stalls in the pipeline and making sure code graphs are constructed in a pipelined manner [26, 27]. Static dataflow accomplishes this via novel compiler techniques to guarantee data is available when it is needed by actors[27]. The model is looking for inspiration in these ideas to increase the performance of programs defined in the codelet model.

Lastly, the model's actors can be thought of as similar to traditional threads, but with dependencies. These actors are much more coarse-grain than their dataflow

brethren, but should be much more fine-grain than traditional operating system threads at the kernel or user space levels to support finer-grain parallelism.

2.4.2 Features

The codelet model is heavily inspired by dataflow, but in particular dynamic dataflow. The model can be thought of as a hybrid dataflow model that is intended to run on many-core Von Neumann systems. Specifically the model takes the concept of an actor and extends it so the actor can have multiple instructions instead of one action like in dataflow. The model builds on previous work done on EARTH.

Actors can be run in parallel as long as their dependencies are met and sufficient resources are available for them to run. Actors are grouped into subgraphs called Threaded Procedures, and it is best to execute these subgraphs in parallel. Each actor is intended to run on traditional Von Neumann cores in parallel when possible. These actors are thought of as the smallest unit of computation a core can do and thus need to be scheduled appropriately.

The actors are much like traditional operating system threads. However, they are non-preemptive and have dependencies to indicate when they should run. Depending on the implementation of the model, threads may voluntarily yield for high latency applications.

Much like dataflow, programs can be represented by generic graphs with dependencies represented as arcs between the actor nodes. Data flows between these nodes as tokens. However, the rules for the execution of actors mainly differ with four stages of execution compared with the two for enabled and fired for dataflow:

- Dormant - Does not have all its input tokens.
- Enabled - Has all its input tokens.
- Ready - Is enabled and resources are available to run it.
- Firing - Has been scheduled to run.

Of these stages, mainly a Ready stage is added to indicate the system has available resources to execute the codelet. This indicates the codelet model is much

closer to a practical implementation compared with dataflow which ignores such details. The dormant stage is redundant as it just means not enabled. All these stages are dependent on token inputs and outputs. Unlike in the dataflow model, tokens are not simply data, and can represent an event. These events may correspond to a system node going down or other events and do not contain any data. Although the code within the actors can be much larger than in dataflow, special control actors from dataflow have been added to allow token level control. Mainly included in the model are the:

- The decider (the switch node for dataflow)
- The conditional merge
- T-gate and F-gate

These actors allow conditional and loop schemes to occur outside the codelets and not just inside. Conditionals and loop schemes allow any type of code to be designed using the graph model and are very important to allow generic representation of code. Without them, programs would need to be built from within each codelet, and dependencies would be handled at the graph level.

Finally the model also defines a way to functionalize a graph in similar manner that a function is defined by a set of instructions in a traditional Von Neumann architectures. The model can define a graph as a Threaded Procedure. These threaded procedures can be called within the code like traditional functions, and their graphs will be duplicated so they can run in parallel. The model indicates these threaded procedure graphs can be automatically produced by a compiler through analysis of traditional functions composed of instructions. In addition to that, these procedures may require additional constraints to run such as locality of available arguments.

2.4.3 Parallelism

Much of the design and function of the codelet model relies on parallelism. Hence, a key question for performance of the model should be: how to reduce parallelism to favor locality? It is known that dynamic dataflow produces maximum

parallelism, but on a machine with limited resources this may be overwhelming even with many cores. According to the Codelet Model paper, parallelism is a key challenge. This is very contradictory because dynamic dataflow can execute new instances of actors as long as there are no dependencies. The codelet model paper claims that software pipelining will be useful for achieving parallelism. Much like traditional methods pipelining loops can be achieved through duplication of actors for simple cases, but may require new methods.

For pipelined loop level schemas on traditional code if each iteration is analogous to the creation of a codelet, at runtime pipelining would be beneficial. This means parallelism can be a challenge. This is also a challenge for parallelism versus the benefit of keeping codelets local.

Many software pipelining papers does not address these specific challenges nor do they provide an adequate level of parallelism [32, 33, 34, 35]. The compiler framework also needs to be good enough to model accurately the many-core chip so that software pipelining is possible. The model proposes the use of Single-dimension software pipelining (SSP) [36, 37, 38, 39]; specifically the multi-core version with extensions for many-core machines [40].

Under SSP, the algorithm selects the most profitable loop, simplifies the loop nest into a 1D loop, performs modulo scheduling, and makes sure resources for parallelism are met. Under the parallel version, wherever the schedule was delayed due to the resource constraints, the delay is executed in parallel using Lamport clocks. Speedup was good for a shared memory system with on-chip local ram, but the number of threads run in parallel is only up to 140. This is quite limited compared with the number of threads available for future many-core systems. Therefore, for massive many-core systems this idea still needs to be investigated in more depth.

2.5 Runtime Systems

A number of parallel runtimes exist that decouple parallel programming from the underlying hardware to a certain degrees: Cilk, Cilk Plus, Habanero C, Habanero

Java, Intel Thread Building Blocks (TBB), Kernel for Adaptive, Asynchronous Parallel and Interactive programming (KAAPI), and the SWift Adaptive Runtime Machine (SWARM). All these runtimes use special compilers to support their features of synchronization and spawning semantics except TBB.

Cilk [41], an extension to the C language from MIT, is similar to regular threading libraries in semantic use. However, the language is much simpler. Threads can be asynchronously spawned using the special keyword *spawn* and can be waited on using the keyword *join*. The language requires the special keyword *cilk* to label functions as spawnable. The major advantage of Cilk is its simplicity and design. The runtime uses work stealing (which it popularized) [42] with guaranteed space bounds and optimal load balancing. Furthermore, the runtime cannot execute legacy code nor have fine-grain synchronization using joins. Much of these problems were later rectified when Cilk's technology was bought by Intel and created a closed source derivative known as Cilk Plus [43]. Cilk Plus adds parallel loop constructs and allows for legacy code, but at the cost of its space bound requirements.

Habanero Java [44] is a spin-off of X10 semantically and uses the features of X10 in the language with a work stealing runtime. It provides *async* and *finish* statements which are similar to Cilk's *cilk* and *join*, but the *finish* statements allow a little finer control. In addition to joining, it supports flexible fine-grain barriers (phasers) and locality aware constructs (places). Habanero C brings many of these features to C by extending C semantically. This makes the underlying runtime very similar in concept to Cilk, but with support for *phasers* and in the future *places*.

TBB [45, 46] is a C++ library that provides a task based approach to threading. In general a number of constructs are provided to make parallelism simpler through parallel loop constructs and concurrent access containers. If neither of these are proficient for the algorithm, task level creation is possible by inheriting the task class and overriding methods. This task parallelism allows a spawn and join syntax. For more advance algorithms, there is a task level continuation available (similar to EARTH's) which requires the user to explicitly pass class instance and variable information to

child threads so they can fill out the data and signal the other threads. TBB requires shared memory via the new keyword to achieve these effects. Furthermore, TBB uses a scalable memory allocator and work-stealing popularized by Cilk [47].

KAAPI [48] is based upon the features of Cilk and extends its basic design for data dependencies. However, KAAPI is a derivative of C++ with special keywords for spawning (*fork*) and a special type qualifier for shared data (*share*). Its data sharing is based upon global memory using these share keywords. The lower level API is an extension of the POSIX Threads API. KAAPI uses a dataflow-like graph to represent closures. XKAAPI makes KAAPI simpler by making it a C only library. XKAAPI allows for dataflow like dependencies by using special templates with the KAAPI++ interface in conjunction with overloading a special structure. These templates are similar to the concept of data-driven futures that are part of the new C++ 11 spec.

Finally SWARM[49], a closed source runtime provided by ET International, allows for thread spawning, joining, and signaling of other threads like in EARTH. In addition, it provides a way to continue the execution of a thread like that provided by TBB. The signaling and data movement management is explicitly done by the programmer much like that in TBB. The major advantage of SWARM is that it can run across different computer nodes via communication through TCP/IP. SWARM claims better performance than OpenMP[50] on certain applications on a shared memory system.

2.6 Architectures

Tapestry is portable and can run on three different architectures x86-64, x86-64 with cache coherent Non-Uniform Memory Access(ccNUMA), and Texas Instruments' C66X processor with three different operating systems Windows, Linux, and SYS/BIOS. In this section, we will describe the TI architecture and the AMD Interlagos architectures.

2.6.1 C66X Processor

The C66X architecture as seen in Figure 2.1 is multi-core digital signal processor that allows for general purpose computations and targets industrial automation, medical imaging, and high performance computing among other things. The 6678 in particular has eight cores and can achieve 160 GFLOPs operating at 1.25 GHZ per core. Each core has 32 KB of L1 Data Cache and 32 KB of L1 program cache which can be configured as SRAM(internal addressable memory) or cache. In addition, each core has 512KB of L2 memory that can also be configured as SRAM or cache. Moreover, there is 4MB of shared memory known as Multicore Shared Memory (MSM) separated from the cores available that can configured as shared L2, L3 cache, or SRAM. Finally, off-chip DDR3 is available for bigger programs. Only L1D and L2 within the same core are coherent. If the MSM is configured as shared L2, it is not coherent. In addition, between cores nothing is coherent. If L2 is configured as shared memory it is not coherent either. Finally, DRAM is not coherent. It is up to the user to manage the consistency themselves. The C66X has 128-bit SIMD instructions for fixed or floating point computations. The processor can execute up to 8 instructions per cycle. Floating point wise, each core can only compute 1, 2, 3, or 4 per cycle and if using double precision, 1 every cycle. Power wise it is rated at 10 watts operating at 1GHZ which puts it in the ultra low range. The system also contains a limited number of locks and has no atomics. The major challenges with this system are managing memory consistency issues and handling atomicity for data dependencies across cores.

2.6.2 AMD Interlagos Processors

The Interlagos processor is a NUMA server processor. The 6234 processor used by this thesis has 12 cores. The particular board utilized by this thesis is the has 48 cores, 4 sockets, and 8 NUMA nodes. The unique feature of the Interlagos processor is that it decouples the FPU from the cores and shares 1 FPU per 2 cores with two 128 bit wide fused multiply-add capability (FMAC) pipelines in what is known as a module. Each NUMA node consists of 3 modules or 6 cores. The advantage here, is that 1

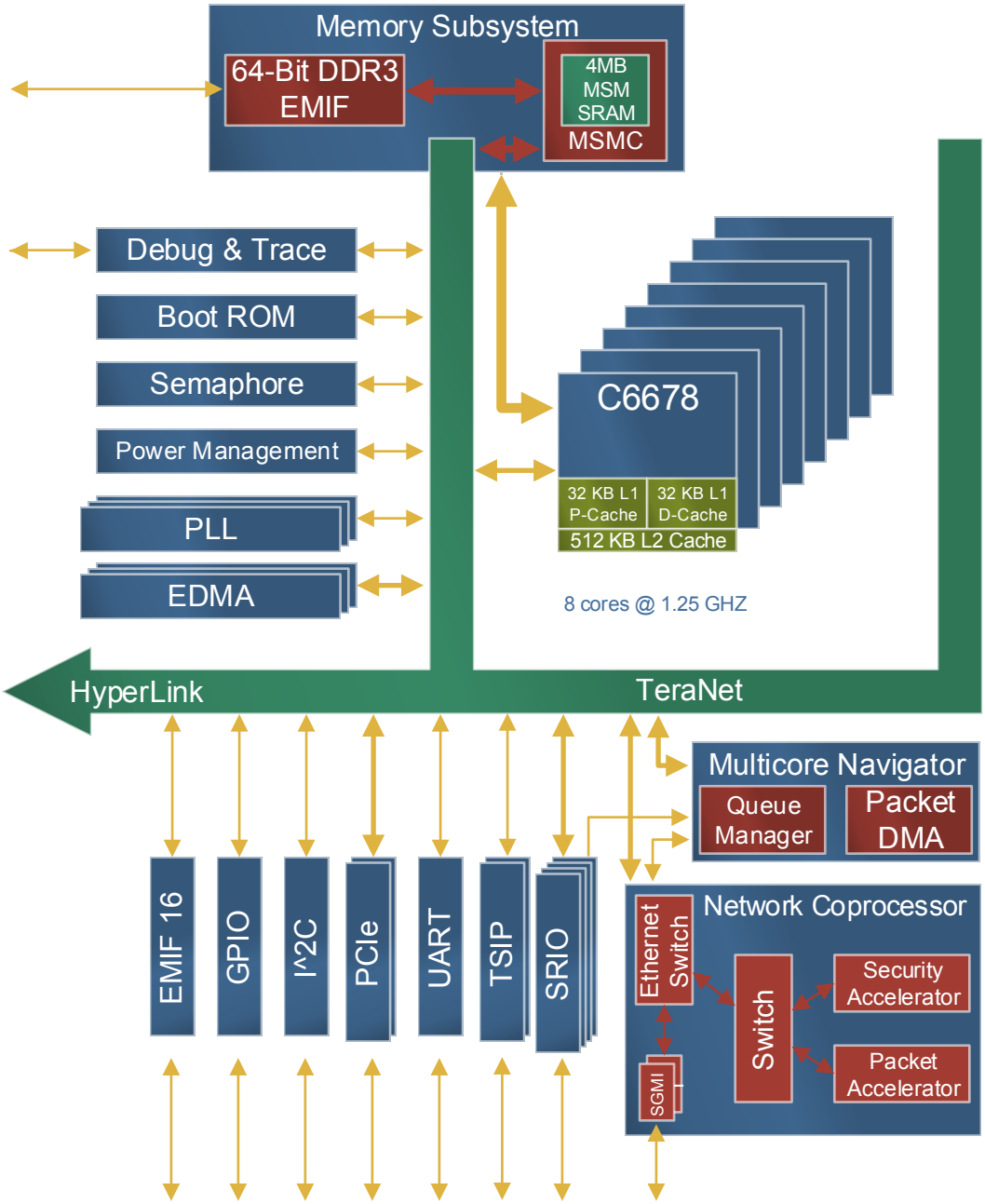


Figure 2.1: Texas Instruments C66X Architecture

core can fully utilize both FMACCS with 256 bit instructions or 2 cores both issuing 128 bit instructions. This means, the processor can execute 4 doubles per cycle for every two cores. In addition, the microarchitecture introduces new FMA4 instructions to allow for fuse-multiply additions. Using FMA4 allows the processor to execute 8 doubles per every 2 cores. The NUMA nodes on the machine are interconnected with HyperTransport[51] for a large number of processors with 6 cores per node. The interconnect configuration can be seen in Figure 2.2. Nodes connect to themselves have a distance of 1. Each connected node has a distance of 1.6 and nodes not connected directly have a distance of 2.2. This means the nodes take 1.6 and 2.2 times longer than connecting to themselves. The main considerations for providing good performance on this machine are utilizing the bandwidth provided by the 8 memories banks while balancing the low-latency of inter-node memory. In general, it is better to get more throughput, by using the 8 memory banks when utilizing all 48 cores instead of pushing all memory operations into 1 node.

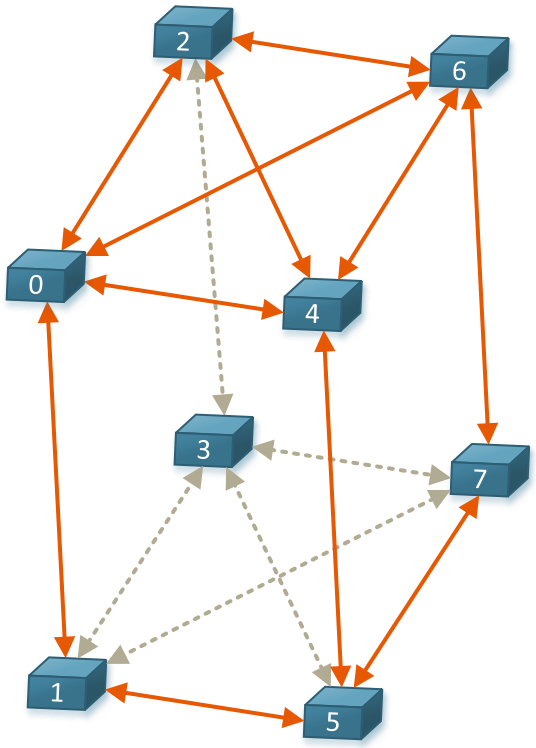


Figure 2.2: AMD Interlagos 6234 NUMA Node Distances: Nodes connected in this picture have a distance of 1.6 and unconnected nodes have a distance of 2.2. Cores are numbered in consecutive order with 0-5 being on node 0, 6-12 on node 1, and so forth. Notice, that some cores in the same chip have a longer distance than others. That is because there are 6 cores per die. So each processor has 2 dies. Those two dies are connected via HyperTransport.

Chapter 3

THREADED DEPENDENCY EXECUTION MODEL

The Threaded Dependency Execution Model (or Tapestry model) was designed to combine the features of execution on shared memory systems with that of the need to execute programs on many-core systems that may not share memory. The Tapestry model specifically allows for fork/join parallelism to be combined with dataflow-like semantics to achieve a program execution that is fork/join, dataflow-like, or a combination of the two within or across nodes by modifying the existing notion of threads to contain dependencies. The model allows programmers to tackle problems using any combination of programming models to achieve performance on any type of system.

3.1 Threaded Dependency Model Overview

Currently in computing, dependencies are described at the instruction level. If a dependency exists between two threads, they occur at the instruction level. If one thread depends on another thread the dependent thread must be manually started up after the other thread finishes.

Threaded dependencies are a way to automatically describe that a thread's results can be the input to other threads, and those threads should start only after that thread has finished copying its result into their argument list. There are number of advantages for this. First and foremost, dependencies on a thread allow for programmers to express complex dependency relationships between threads. Second, this gives the threads a partial order, and the scheduler can handle the thread scheduling instead of the programmer. Third, fine-grain parallelism can be easily expressed by breaking a serial program up in terms of threads and dependencies.

To get a better understanding of threaded dependencies, a generic model is provided in the following sections that summarizes how ideas from previous works are combined into are more versatile execution model (The Tapestry model). The model has number of features and introduces many new challenges to computing.

In the model, threads are the smallest unit of processing that can be scheduled. They can be thought of as set of instructions that are sequentially-executed (Instruction-level Parallelism may apply), can voluntarily yield execution, and be atomically-scheduled. Threads usually contain arguments to do work upon and return a value. These may be of any type. In addition, the arguments may come from other threads. Thus, these arguments can represent dependencies between threads. Furthermore, the dependencies may form loops or self-loops. Threads can also be waited on for the result of their work.

The Tapestry model combines standard threads with that of dataflow without losing the traditional semantics incorporated in threading models found in classical parallel architectures. The idea is that a thread can be thought of as an actor in dataflow. Its arguments can be thought of as arcs going into that thread or actor. Its return can be thought of as an arc leaving the thread or actor.

3.2 Actors as Threads

Actors represent a set of serial instructions and are similar in concept to that of Macro-dataflow[52] and its predecessors, but may be voluntarily suspended, form loops, may be referenced by other actors, and be joined on like in fork/join parallelism. These actors can be thought of as threads and are naturally represented by a standard function or method with inputs into them being represented as arguments. Hence, actors are represented very simply.

Thus, the basic building block of the Tapestry model is the thread or actor as seen in Figure 3.1. The thread allows for all sorts of parallelism concepts including parallel loops found in OpenMP[53], divide and conquer found in Cilk[41], the actors found in dataflow[54], loops found in TIDeFlow[54], the codelets found in Codelet

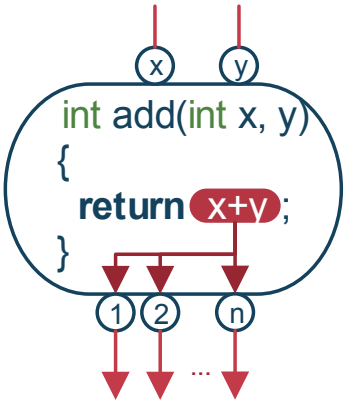


Figure 3.1: Tapestry Actor: The Tapestry actor represents a thread. Each actor is described by a function representing serial code with input arguments into that function. On return that function's results will be available to its dependent threads. An actor is enabled once all its inputs are available. However, the actor may not be scheduled to run until resources are available or if the user defined a specific constraint such as execute only on X processor.

Model[15], pipeline parallelism, and software pipelining. The state of a Tapestry actor is defined by the function it executes, the availability of its arguments, the availability of system resources, user defined constraints, and if the actor is part of a loop.

Actors in Tapestry can be created during runtime by other actors which differs completely from EARTH and TIDeFlow which are static. Furthermore, multiple graphs can be executing at once in the Tapestry model.

3.2.1 Methods

An actor represented by a method (not a function) in Tapestry requires a class stored in memory. Tapestry dictates that threads with a method executing on a specific class at a certain memory location most be only be executed where that memory location is available to that method. This means executing two methods using the same class memory across two separate physical memory locations is not possible unless a global shared memory is employed. This makes sense because allowing the memory to exist across separate nodes introduces all types of problems on copying the memory and how to update the memory copy across multiple nodes. Not only are these challenging

problems, allowing this behavior would introduce much inefficiency into method actors.

3.2.2 Merge and Switch Actors

In dataflow and the Codelet model, there are special actors best thought of as the conditional input (like a multiplexer) and conditional output (like a demultiplexer). The input actor has multiple inputs with one output. The conditional input actor only has one operation: to choose an input depending on what is on the its condition line and place the choice on its output. Similarly, the conditional output actor has one input and multiple outputs. Depending what is on the actor's condition line, the actor's job is to take the input and choose which output line to place the input on.

The problem with these actors is that they come from dataflow and only perform a few specialized operations. Most likely in conventional systems, the work they do is much smaller than the time to start and stop these actors. Hence, they have large amount of potential overhead.

To remedy this problem, Tapestry allows any actor to be defined as a conditional input, conditional output, or both. However, the firing rules for a conditional input actor are still fairly complex compared to any other actor in dataflow and much harder to represent because its condition is only used during firing (not within the body like the conditional output).

So, to simplify the work of conditional input for loops et cetera, Tapestry introduces a special conditional input actor, that allows for the condition for the next input selection to be decided within the body of the actor. Tapestry calls this the intra-conditional input actor. The actor works by being initialized to select a certain line initially. It will only fire if that line has a token on it. During firing, the actor can modify the next line selection depending on whatever conditions the programmer chooses. In Section 3.5.1, I show how a standard dataflow loop can be simplified with these notions.

3.2.3 Executing Actors

Executing actors is similar to that of dataflow: an actor can be scheduled or fired when all its input arguments or tokens are available as long as all resources and constraints are met. When an actor runs, it consumes its input arguments. An actor without inputs can be immediately scheduled. When an actor finishes execution (on function return), it will write results to its output arcs if any exist as long as it isn't continued on as another thread. The actor is then deallocated unless it is part of a loop or has references to it. A full list of executing states and conditions is described in Section 3.3.

3.2.4 Special Signals

During execution of an actor, the user may generate three special signals that may modify the resulting state of a thread on return: *continue*, *loop end*, and *exit*. The first signal, called *continue* will cause the executing actor on return to transform into new actor retaining any output dependencies. New input dependencies may be created. Because the actor is transformed into a new actor, it will no longer signal its dependents (produce tokens). This creates a split-phase transaction. The other signal, *loop end* will cause an actor that is part of a loop to stop executing on return. This signal is used for joining on program loops and allows for the memory of a Tapestry loop graph to be freed. This signal is generated by default on non-loop threads. Finally, the user can end the execution of a program with the *exit* signal.

3.3 States

An actor in Tapestry may be in one of seven states as seen in finite-state machine in Figure 3.2. These states represent how an actor lives and dies in a program execution. When an actor is defined by a programmer, it enters the *waiting* state. In this state, the actor waits until all it dependencies are met which can include resources and constraints. Once this information is provided, the actor transfers to the *schedule* state and waits to be scheduled. Once scheduled, the actor executes during the *running* state.

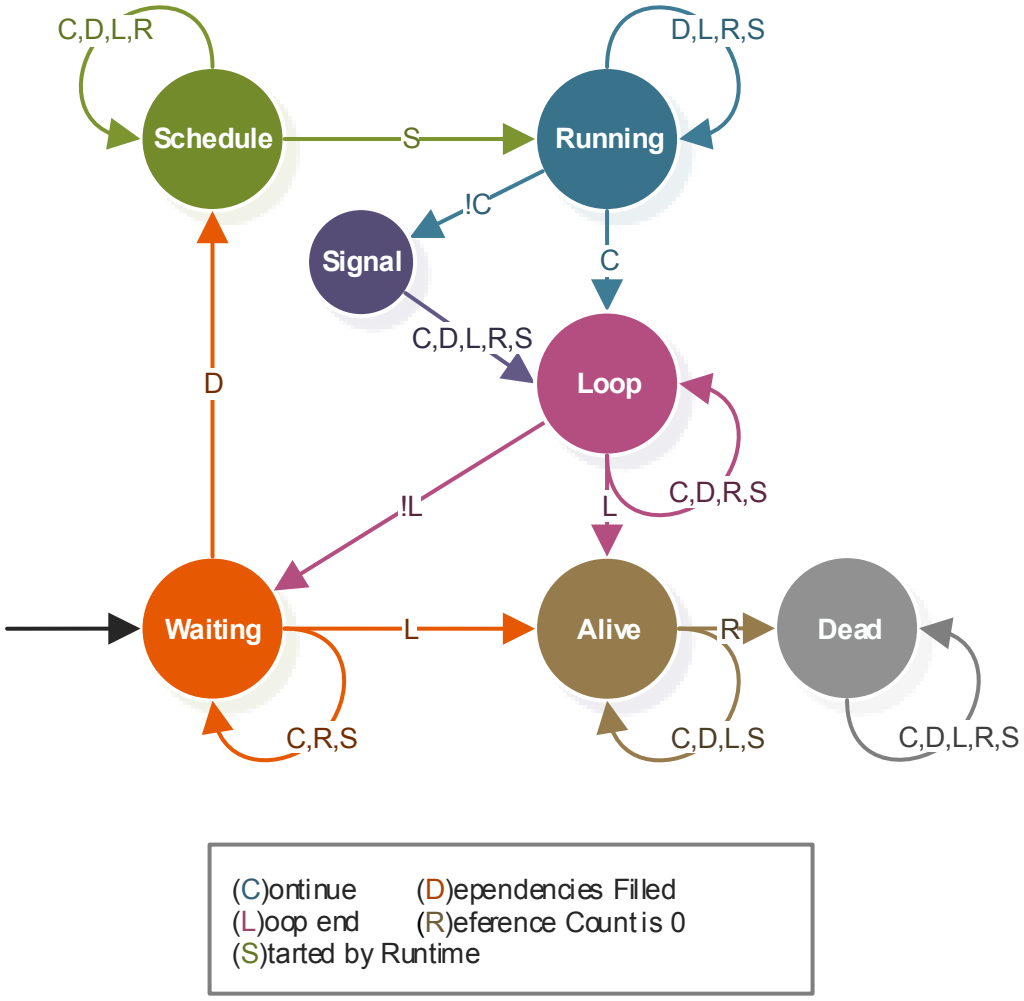


Figure 3.2: Tapestry Finite-State Machine: Tapestry actors can be in seven different states. These states allow for actors and actor graphs to be reused and decommissioned if there are no references to them in memory.

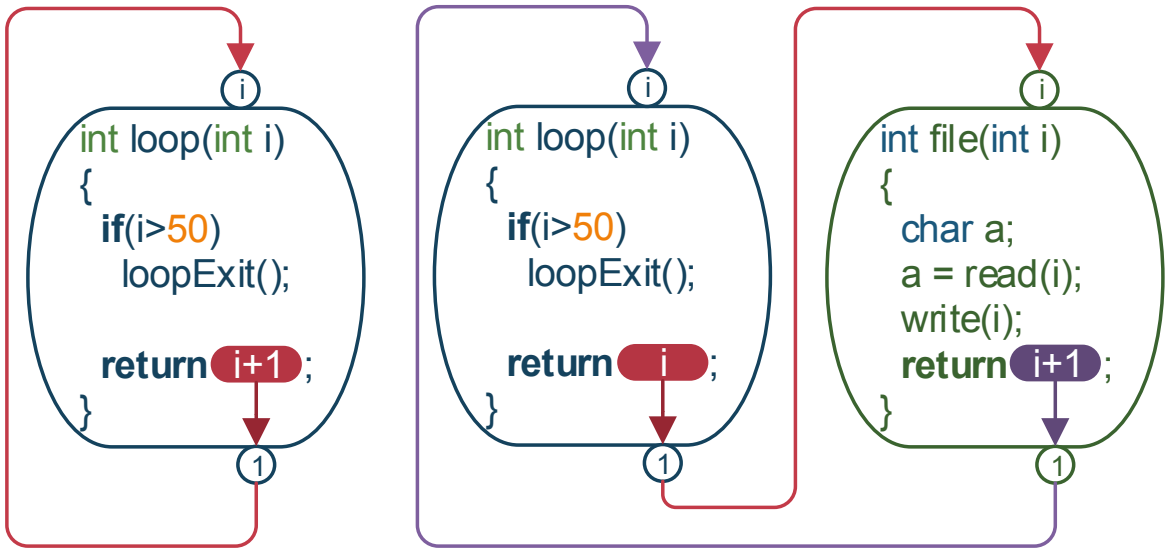
If the actor did not initiate a split-phase transaction by the time it finishes executing, the actor will enter the *signal* state and signal its dependents. Otherwise it enters the *loop* state. This state causes the actor to loop again and enter the *waiting* state if the actor or runtime did not assert *loop end*. If *loop end* is asserted, the actor enters the *alive* state which waits for its reference count to go to zero. Once this happens, the actor enters the *dead* state and can be deallocated. In the *waiting* state, the runtime can cause a waiting actor to end.

3.4 Arcs

Arcs in a Tapestry program represent dependencies between actors. These dependencies can be references, pointers, or any other data type. These types are copied between output of an actor into the input of other actors. Thus, if the Tapestry model is used on a non-shared memory system these pointers or references may not be valid. This is different from EARTH and TIDeFlow which require shared memory[14, 54]. In addition, the model can allow for multiple tokens per arc. However, this can be costly to implement. Thus, a simplified version of the model may only support one token per arc. In the one token per arc version, the stipulation is the actor must wait for each dependent actor to finish before it signals. For self-loops (Section 3.5), the stipulation is that it will always signal itself even if not finished because the actor cannot be finished. However, a race condition can occur if the self signal is not the last signal to be executed for pipelines (Section 3.6). Thus, for self-loops that initiate a pipeline, the self-signal must be the last signal executed.

3.5 Loops

Tapestry loops were designed for any type of loop construction: *do while*, *while*, or *for* loops. In addition, the ending condition of a loop can be determined at runtime just like in regular loops. This is in contrast to the TIDeFlow model that only allows for a static number of loop iterations determined at the start of a program. Furthermore,



(a) Self-Loop (b) Loop

Figure 3.3: Tapestry Actor Loop: The Tapestry model provides two basic loop constructs for a program graph. The self-loop in *a* works by causing the actor execute 50 times if the initial input is 0. This actor can have other code inside it; so, it does actual work instead of just looping. Figure *b* shows the a basic two actor *for* loop that can be generalized to have more actors.

Tapestry can have multiple loops running and supports split-phase transactions on loop actors.

The basic two loop constructs are the self-loop and a loop of actors as seen in Figure 3.3. Self-signaling has a special stipulation for the one token per arc version of Tapestry as indicated in Section 3.4. To stop a loop from repeating, the actor must signal the loop will exit. On whichever actor the loop exit signal occurs in the loop, it is the job of that actor to signal its dependents to stop waiting.

Since each actor in a loop requires input dependencies, this means the body of a standard loop will only have one actor per loop chain ever executing. Loops like those in Figure 3.3 most likely are more efficient as serial code. However like in OpenMP or TIDeFlow, the programmer can create multiple parallel actors that can execute simultaneously. In addition, the stages of the loop body do not have to barrier like in TIDeFlow.

These loops however cannot be nested with another program graph without causing them to wait for additional inputs. But, they are simple and useful for outer program loops. The next section describes how a more composable loop can be formed with Tapestry.

3.5.1 Composable Loops

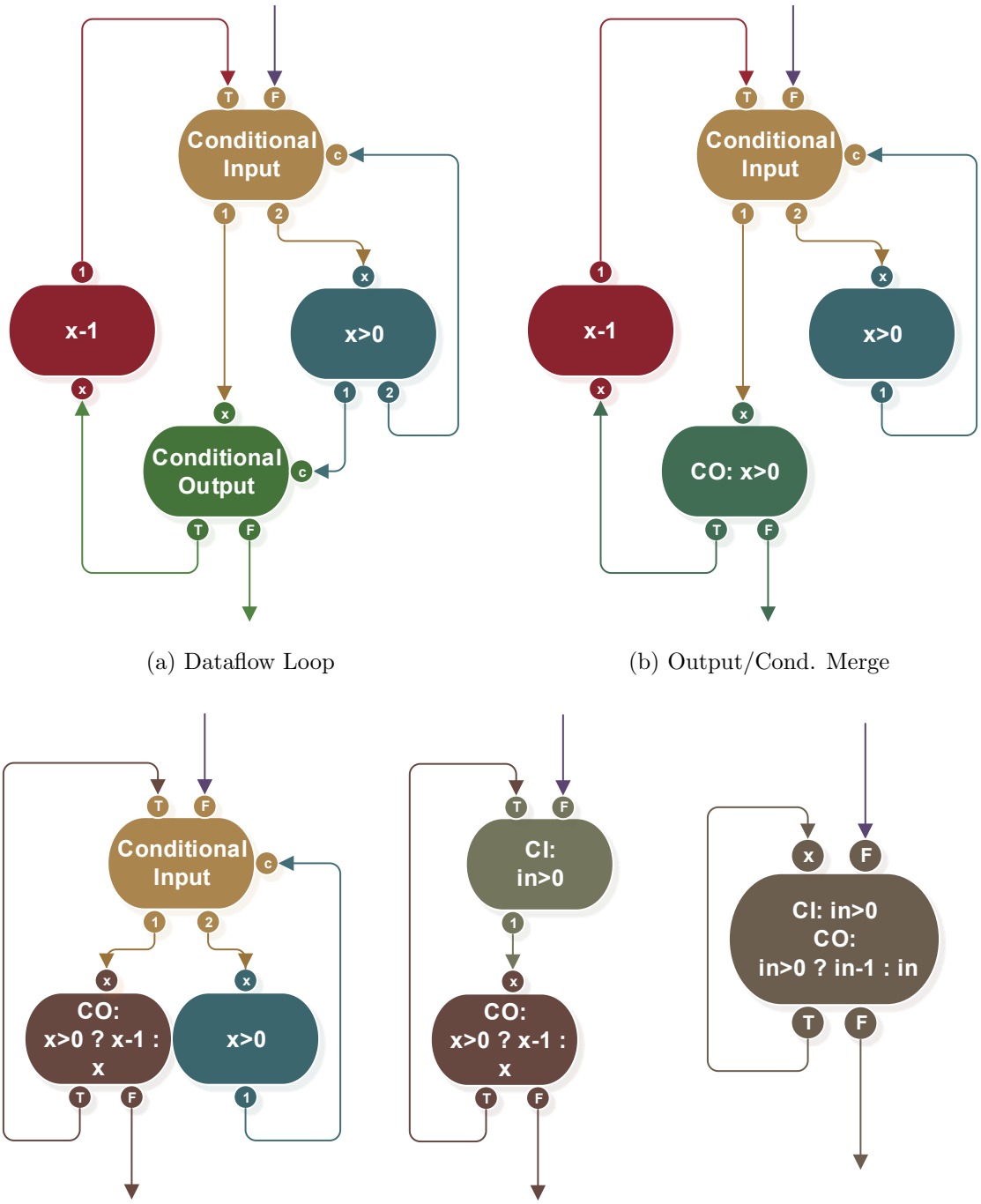
The main conditions for a loop to be composable in dataflow is that it must have a conditional input and conditional output. The problem here is that for one token inputted into a loop schema from some source must result in that loop to execute a number of times, and once the loop finishes, it returns one token to that source. This requires a conditional input into the loop and a conditional output. This means four actors are needed to achieve a classical dataflow loop as seen in Figure 3.4a. However, the condition, conditional input, and conditional output are very fine-grain and not suitable for a modern threaded systems. They have a huge overhead a reduce processor availability. Furthermore, they needlessly complicate a program graph and increase program memory requirements.

Tapestry solves these problems by allowing any actor to be a conditional input, conditional output, or both. In addition, the intra-conditional input actor can be used to further simplify things. Thus, Tapestry can achieve a composable loop with one actor as seen in Figure 3.4e. Figure 3.4 explains how a dataflow loop can be merged into one actor step by step.

The final note here is that Tapestry requires the actor when exiting a loop to use the loop exit signal. This indicates that the loop is no longer referencing any memory and may be freed if no other actor has a reference to it. In addition, this allows a programmer to join on a loop of executions.

3.5.2 Loop Nest

Tapestry supports loop nests like dataflow, but also through the simplified composable loop scheme described in Section 3.5.1. Loop nests only require one actor per



(a) Dataflow Loop

(b) Output/Cond. Merge

(c) Output/Subtract Merge

(d) Input/Cond. Merge

(e) Input/Output Merge

Figure 3.4: Dataflow Loop Reduction: Dataflow and the Codelet model require four actors to achieve a composable loop as seen in Figure a. These states can be reduced easily into one actor in Tapestry. In figures *b*, *c*, *d* each actor is merged until only one remains. In *b*, the x>0 is moved easily moved into the conditional output node since it can do multiple instructions. In *c*, the subtract is moved into the output which outputs x-1 on the T line and x on the F line. In *d*, the condition is merged into the conditional input and forms a intra-conditional input. The line F is set as default switch, and changes on output depending if the input>0. Finally in *e*, the input/output nodes are merged.

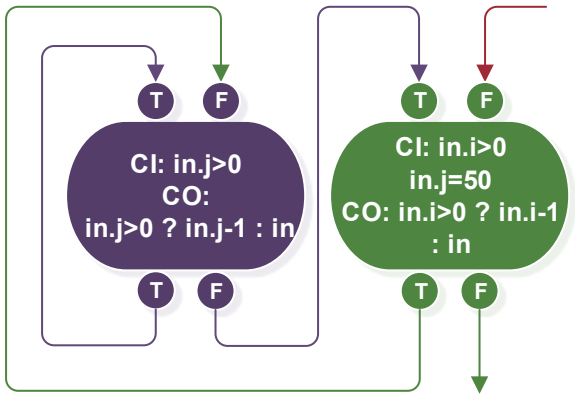


Figure 3.5: Tapestry Loop Nest: This is just a basic i/j loop nest. The input stores two ints i and j. The inner purple loop is ran 50 times for each iteration of the green outer loop. This loop nest is composable and allows for other programs or loops to nest it.

nest. In addition, the outer loop does not need to use a composable loop actor. The most basic loop nest can be seen in Figure 3.5.

Loop nests are important for many scientific applications such as stencils, matrix multiplication, and LU decomposition. Tapestry can support these within an actor or outside of one.

In addition, each loop in the nest is required to call a loop exit signal even if composable to allow memory to be dynamically freed if no references are available.

3.5.3 Software Pipelining

Software pipelining is a technique to optimize loops by reordering instructions to reduce overall execution time by allowing certain instructions to occur in parallel. Software pipelining was designed for very long instruction architecture (VLIW), but is effective in modern architectures that have some form of instruction level parallelism (ILP). Compilers have techniques to achieve automatic-software pipelining.

Similarly, thread loops in Tapestry can benefit from parallelization by allowing certain threads to be duplicated and execute in parallel. However, this can be thought of as more general problem of how to best parallelize a loop with dependencies between iterations, but applied at the thread level. It is very similar to problem faced by

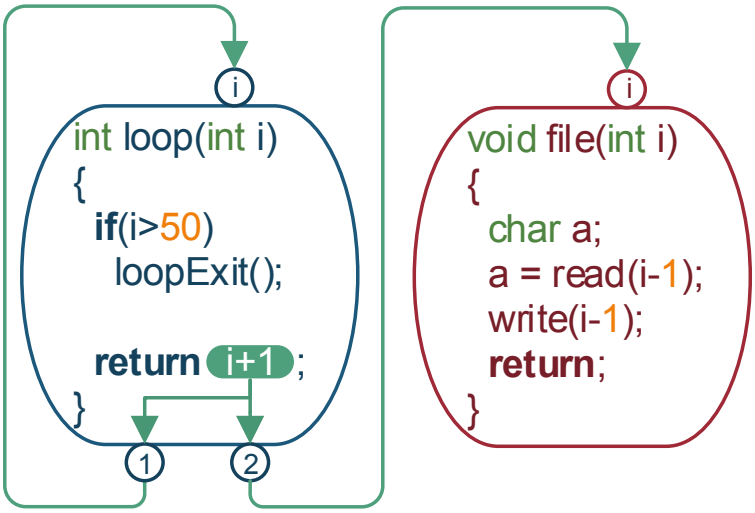


Figure 3.6: Tapestry Pipeline: This is a modification of the loop in Figure 3.3b to be a pipeline. In this case the loop actor and file actor can execute simultaneously.

OpenMP programmers when trying to parallelize loops that may contain loop-carried dependencies. Thus, the Tapestry model can benefit from parallelization of loop bodies achieved through software pipelining-like techniques.

3.6 Pipelines

An actor that is not part of a loop, but is dependent on a loop creates a natural pipeline in Tapestry. Figure 3.6 shows a simple two stage pipeline. Pipelines in Tapestry allow for true pipeline parallelism which is not addressed in the Codelet Model or TIDeFlow. In the model, pipelines allow for multiple stages of a pipeline to be active and computing in parallel.

Pipelines in Tapestry are different from the pipeline parallelism described Orozco's Thesis [54]. Contrary to TIDeFlow, Tapestry pipelines are true pipelines and do not require multiple tokens per arc to achieve pipeline parallelism. This is discussed in great detail in Figures 3.8 and 3.9. In short, the pipeline parallelism in TIDeFlow is limited by the length of time it takes to execute one loop iteration of a loop. If this time is long, more tokens need to be initially placed on the input to achieve the required parallelism.

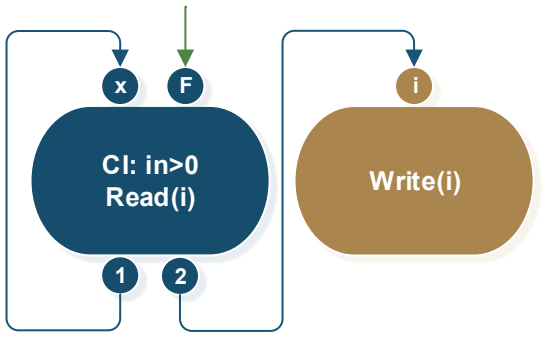


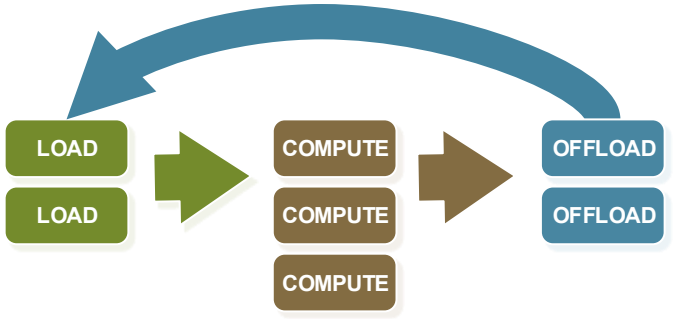
Figure 3.7: Composable Pipeline: To make a pipeline composable the input needs to modified to be a intra-conditional input. The output is duplicated to the loop and the pipeline.

Finally, pipelines in Tapestry can be made composable by using the intra-conditional input for the producer loop as seen in Figure 3.7.

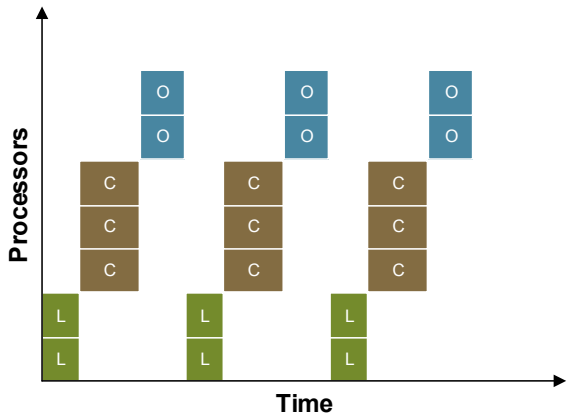
3.7 Split-Phase Transactions

Tapestry supports thread-level continuations from the EARTH model. These continuations are similar in concept to a join operation. A join operation on a thread causes a thread to suspend execution until the resulting thread has finished execution. To achieve this, the thread's stack information must be preserved. However, this can increase program memory requirements. A continuation allows the executing thread finish execution and continue on as a new thread with conditions to start that continuation (such as being dependent on a thread). This allows for join operations to be broken up into a phase of before and after a join without the need of suspending the thread and preserving its stack space.

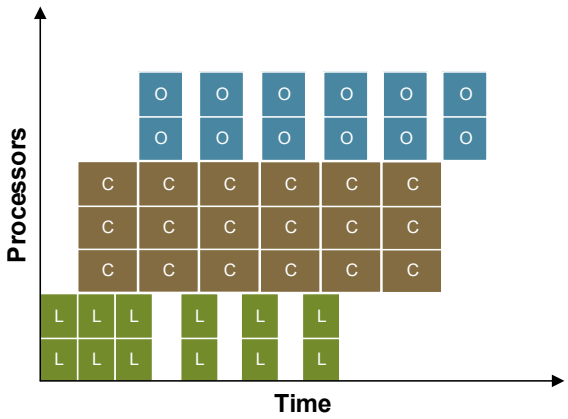
A continuation in Tapestry however has special issue to address because these continuations occur on actors that can be within a loop. So a question arises if those actors when continued will be forever continued on subsequent iterations or should the continuations be treated as a temporary phase that is part of the current executing actor? The Tapestry model takes the latter stance. Tapestry says that a split-phase continuation of an actor should be seen as one actor that has multiple temporary



(a) Loop

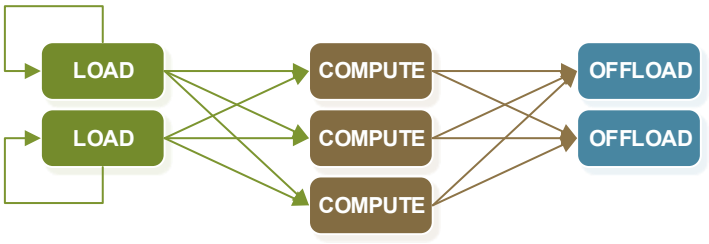


(b) 1 Token Trace

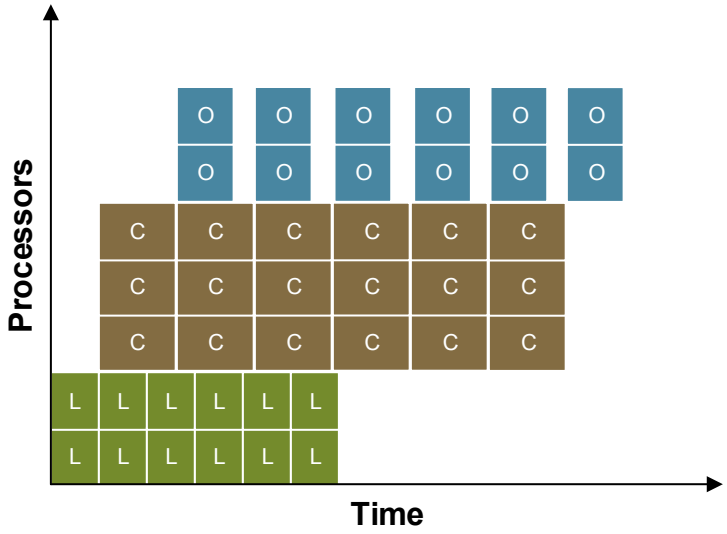


(c) 3 Token Trace

Figure 3.8: Loop Limited Parallelism: This example is taken and modified from Orozco's Thesis on Page 58 [54]. This program works in both TIDeFlow and Tapestry. The program loads some tiles, computes them, and stores their result. Notice, that the arrows represent a full barrier of dependencies between each stage. Assuming one token is inputted the parallelism of this example is quite limited as seen in the program trace in Figure *b*. Orozco points out this can be remedied by placing three tokens on the input (off-load to load edge) and leads to a form of pipelined parallelism as seen in the trace in Figure *c*. However, this requires multiple tokens per arc which are more costly to implement. In addition, the amount of tokens one needs to initialize is dependent on the total length of one full loop execution. If too little tokens are used it will lead to stalls. Thus, a programmer or compiler would have to use some heuristic to determine the number of tokens to input initially. Both current implementations of TIDeFlow or Tapestry do not support placing multiple tokens. However in Tapestry, the loop can be modified as seen in Figure 3.9 to be a pipeline and achieve true pipeline parallelism without a compiler, user intervention, or multiple tokens per arc.

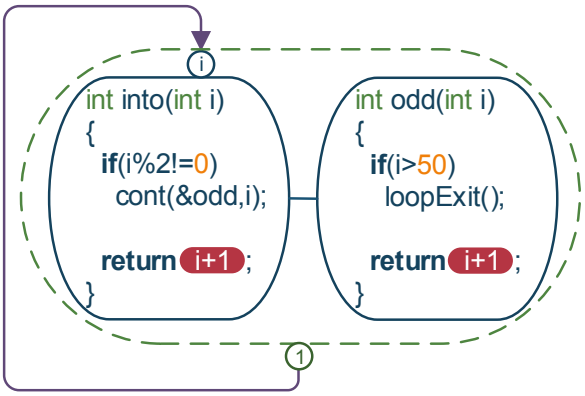


(a) Pipeline

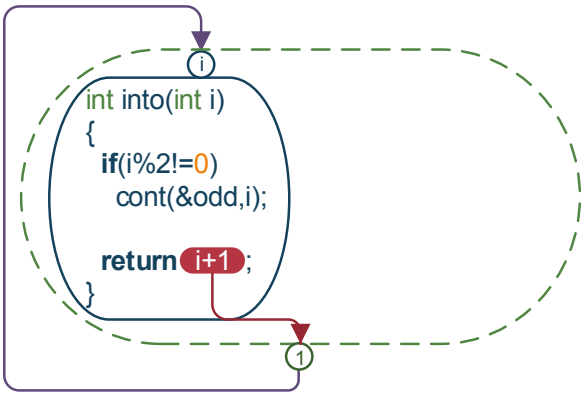


(b) 1 Token Trace

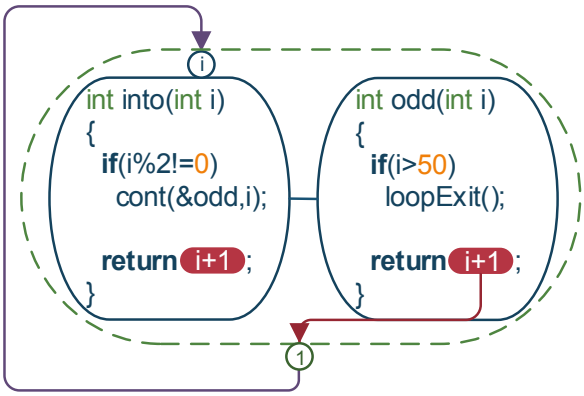
Figure 3.9: Tapestry Pipeline Parallelism: In Tapestry the loop from Figure 3.8a was modified to remove the back-edge because it limited the parallelism. Notice in Figure a, a self-loop is used to create a pipeline. The dependencies in this figure are spelled out more clearly also. Unlike in the loop in 3.8a, once the load stage completes it can immediately begin work again. In Figure *b*, the trace shows the overlapping of each stage of the program to achieve classical pipeline parallelism which does not contain the stalls seen between the loads in the 3 token loop version (Figure 3.8c). To achieve this parallelism, only one token per load is initially needed. In addition, this can be done with the one token per arc version of Tapestry and has been in the current implementation.



(a) Split-Phase Loop



(b) Even Iteration



(c) Odd Iteration

Figure 3.10: Split-Phase Transaction in a Loop: The split-phase transaction occurs depending on if the current iteration is even of odd. On a even iteration as seen in figure *b*, the loop operates normally. Yet, when the iteration is odd like in figure *c*, the into actor initiates a continuation as the odd actor and no longer signals. The odd actor completes the signaling.

phases. These phases are invoked within the actor depending on certain conditions. Figure 3.10 shows how an actor behaves during continuations during a loop.

3.8 References and Joins

The final piece of the Tapestry model is that it supports references to other threads within threads. These references may be used to read the result of an executing thread. Thus, Tapestry supports tryJoins, joins, and timed joins on these references to receive the results from a thread. These joins can be used to implement futures. Futures may be used to form an EARTH-like execution and the Tapestry model provides

that model should support continuations on EARTH-like executions with futures. In addition, joins can be used in conjunction with other operations. Finally, these threads can start other threads.

3.9 Comparison to Other CAPSL Models

The Tapestry model is more general and less vague compared to the other models introduced by the Computer Architecture Parallel Systems Laboratory (CAPSL) and its predecessors. Figure 3.11 shows the interrelationship of the models introduced by CAPSL. The Tapestry model supports traditional threading concepts unlike EARTH, TIDeFlow, or the Codelet model. These are very important concepts to achieve performance for shared memory systems. Moreover, Tapestry allows the programmer to intermix the concepts to achieve performance on any type of system. A simplified version of Tapestry is also available that does not support loops since loop dependencies can add complexity to a program and runtime. A full comparison is provided in Table 3.1.

Tapestry introduces many new concepts: such as methods, method class memory grouping, single-actor composable loops, pipelines, any-actor conditional input/output nodes, and defines how split-phase transactions work in loops. In addition, a full finite-state machine is provided that shows how the execution model can be easily implemented.

3.10 Scheduling

For the Tapestry model, the scheduler can be thought as a very abstract and general concept with one key idea: it will issue threads to each available processor. The schedule can be any schedule as long as the threads maintain a partial order as dictated by their dependencies and are suspended when data is not available from a join request. The implementation can use a centralized queue, distributed queues, or a hierarchal queues, et cetera. Figure 3.12 shows a diagram for this model.

	EARTH	TIDeFlow	Codelets	Sim. Tapestry	Tapestry
Dependencies	E	X	E	X	X
Split-Phase Trans.	X		X	X	X
Groupings (Memory)	E		E	X	X
Loops		X	X		X
Composable Loops			X		X
Split-phase in Loops			?		X
Pipelines			?		X
Any-actor cond. I/O					X
Methods				X	X
References				X	X
Join Operations				X	X
Memory Model	Shared	Shared	LC[C]	N/C++	N/C++

Table 3.1: Summary of Execution Models: The first thing to note here is that dependencies are coupled to threads or actors in TIDeFlow and Tapestry and signaling is part of the execution. Whereas, the other models have explicit signaling (E). In addition, the groupings in Tapestry are based on method memory and execute in a shared memory environment, but EARTH and codelets have threaded procedures that are more explicit. How split-phase transactions work in loops or if they do is not specified in the Codelet Model (?). Furthermore, the Codelet Model seems like it can support pipelines, but it is not specified. Finally, Tapestry only requires memory consistency within a shared memory environment. It utilizes the C++ memory model.

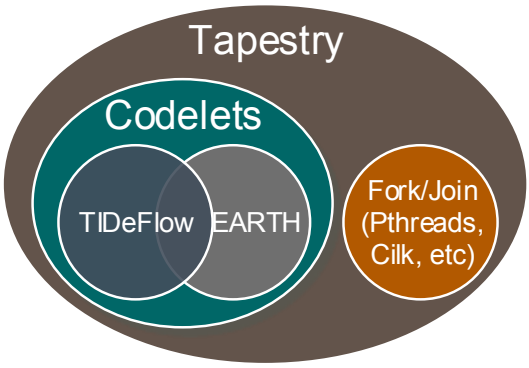
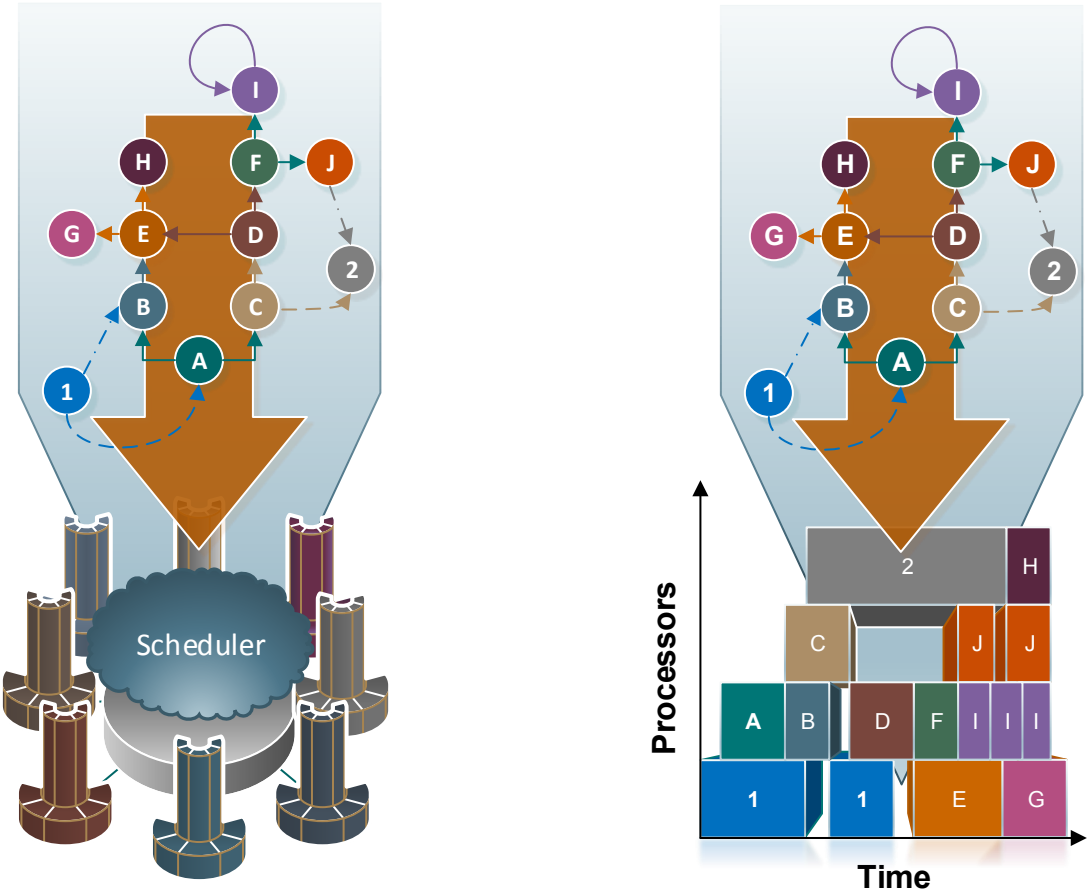


Figure 3.11: Tapestry Features: The Tapestry model subsumes all the features of the Codelet model and fork/join parallelism.



(a) Model Abstract Machine

(b) Model Scheduling

Figure 3.12: Model of Threaded Dependencies: In figure *a*, threads may form dependencies as indicated by the solid lines, spawn other threads as indicated by dash lines, and join on other threads as indicated by the dash/dot lines. This information can be used by some abstract scheduler that exists in some form across machines that may not share memory to form a schedule such as that in figure *b*. This schedule must maintain ordering of threads in respect to dependencies that come in the form of joins or arguments.

3.11 Examples

Figure 3.13 shows an example of three threads working on data without the use of dependencies. Each thread can execute concurrently as long as it is issued to the scheduler before another thread finishes. It is the job of the scheduler to determine on which CPU and when a thread is executed. A C++ API is provided to better understand: the member function `start` issues a thread to the scheduler and `join` waits for the thread to complete work. The keyword `Thread` is used to declare a variable as part of the thread class. The member variable `argument` is used to specify an argument to the thread and the member variable `function` is used to specify the function the thread will execute.

In the model, dependencies can be easily added and simplify code for the programmer. See Figure 3.14 for an example. The member function `dependsOn` is used to indicate that a variable depends on argument thread. The scheduler will handle verifying if the thread can run or not after it is issued with the start command. Once the thread finishes that dependency depends on, the dependency will execute.

```
1  void ThreadFunction ( int* argument )
2  {
3      *argument *= 5;
4  }
5
6  int main ( void )
7  {
8      Thread a, b, c;
9      int location1, location2;
10
11     a.function = b.function = c.function = &ThreadFunction;
12     c.argument = a.argument = &location1;
13     b.argument = &location2;
14
15     //Run the threads
16     a.start();
17     b.start();
18
19     //C needs to wait for a
20     a.join();
21
22     c.start();
23
24     b.join();
25     c.join();
26
27     cout << "Results: " << *location1 << " " << *location2 << endl;
28
29     return 0;
30 }
```

Figure 3.13: C++ Thread Example: Notice thread `c` must wait for thread `a` to finish before working on the data at the same location. This means thread `a` and `b` or `b` and `c` can execute concurrently, but not `a` and `c`. The programmer has to explicitly handle the dependency themselves by correctly scheduling the functions.

```
1  void ThreadFunction ( int* argument )
2  {
3      *argument *= 5;
4  }
5
6  int main ( void )
7  {
8      Thread a, b, c;
9      int location1, location2;
10
11     a.function = b.function = c.function = &ThreadFunction;
12     c.argument = a.argument = &location1;
13     b.argument = &location2;
14     c.dependsOn( a );
15
16     a.start();
17     b.start();
18     c.start();
19
20     a.join();
21     b.join();
22     c.join();
23
24     cout << "Results: " << *location1 << " " << *location2 << endl;
25
26     return 0;
27 }
```

Figure 3.14: C++ Thread Dependency Example: Code from Figure 3.13 is simplified using dependencies.

Chapter 4

TAPESTRY OVERVIEW

Tapestry is a framework for designing new execution models, synchronization features, or runtimes. The current major component of Tapestry is its framework for threading and synchronization. However, I plan to add a graphical language and simulation components to test new execution models.

4.1 Framework

The framework is divided into two components named Waft and Warp. Tapestry Waft is glue to allow for thread creation using runtimes, libraries, or low level OS system components. Tapestry Warp is the model that allows for low level or library supported synchronization features to be used in conjunction with thread level synchronizations. See Figure 4.1 for an overview of the Framework.

The aim of Waft is to use libraries or runtimes to facilitate thread creation, running, and support the features of C++ style threads. Waft can run on top of any library for threading even if the library is in user or kernel space. Waft has been successfully run on top of Pthreads, Windows Threads, Cilk Plus, OpenMP, and my user created runtime Tapestry Fibers. Waft Threads have a number of features over standard C threads including groupings, inheritance, thread context passing, and variable argument length and type.

The aim of Warp is to support threaded dependencies, asyncs, split-phase transactions, joins, futures, promises, and mutexes. Threaded dependencies are dependencies expressed at the thread level. Threaded dependencies and asyncs are used in conjunction with thread creation to support thread level synchronization. Ayncs are function calls that occur asynchronously of the executing thread. Joins are way to

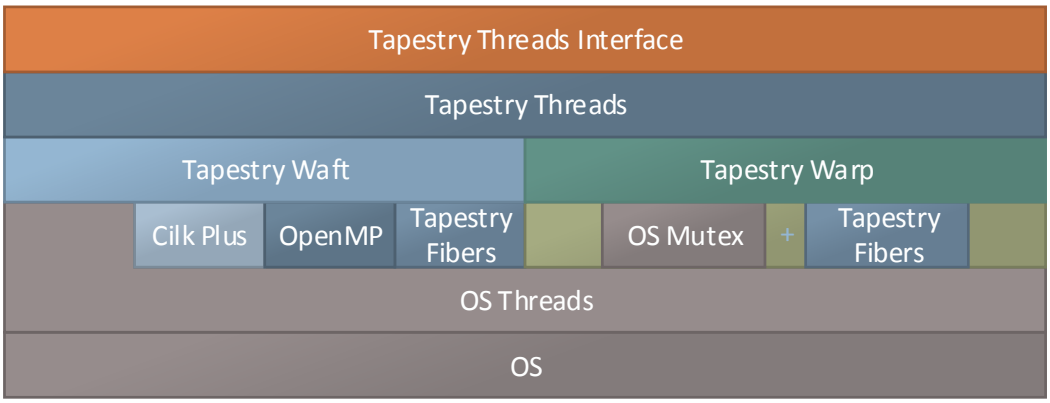


Figure 4.1: Tapestry Framework: The two major components of the framework are Waft and Warp which make up the interface for the user. Waft is for thread creation and can run on top the lower level threading libraries. Warp uses lower level mutex libraries together with Tapestry Fibers to provide new synchronization concepts such as thread level dependencies.

wait for another thread to finish execution. Split-phase transactions allow a thread to continue work after a specified condition is met. Futures are way to indicate the data will be filled in the future. If the data is used before it is filled, the thread will wait for it to be filled. A promise is way to fill a future value and indicate it can be read. Mutexes are just the standard way to synchronize access to memory which includes locks and barriers.

Extending upon the design of dependencies in Chapter 3, the framework interface is very simple. Referencing Figure 4.2b, threads are created using the `Thread` class. The constructor takes a function, context, and any number of arguments in those orders. Contexts are optional, but can be used for sharing data or to create a closure. They provide a way for memory management within threads. Contexts are created using classes as seen in Figure 4.2a. These contexts are a simple and effective way to do data management compared to Figure 3.14. They can be created in shared memory or if they are not passed to a thread, they will be constructed when the thread is run using local memory. They can be shared using the class `this` pointer.

```
1  class Storage
2  {
3    private:
4      int storage;
5
6    public:
7
8    Storage( int storeInit )
9    {
10     storage = storeInit;
11   }
12
13   void up ( int amount )
14   {
15     storage *= amount;
16   }
17
18   int get()
19   {
20     return storage;
21   }
22
23 }
```

(a) Shared Data Class

```
1  int main ( void )
2  {
3      Storage shared(7), unShared(8);
4
5      Thread a(&Storage::up, &shared, 5);
6      Thread b(&Storage::up, &unShared,
           2);
7      Thread c(&Storage::up, &shared, 4);
8
9      c.dependsOn( a );
10
11     a.start();
12     b.start();
13
14     b.join();
15     c.join();
16
17     cout << "Results: " << shared.get()
18          << " " << unShared.get()
19          << endl;
20
21     return 0;
22 }
```

(b) Tapestry Code

Figure 4.2: Tapestry Thread Example: The class Storage provides encapsulation or data hiding as a context for any Thread. This includes members to modify the internal data of the class. In the code, context `shared` is shared by threads `a` and `c`. In the running of the program, the shared context is initialized to 7. When thread `a` finishes running, the shared context will have a value of 35. At this point, the scheduler will signal `c` to run and it will use the shared context and update it to 140. Notice `c` doesn't need to wait on `a`, but `a` must run for `c` to start. Thread `b` only ever uses `unShared` which is initialized to 8 and becomes 16 by the time `b` is joined.

Lastly, the most important feature is the addition of data driven dependencies. The example in Figure 4.2 is expanded to use data driven dependencies as seen in Figure 4.3. On thread construction if all the arguments are not filled out for a function, then the remaining values will be filled in the future by another thread. Once a thread dependency is created using the `dependsOn` member function, the dependent threads argument will be filled by the thread it depends on. The value returned by the thread it depends on will be the value that fills that argument. This is one of the main features needed for data-centric threads. Of course these are just the basic extensions to the dependency scheme presented in Chapter 3. Chapter 5 describes all the features of the Tapestry Threads.

The framework is very modular and portable which will be important to future exascale design. It quickly allows programmers to explore future designs or use current features with future systems. It only requires a C++98 compiler and does not need any unique language compilers nor does it extend the C++ language. On the modular side, it separates synchronization features from thread level creation and can be connected to lower level libraries.

4.2 Wrapper Design

Tapestry employs a simple wrapper method to allow for C++ style threads, asyncs, dependencies, information hiding, et cetera. To accomplish this Tapestry creates a wrapper callback function `void * ThreadRunner( void * argument )` as in Figure 4.4 that wraps a virtual callback function on the ThreadData class. This virtual callback function is implemented by various templates which inherent the `ThreadData` class allowing for any type and combination of arguments for a thread. Tapestry allocates memory for a templated inherited thread. Then, it creates a tuple using a void pointer to that memory and the `ThreadRunner` function. This tuple is eventually passed to a thread handler class that invokes the lower level threading library such as the operating system library or some runtime.

```
1  int up ( int amount, int storage )
2  {
3     storage *= amount;
4
5     return storage;
6  }
7
8  int main ( void )
9  {
10     Thread a(&up, 5, 4);
11     Thread b(&up, 2, 4);
12     Thread c(&up, 4);
13
14     c.dependsOn( a );
15
16     a.start();
17     b.start();
18
19     cout << "Results: " << c.joinValue <int> () << " "
20          << c.joinValue <int> () << endl;
21
22     return 0;
23 }
```

Figure 4.3: Tapestry Thread Data Driven Example: Expanding upon Figure 4.2, this code uses pass by value to update the value that lives across threads. The first thing to note, is the `up` thread function takes two arguments. When creating threads such as `a`, `b`, or `c`, if the argument passed is less than the arguments of the function, the framework assumes those arguments are met by a thread dependency. In this example, `a` and `b` both do not have dependencies because all their arguments are filled when created, but `c` will get the value of storage at later time since it is empty. `c` using the `dependsOn` member function indicates this thread will be filled by `a`. Finally, the return value is what value will be passed by the function to any dependent threads. Note: the `joinValue` function just joins and returns the value returned from the thread.

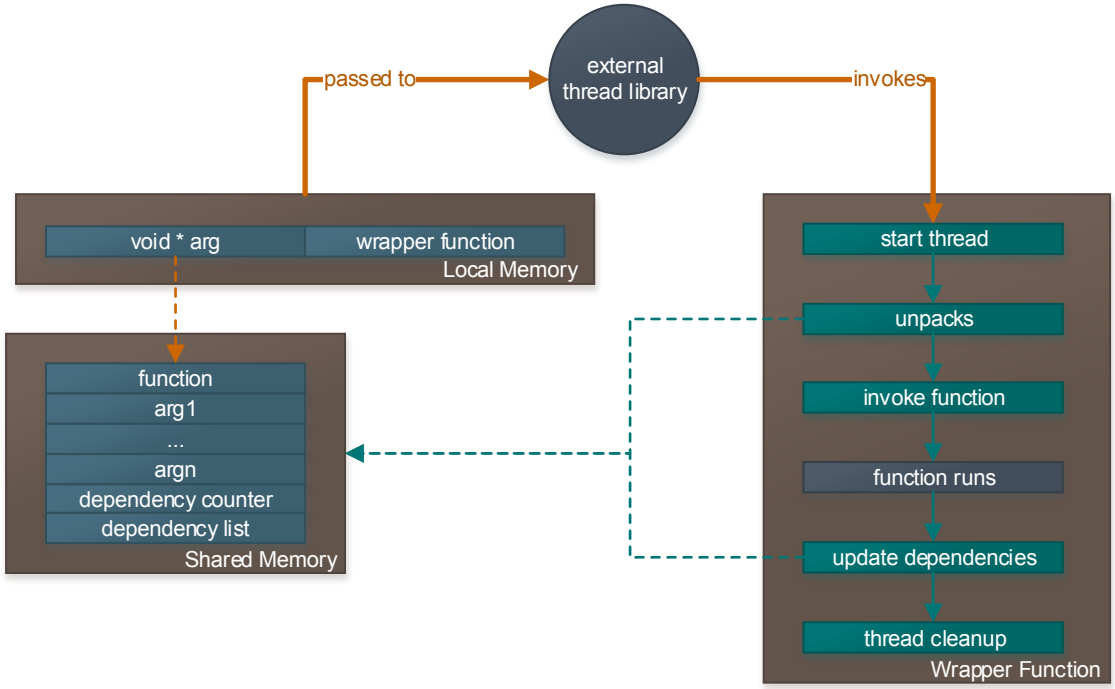


Figure 4.4: Tapestry Wrapper View: Here we see the thread memory is stored in some location such as the heap with a dependency list. Once a wrapper thread gets invoked by the external library, it will statically cast the `void *` into the `ThreadData *` class. Using inheritance, it will invoke the thread start function which will run the inherited templated class's implemented run method. Once a thread completes, Tapestry will copy the return argument from the thread into any dependencies in the dependency list and decrement their dependency counter.

Tapestry's wrapper thread allows for the implementation of all of Tapestry's features without the use of a lower level runtime. The wrapper thread will unpack and run a thread using the thread's most parent class pointer to invoke a Thread callback as an implemented virtual method. This method will run and return a result into the memory location of the thread. Tapestry will then copy the return into the resulting threads and decrement the dependency count.

Tapestry allows arguments to a thread to be filled by other threads using a dependencies as explained above. Because Tapestry allows arguments to be filled by value in combination to other threads filling them it needs a special design. To accomplish this, templates allow for partial filling of arguments at construction or

```
1 template<class R, class C, class A1, class A2>
2   Thread( R (C::*funct)( A1, A2 ), C * inst )
3 {
4   //Code here...
5 }
```

Figure 4.5: Tapestry Argument Template: In this example we see how Tapestry determines argument, class, and return types from the partial specialization of the member types on the member function pointer. Furthermore, we see that the class type from the member function pointer must match the class pointer type passed in.

```
1 template<class R, class C, class A1, class A2>
2   Thread( R (C::*funct)( A1, A2 ), C * inst, A1 arg1 )
3 {
4   //Code here...
5 }
```

Figure 4.6: Tapestry Partial Arguments: In this example we see that Tapestry through specialization on the function member pointer knows there are two arguments, but argument one is already met and thus it should set up the thread so there is only one signal needed and that signal will store data at argument two.

the user can fill them in with the special fill function. Additionally, Tapestry uses specialization on the function pointer or member function pointers passed into the `Thread` constructor to determine the argument information it needs to store and how many arguments will be met by threads as seen in Figure 4.5. If the function pointer has two arguments, but only one is passed to the `Thread` constructor Tapestry knows that one argument needs to be met by another thread as seen in Figure 4.6.

4.2.1 Optimizations

4.2.1.1 Fine-Grain Optimizations

Tapestry allows for locality optimizations when parallelism is overly abundant, and it is advantageous to begin executing the work locally. If a thread has dependencies Tapestry can skip the local queue and begin working on a thread immediately and meet its dependencies without the extra overhead of pushing and pulling from a queue. This is called Locality Optimized (LO). Furthermore, if the thread has only fork/join style dependencies like a regular recursive function call, Tapestry can bypass its wrapper

layer and begin executing the functions on the stack. This allows for even more locality and is known as Super Locality Optimized (SLO). The user can turn this at anytime or allow a lower level runtime or thread system turn this on.

For instance, Tapestry Fibers have the ability to monitor the amount of parallelism available by checking the amount of items in a thread's local queue. It then can turn on SLO or LO optimizations at runtime and significantly speedup code if a certain amount of parallelism is achieved. In addition to these locality aware optimizations, we know on fork/join parallelism that the join should be at the top of the stack. Because of this, Tapestry can pop the thread directly off the stack and execute it directly without requiring the scheduler to do this. This optimization is called Fast-Join (FJ).

Tapestry was specifically designed so that a thread regardless of dependency information could be called with the runtime or serially by a user. This allows for the user to be able to optimize their code and allows for the LO and SLO optimizations being possible including the Dynamic Continuation feature that is a contribution of this thesis. The Dynamic Continuation feature will be explained in more detail in the Chapter 5.

4.2.1.2 Other Optimizations

In addition to parallelism optimizations, Tapestry, when allocating threads, can use a built in memory manager. For recursive parallelism like in regular recursive code, Tapestry can employ a stack based memory manager with ultra low overhead and no locking due to guarantee that a thread executing will not swap out onto another core. For other types of parallelism, a random table based lock memory manager is available with various level block sizes for memory locations.

Finally Tapestry can employ thread level caching or completion. Tapestry can cache or store thread states, so when a redundant thread state is encountered Tapestry can just use the results stored. This optimization will turn tree search algorithms into a graph search by default and reduce redundant computations. This caching can occur at the thread level or throughout the whole system.

4.2.2 Tapestry

Tapestry's implementation on shared memory systems spans multiple layers. The first layer is Tapestry Warp and Waft. In Waft, Tapestry Threads are implemented using a combination of the heap and stack. The main difference compared with regular threads are:

1. Storage
2. Invocation

Storage:

Each Tapestry Thread has two layers: the data of the residing thread and context to invoke, join, and refer to a thread. Contexts or interface, are up to the user to decide where to store using the `Thread` keyword, but once a thread is created, its arguments, function pointer, class context pointer, and dependency information are allocated on the heap or via a memory manager. A reference to this data is stored in the thread interface. If the interface context is copied, the thread data stays in memory wherever it is allocated and only will ever have one copy. Threads also keep a reference count to the data to determine if the data is being used. If the reference count falls to 0, then the thread will be deallocated and returned to the heap or memory manager. The thread data is passed to the lower level thread library to run, and the thread frame is most likely allocated to the heap and has a limited stack size. The arguments for a thread are copied to on the return from another dependent thread, and the reference count is decremented during this time.

When a thread is created, memory is allocated for the thread using either a resource manager or the heap. Once allocated, the data is initialized. Variable size arguments are handled statically using templates which guarantees up to nine arguments can be passed of any size or type at the current time. Structures can be used to pack multiple arguments coming from one thread or for greater size inputs. Information about argument type are gathered from the function pointer. If dependencies are involved, space is allocated for a integer counter and another integer variable to store

the number of dependencies. Because Tapestry uses function pointers and no special syntax, legacy or library code is easily supported, and thus serial function calls are allocated using the conventional stack or thread stack. Legacy or library code, can be easily parallelized because of these feature also. The semantics for calling a function to be executed as a thread is determined by the lower layer. Usually this just means, the function is executed on that threads stack, and it returns to that thread after executing that function.

Invocation:

Tapestry only uses the lower level thread runtime such as Pthread, Win32 Threads, or a runtime to execute the threads as specified by Tapestry. Tapestry in itself is decentralized on shared memory systems and each thread contains the information about which thread they should signal and where. Signaling occurs through a special counter that is atomically updated when dependencies are met. Once, all the dependencies are met, the thread will be run using the lower layer. If the thread has no dependencies, it will be immediately run. Depending on the lower layer implementation, it may be immediately run such as in Pthreads, but it could also be placed into a scheduling queue. If a thread is invoked with the start member before dependencies are met, the thread will not run.

4.3 Contributions

The two main contributions introduced in this chapter, are the framework designed for exploring synchronization and threading features that is independent of the lower level libraries provided and the data-centric approach to threads.

The portable framework that uses only C++ is essential for exploring new execution models and providing a transitory step for future system designers. The modular nature is essential for system level design.

Note, the importance of this idea for data-centric threads. It is essential for dependencies to be passed easily as part of the threading model to allow for the runtime

to determine how data needs to be moved to arrive to that thread. In shared memory this can be easily done. However, in future many-core systems without shared memory, the runtime may need to copy the data from local memory to other memory. This idea combined in unison with fast thread creation and keeping data local is essential for fine grain parallelism in future exascale machines. And, fine grain parallelism is essential to finding parallelism.

Chapter 5

TAPESTRY THREADS

Tapestry Threads is a framework for developing execution models and runtimes using only C++98. Threads is a C++ interface with glue that allows for the use of various synchronization and threading primitives. The interface also combines the features of Waft and Warp to allow for data-centric threads. It has many features, is modularly designed, and contains a number of contributions. In the design of Tapestry Threads, three separate components were created to build an integrated system:

1. Tapestry Waft: a threading model that provides glue to support C++ style threads including classes, inheritance, modularity, encapsulation, thread creation, joining, asynchronous function calls, and continuations.
2. Tapestry Warp: a synchronization model to provide mutexes, barriers, and thread level dependencies.
3. Tapestry Fibers: A heavily optimized runtime written for shared memory systems that uses work stealing/sharing to support fast thread creation.

5.1 Features

Tapestry features a number of important ideas to create a simple programming environment. These include full support for all of C++ features with threads, synchronization constructs, thread level dependencies, dynamic continuations, and simple general parallelism. Tapestry threads are much simpler and more flexible than standard C threads because they are created with C++ and include the synchronization features within the thread. Furthermore, Tapestry Threads support thread level dependencies with dynamic continuations which adds even more flexibility to the model. Lastly, function level parallelism can easily be achieved with general parallelism applied through the `async` keyword.

5.1.1 C++ Threads

Threading in Tapestry uses all the features of C++ for thread creation. This allows for thread creation using either function pointers or member function pointers.

5.1.1.1 Thread Creation, Running, and Joining

Thread creation is different from standard C libraries and is akin to the threading feature in the yet to be supported C++ 11 standard. Thread creation occurs via the `Thread` class. The API for Thread creation can be somewhat complicated to understand because it uses advance template features, but this provides many advantages compared to the standard C thread creation:

```
template < class R, class A1, ... class AN>
void Thread ( R ( * functPtr )( A1, ... AN ), A1 arg1, ... AN argN );
```

Figure 5.1: Tapestry Thread Creation API

The constructor first takes a function pointer (`functPtr`) with a return type of `R` that has argument types up to `AN`. Next, the constructor takes a list of arguments (`arg1` to `argN`) to pass to that function when the thread is running. Note the argument types passed are in the same order as function pointer signature and must match those of the function pointer signature otherwise a compile time error will result. An example of thread creation with and without arguments is shown in Figure 5.3. Calling `start()` will cause the current thread to be declared as ready to run. In addition, `start( int core)` will suggest for the thread to run on a certain core. Threads can return any type of value which is different from the standard C threads. The value can be obtained using the `joinValue` member function:

```
template < class R >
R Thread::joinValue ( void );
```

Figure 5.2: Tapestry Thread Join API

```
1  void th1( void )
2  {
3      cout << "Hi" << endl;
4      return;
5  }
6
7  int main ( void )
8  {
9      Thread thread( &th1 );
10
11     thread.start();
12     thread.join();
13
14     return 0;
15 }
```

(a) Creation

```
1
2  int th2( void )
3  {
4      return 5;
5  }
6
7
8  int main ( void )
9  {
10     Thread thread( &th2 );
11
12     thread.start();
13
14     return  thread.joinValue <int> ();
15 }
```

(b) Returns

```
1  int th3( int arg1, float arg2 )
2  {
3      int temp = static_cast <int> (arg2);
4
5      return arg1 +  temp;
6  }
7
8  int main ( void )
9  {
10     Thread thread( &th3, 1, 2.1);
11
12     thread.start();
13
14     return  thread.joinValue <int> ();
15 }
```

(c) Arguments

Figure 5.3: Tapestry Thread Creation Examples: In Figure *a*, a thread is easily declared using the class type `Thread` and passing a function pointer to it. Threads can be declared ready to execute with `start` member function, and users can wait for a thread to finish with the `join` member function. In Figure *b*, the value of returned by a thread can be returned using the `joinValue` member function which will wait for a thread to finish running to get the value. Finally, Figure *c* shows how values for thread arguments can be easily passed into a thread on declaration. `arg1` will be 1. and `arg2` will be 2.1.

The value returned is statically cast to the type `R`. If the thread hasn't finished running, the current thread will wait. An example of this is shown in 5.3b and 5.3c.

Finally, calling the member function `join( void )` on the thread variable will cause the current thread to wait for it to finish as seen in 5.3a. Joins will cause the current executing thread to yield the CPU if the thread they are waiting on hasn't finished.

5.1.1.2 Classes and Contexts

Unlike traditional C, thread creation can occur using member function pointers:

```
template < class R, Class C, class A1, ... class AN>
void Thread ( R ( C::*functPtr )( A1, ... AN ), C* context,
              A1 arg1, ... AN argN );
```

Figure 5.4: Tapestry Thread API For Methods

It takes similar arguments to thread creation using normal function pointers, except it takes a class `C` context reference which will be used to call the member function. The class context must have that member function and match the class of the member function pointer. Additionally, the context to use is an optional argument to the thread creation:

```
template < class R, Class C, class A1, ... class AN>
void Thread ( R ( C::*functPtr )( A1, ... AN ), A1 arg1, ... AN argN );
```

Figure 5.5: Tapestry Thread API For Methods Without A Context

If the context argument isn't passed to the thread, it will be created when the thread begins running on the local core or it will be placed in shared memory depending on the architecture.

Contexts, because they are classes, have all the same features as C++ classes. Information hiding, encapsulation, inheritance, polymorphism, et cetera are all supported. This allows for C++'s multi-paradigm approaches to be applied to threads. Figure 5.8 shows an example that applies information hiding and encapsulation to update shared memory values via threads and contexts. This important because it allows programmers to choose various paradigms or an inter-mixture of them to solve the problems. This flexibility lets them use the best tool for a job, since no one paradigm solves all problems in the easiest or most efficient way.

5.1.2 Dependencies

Tapestry supports thread level dependencies. Dependencies are expressed through the `dependsOn` member function:

```
void Thread::dependsOn ( Thread& threadMaster );
```

Figure 5.6: Tapestry Dependency API

This function indicates that the calling thread is dependent on `threadMaster`. This means `threadMaster` needs to finish before the caller can begin executing. Passing information to dependent threads can occur in four different ways: via sharing contexts, shared variables, during thread creation, or by using the dependency passing design that occurs by having a function pointer with more arguments than get filled in during thread creation:

```
template < class R, class A1, ... class AN1>
void Thread ( R ( *functPtr )( A1, ... AN1 ), A1 arg1, ... AN2 argN2 );
```

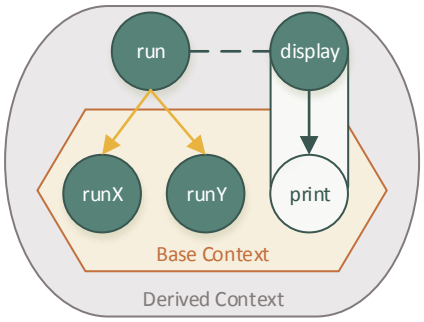
Figure 5.7: Tapestry Dependency Design: The prototype `R ( *functPtr )( A1, ... AN1 )` is matched to `arg1` to `argN2`. If an argument exists in the prototype, but isn't passed into the constructor it is assumed to be met by another thread.

```
1  class Base
2  {
3  private:
4    int x, y;
5
6  public:
7
8
9    Base ():
10   x(0),
11   y(0)
12     {
13
14   }
15
16   void runX( int x1 )
17   {
18     x += x1;
19   }
20
21     void runY( int y1 )
22   {
23     y += y1;
24   }
25
26   void print()
27   {
28         cout << "X: " << x
29              << " Y: " <<
30              y << endl;
31   }
32 };
```

(a) Base

```
1  class Derived:
2    public Base
3  {
4  private:
5    int z;
6
7  public:
8    void run ( )
9    {
10     z = 7;
11     Thread base1( &Base::run, this,
12                          5 );
13     Thread base2( &Base::run, this,
14                          4 );
15
16         base1.start();
17         base2.start();
18
19         base1.join();
20         base2.join();
21
22         display();
23   }
24
25   void display ( )
26   {
27     print();
28     cout << "Z: " << z << endl;
29
30     return 0;
31   }
32 };
```

(b) Derived



(c) Diagram

Figure 5.8: Tapestry Context Inheritance: This example shows how contexts can be used to create information hiding with threads. `Base1` and `base2` threads run in parallel and update the `x` and `y` values hidden from the derived class. The derived class uses the member function print to display their values.

If the number of arguments passed is less than the number of arguments in the function pointer, then the remaining arguments need to be filled by another thread. The signal information for this is stored in the class. Whatever value is returned by the dependency thread will be passed to the dependent thread. Using the `dependsOn` method will cause the dependencies to be filled in order. If the `dependsOn` is called more than the argument count, then the thread will wait for those additional threads, but not get any data from them. If the thread is started before the dependencies are filled, the thread will wait until the dependencies are added and run. When a thread is run and it has dependencies, the signaling is taken care of implicitly by the `Thread` class.

However, during thread creation there is a far more simple way to add dependencies if values are being passed to arguments:

```
template < class R, Class C, class A1, ... class AN >
void Thread ( R ( C::*functPtr )( A1, ... AN ),Thread& arg1, ... Thread
    & argN );
```

Figure 5.9: Tapestry Creation Via Dependency API

Threads the dependent thread need, are passed in during threads creation. Each argument in the function pointer `A1` to `AN` will be filled by the subsequent thread argument with the same number. This methodology can be mixed with the `dependsOn` method.

Example code of thread level dependencies is shown in Figure 5.10 that uses both methodologies described.

5.1.2.1 Dependency Loops and Pipelines

Tapestry supports dependency loops (Figure 5.12) and self loops (Figure 5.11) of any kind including while and do while (Figure 5.13) . A dependency loop occurs when a dependency say A depends on another dependency say B that in some form

```
1
2
3
4
5
6
7
8
9
10 int add ( int x, int y )
11 {
12     return x + y;
13 }
14
15 int main()
16 {
17     Thread t1( &add, 1, 2 );
18     Thread t2( &add, 1, t1 );
19     Thread t3( &add, t1, t2 );
20
21     t1.start();
22
23     return t3.joinValue <int> ();
24 }
```

(a) During Creation

```
1  int add ( int x, int y )
2  {
3      return x + y;
4  }
5
6  int main()
7  {
8      Thread t1( &add, 1, 2 );
9      Thread t2( &add, 1 );
10     Thread t3( &add );
11
12     //Fills y
13     t2.dependsOn( t1 );
14
15     //Fills x
16     t3.dependsOn( t1 );
17
18     //Fills y
19     t3.dependsOn( t2 );
20
21     t1.start();
22
23     return t3.joinValue <int> ();
24 }
```

(b) Using dependsOn

Figure 5.10: Tapestry Thread Dependencies: These two different codes use thread level dependencies and are equivalent returning the value 7.

depends on the results of A. This could occur further up the chain of dependencies from B. Loops allow the same thread to execute multiple times.

Using a loop, data can be continuously passed to a thread that is dependent on that loop effectively creating a pipeline (see Figure 5.14). Results will not be written to a thread if it is still executing. Thus, Tapestry acts like static dataflow in this regard and allows for pipeline parallelism.

To create a loop, the `dependsOn` member function is used to cause dependencies between threads to form a loop of dependencies like in Figure 5.11 where thread `a` depends on itself.

Loops need an initial value to start, but also have a dependency. Tapestry understands that if a thread has N arguments and 0 dependencies when created that those arguments may be just be initial inputs that can over written. Thus, if a user calls `dependsOn` after all the arguments are filled, Tapestry will cycle its slot to the first argument slot and use subsequent slots for each call to `dependsOn`. In this way users can create initial values on the input lines.

Because a loop thread is not seen as different from a regular thread, users need to indicate that the cyclic dependency is initially filled and met for the starting loop thread. This accomplished on start via the `startLoop` member function which will begin executing a thread regardless of any dependencies it may have. To stop executing the cycling loop, Tapestry provides the `Thread::exitLoop` to end a loop's execution from within the loop. Furthermore, any thread that calls `Thread::exitLoop` can be joined on with the `joinLoop` and the `joinValueLoop` which work like their non-loop counter parts, but will only stop blocking on a `Thread::exitLoop` call. Regular joining is also possible on the threads, but will stop blocking on the first time a thread is called. Allowing the programmer control over the exit condition means they can build any type of loop.

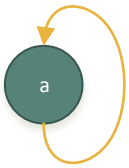
Finally, loops can have multiple parallel executing components. This can be used to build parallel pipelines as seen in Figure 5.15. These pipelines allow for classical pipeline parallelism.

```
1  int a ( int i )
2  {
3    if( i >= 0)
4      return i-1;
5    else
6    {
7      Thread::loopExit();
8      return i;
9    }
10 }
```

(a) Tapestry Self Loop Code

```
1  int main()
2  {
3    Thread t1( &a, 50 );
4
5    t1.dependsOn( t1 );
6
7    t1.startLoop();
8
9    return t1.joinValueLoop<int>();
10 }
```

(b) Tapestry Self Loop Code



(c) Tapestry Self Loop Diagram

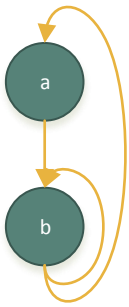
Figure 5.11: Tapestry Self Loop Example: In this example, `a` depends on itself. The user initially passes 50 to the `a` thread. The thread will continue signal itself to run until `loopExit` is called.

```
1  int a ( int i )
2  {
3    if( i >= 0)
4      return i-1;
5    else
6    {
7      Thread::loopExit();
8      return i;
9    }
10 }
11
12 int b( int i, int j )
13 {
14   return i-j/2;
15 }
```

(a) Tapestry Nested Loop Code

```
1
2
3  int main()
4  {
5    Thread t1( &a, 50 );
6    Thread t2( &b, 25 );
7
8    t1.dependsOn( t2 );
9    t2.dependsOn( t1 );
10   t2.dependsOn( t2 );
11
12   t1.startLoop();
13
14   return t1.joinValueLoop<int>();
15 }
```

(b) Tapestry Nested Loop Code



(c) Tapestry Nested Loop Diagram

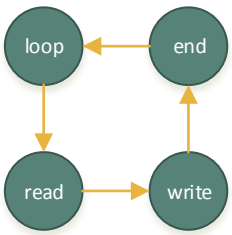
Figure 5.12: Tapestry Nested Loop Example: Tapestry fully supports any type of loops and loops within loops. It is not restricted to the classical `while` or `do while` loops. This example creates a nested loop that depends on results from an outer loop.

```
1
2
3  struct sensor;
4
5  sensor read( int i )
6  {
7    return readSensor(i);
8  }
9
10 bool write( char * y, sensor x  )
11 {
12   return writeSensor( y, x );
13 }
14
15 int end(int i, bool result)
16 {
17   if( i < 0)
18     Thread::loopExit();
19
20   return i;
21 }
```

(a) Tapestry Do While Code

```
1  int loop ( int i )
2  {
3    return i-1;
4  }
5
6  int main()
7  {
8    Thread t1( &loop, 50 );
9    Thread t2( &read );
10   Thread t3( &write, "test" );
11   Thread t4( &end );
12
13   t2.dependsOn( t1 );
14   t1.dependsOn( t4 );
15   t3.dependsOn( t2 );
16   t4.dependsOn( t1, t3)
17
18   t1.startLoop();
19
20   return t4.joinLoop ();
21 }
```

(b) Tapestry Do While Code



(c) Tapestry Do While Diagram

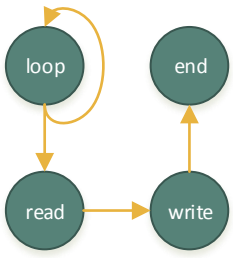
Figure 5.13: Tapestry Do While Example: Here Tapestry creates a sensor application do while loop that reads a sensor value and writes it to a file 50 times. Notice, one stage of the application can only be running at a time. This code is modified to allow for Pipeline Parallelism in Figure 5.14.

```
1  struct sensor;
2  {
3    int i;
4    //Other stuff
5  };
6
7  int loop ( int i )
8  {
9    if( i < 0)
10     Thread::loopExit();
11
12   return i-1;
13 }
14
15 sensor read( int i )
16 {
17   return readSensor(i);
18 }
19
20 bool write( char * y, sensor x  )
21 {
22   writeSensor( y, x );
23
24   if( x.i < 0)
25     return true;
26   else
27     return false;
28 }
```

(a) Tapestry Pipeline Code

```
1
2
3
4
5
6  void end( bool result )
7  {
8    if( result )
9      Thread::loopExit();
10 }
11
12 int main()
13 {
14   Thread t1( &loop, 50 );
15   Thread t2( &read );
16   Thread t3( &write, "test" );
17   Thread t4( &end );
18
19   t2.dependsOn( t1 );
20   t4.dependsOn( t1 );
21   t1.dependsOn( t1 );
22   t3.dependsOn( t2 );
23   t4.dependsOn( t3 );
24
25   t1.startLoop();
26
27   return t4.joinLoop ();
28 }
```

(b) Tapestry Pipeline Code



(c) Tapestry Pipeline Diagram

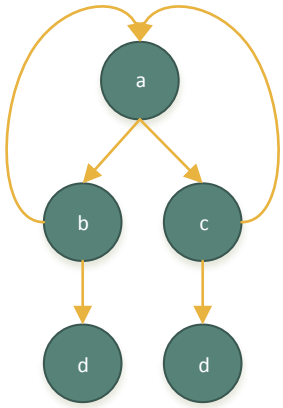
Figure 5.14: Tapestry Pipeline Example: In this example all 4 stages of the pipeline can be alive because `loop` will write its next increment into read as long as the last read is done. Thus, at time T: a T `loop`, a T-1 `read`, T-2 `write`, and T-3 `end` can be executing.

```
1  int a ( int i, int j )
2  {
3    if( i < 0 && j < 0)
4      Thread::loopExit();
5
6    return i-j;
7  }
8
9  int b( int i )
10 {
11   return i-1;
12 }
13
14 int c( int i )
15 {
16   return i-2;
17 }
18
19 void d( int i )
20 {
21   if( i < 0)
22     Thread::loopExit();
23   else
24     std::cout << " I: "
25               << std::endl;
26 }
```

(a) Tapestry Parallel-Loop Code

```
1
2
3
4
5  int main()
6  {
7    Thread t1( &a, 50, 1 );
8    Thread t2( &b );
9    Thread t3( &c );
10   Thread t4( &d );
11   Thread t5( &d );
12
13   t4.dependsOn( t2 );
14   t5.dependsOn( t3 );
15   t1.dependsOn( t2 );
16   t1.dependsOn( t3 );
17   t2.dependsOn( t1 );
18   t3.dependsOn( t1 );
19
20   t1.startLoop();
21
22   t4.joinLoop();
23   t5.joinLoop();
24
25   return 0;
26 }
```

(b) Tapestry Parallel-Loop Code



(c) Tapestry Parallel-Loop Diagram

Figure 5.15: Tapestry Parallel-Pipeline Example: In this example Tapestry creates a parallel pipeline with a parallel loop execution.

5.1.3 Synchronization

Tapestry provides a few basic synchronization features: mutexes, timed mutexes, and atomics. These just wrap available OS api calls and add glue where needed to provide better functionality.

Mutexes are the basic mutual exclusion and synchronization feature that comes standard with all threading libraries. They provide protection to shared data using locking mechanism provided by the mutex:

```
void Mutex( void );
void Mutex::lock();
void Mutex::unlock();
bool Mutex::tryLock();
```

Figure 5.16: Tapestry Mutex API

Prior to accessing data that needs to be synchronized between threads protected by a mutex, the `lock` or `tryLock` methods need to be called. Only one thread will hold the lock at a time. The `tryLock` method will return false if the lock is currently locked by another thread and will not block. Once a thread is done accessing the shared data, it needs to call the `unlock` method to allow other threads acquire the lock and subsequently access to the shared data. These can be wrappers to underlying OS libraries or user implemented. On x86-64, these use the lower level libraries provided by Linux and Windows.

Lastly, the atomic interface provides an interface to low-level atomic operations such as fetch-and-add.

5.1.4 General Parallelism

General parallelism can be expressed at the function level. Functions can be executed asynchronously using the `async` keyword:

```
template<class R, class A1, ... class AN>
static Future < R > Thread::async ( R (*funct)( A1, ... AN ), A1 arg1
    ... AN argN )

template<class R, class C, class A1, ... class AN>
static Future < R > Thread::async ( R (C::*funct)( A1, ... AN ), C*
    context, A1 arg1 ... AN argN )
```

Figure 5.17: Tapestry Async API

`Async` supports both function pointers and function member pointers and will begin execution of the function immediately. The future value returned is the same type as the return value of the function pointer. The value returned will voluntarily yield until the result is available if accessed. The important feature of `async` is that it allows free parallelism by giving the programmer a one line replacement to asynchronously execute function calls. This can be seen in Figure 5.19.

5.1.5 Parallel For

Tapestry supports embarrassingly parallel for statements:

```
template<class A1, ... class AN>
Thread::parallelFor ( int start, int end, int stride, int threadCount,
    void (*funct)( A1, ... AN ), A1 * arg1 ... AN argN );

Thread::parallelFor ( int start, int end, int stride, int threadCount,
    void (*funct)( int ));

Thread::parallelFor ( int start, int end, int stride, int threadCount,
    void (*funct)());
```

Figure 5.18: Tapestry Parallel For API

```
1  int square ( int x )
2  {
3      return x * x;
4  }
5
6  int main()
7  {
8
9      int x[50];
10     int checksum = 0;
11
12     for( int i=0; i<50; i++)
13         x[i] = square( i );
14
15     for( int i=0; i<50; i++)
16         checksum += x [i];
17
18     return checksum;
19 }
```

(a) Serial Code

```
1  int square ( int x )
2  {
3      return x * x;
4  }
5
6  int main()
7  {
8
9      Future <int> x[50];
10     int checksum = 0;
11
12     for( int i=0; i<50; i++)
13         x[i] = async( &square, i);
14
15     for( int i=0; i<50; i++)
16         checksum += x[i];
17
18     return checksum;
19 }
```

(b) Async Code

Figure 5.19: Tapestry Async Example: In Figure *a*, a simple checksum algorithm is performed in serial. In Figure *b*, the code is parallelized using `async`. First, the `x` array values are now declared as `Futures`. Then, the square values are computed in parallel with the `async` keyword. The values are added back to the checksum. Note, how little the code changes between the parallel and serial versions.

The first argument is assumed to be i-addressable. If no arguments are passed into the loop construct and the first argument to the loop function is an int, Tapestry will place consecutive iterations of i into the each parallel invocation of funct's first argument. The loop construct supports up to 9 arguments. `Stride` determines how much each loop value moves forward. A negative `stride` will move backwards. Setting the `threadCount` to `Thread::coreCount` will produce a completely static schedule with the work distributed as evenly as possible over the cores. Setting the `threadCount` to the absolute value of $\frac{end-start}{stride}$ will produce a completely dynamic schedule with 1 iteration per thread. Anywhere in between those values and the schedule will be a hybrid.

5.1.6 Continuations

Threading usually applies at the function level with one stack per thread and no sharing between threads, but the use of classes and the this pointer facilitates a methodology to allow threads to share key variables which allows for a simple and efficient way to break a thread into multiple executions. These classes are an easy way for programmer to create a closure. Furthermore, continuations are an effective way to make a serial function more fine-grain by splitting the function. Additionally, non-stack information and signaling information will be copied by the scheduler when using the continue statement.

The statements are:

```
template<class R, class C, class A1, ... class AN>
void Thread::cont ( Thread& arg1, ... Thread& argN )
```

Figure 5.20: Tapestry Continuation API

Using continue(cont) will make the current thread continue as the threads in the list. Continued threads can have new dependencies the current thread did not have. Continuing a currently executing thread multiple times is known as a *Dynamic Continuation*.

The `cont` statement will copy the current executing threads signaling information to a new thread that will be invoked with the function, context, and arguments given. Additionally, the continue statement will cause the current thread not to signal its dependents. This effectively causes the current thread to continue as a new thread with different information which includes a new context. If the context wishes to be preserved, use the this pointer. The continue statement will spawn any dependent threads of the continuing thread. An example of threads using continuations can be seen in Figure 5.21. Using the *Dynamic Continuation* can lead to interesting execution time optimizations.

Calling `cont` again after it has already been called causes another continuation to occur, but no signaling information will be copied because the signaling information was cleared on the first call to continue. Programmers can add more signaling information by using the `dependsOnThis` statement to add signaling information to the currently executing thread. I call this real time signaling. See Figure 5.22 for an example of an optimization that uses real time signaling with dynamic continuations to create a locality optimization. Additionally, because Tapestry's thread model allows threads to be a function, this means a programmer or runtime can decide if they want to execute a thread locally at execution time.

5.2 Support for Many Execution Models

The Tapestry model supports the programming models of EARTH and Codelets described in [15], fork/join using general threading, and static graphs. This section uses the basic serial Fibonacci code (Figure 5.23) and parallelizes it with various programming models and describes the advantages and disadvantages of each approach.

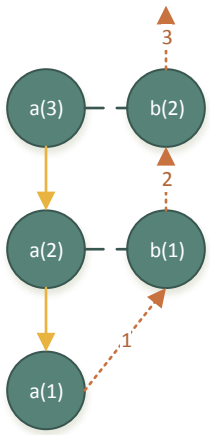
5.2.1 EARTH and Codelets

The EARTH threading model is data driven as described by Theobald[14] and has four unique attributes:

1. Multiple program counters.

```
1  int b ( int x )
2  {
3      return x + x;
4  }
5
6  int a ( int x )
7  {
8      if( x < 2 )
9          return x;
10     else
11     {
12         Thread t1( &a, x - 1 );
13         Thread t2( &b );
14         t2.dependsOn( t1 );
15         Thread::cont( t2 );
16     t2.start();
17         return 0;
18     }
19 }
20
21 int main()
22 {
23     Thread t1( &a, 3 );
24
25     t1.start();
26
27     return t1.joinValue <int> ();
28 }
```

(a) Continuation Code

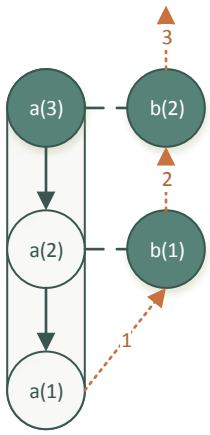


(b) Continuation Diagram

Figure 5.21: Tapestry Continuation Example: Thread `b(1)` and `b(2)` are the continuations of `a(1)` and `a(2)` respectively. Dashed lines represent continuations, dotted represent signals, and solid represent creations. `a(2)` and `a(1)` are spawned because they are dependencies of the `b` continuation. Notice `a(2)` and `a(1)`'s signals are copied to `b(2)` and `b(1)` respectively.

```
1  int b ( int x )
2  {
3      return x + x;
4  }
5
6  int a ( int x )
7  {
8      if( x < 2 )
9          return x;
10     else
11     {
12         Thread t1( &b );
13         Thread::cont( t1 );
14     t1.dependsOnThis( );
15     t1.start();
16         return a( x - 1 );
17     }
18 }
19
20 int main()
21 {
22     Thread t1( &a, 3 );
23
24     t1.start();
25
26     return t1.joinValue <int> ();
27 }
```

(a) Dynamic Continuation Code

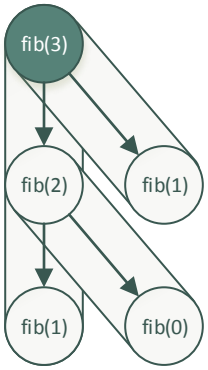


(b) Dynamic Continuation Diagram

Figure 5.22: Tapestry RT Continuation Example: Thread `b(1)` and `b(2)` are the continuation of `a(1)` and `a(2)` respectively. Dashed lines represent continuations, dotted represent signals, and solid represent creations. Notice unlike in Figure 5.21 `a(2)` and a(1) are recursively called from the same thread. Thus, `a(1)`, `a(2)`, and `a(3)` reside in the same thread. This means the a thread is spawning multiple continuations and linking them together. `a(2)` and `a(1)` are recursively called because they are dependencies of the `b` continuation. Notice `a(2)` and `a(1)`'s signals are copied to `b(2)` and `b(1)` respectively.

```
1  int fib ( int n )
2  {
3      if( n < 2 )
4          return n;
5      else
6      {
7          int x = fib ( n - 1 );
8          int y = fib ( n - 2 );
9
10         return  x + y;
11     }
12 }
```

(a) Serial Fib Code



(b) Serial Fib Diagram

Figure 5.23: Serial Fib Example: In this example we recursively compute the Fibonacci sequence. Assuming the value of `n` is passed into the `fib` function, the algorithm will recursively call `n-2` and `n-1` until `n` <2. At which point it will return. The diagram shows a call graph for `fib(3)` which should return a final value of 2.

2. Programs are divided into small sequences of instructions in a two-level hierarchy of threads: Threaded Procedures and Fibers.

3. Execution order among threads is determined by data and control dependencies specified in the program.

4. The local context for functions is allocated on the heap rather than the stack.

The second attribute just means that functions are broken up into multiple threads. These threads are like tasks in the fact they are distributed and executed on CPUs that are available. More threads will not be executing than cores available. The first thread is always invoked and the other thread needs to be invoked via a signal. The signal is stored in the local context. The primary difference between Tapestry and EARTH is that Tapestry allows signaling to occur at the thread level invocation. There is a one to one mapping between threads and functions.

The thread level dependency mapping Tapestry provides is much more powerful and has all the features of EARTH in conjunction with the use of classes for contexts if needed. In EARTH, context sharing is required to achieve passing of results of a fiber

to the next, but this is not the case in Tapestry. Tapestry doesn't need contexts for such a simple example.

Codelets extend upon the ideas of EARTH and add resources and signaling that can be introduced as dependencies. This information could be power requirements or other information such as system failure. Tapestry fully supports this by allowing any data type to be passed into the model.

The last idea to note, is that continuations are dynamic unlike in the EARTH model or Codelet model and allow for far more powerful expressions and optimizations not capable with these two models. Tapestry's continuations are more dynamic because they allow a user to create multiple continuations from the same thread. This allows for a thread to serially execute threads when beneficial instead of using the runtime.

According to Theobald the advantages of EARTH model comes from:

- The splitting of functions into multiple threads and assuming they are non-preemptive, which allows for long latency operations to be placed in a separate thread from other operations and thus allowing the operations to run concurrently without blocking the processor.
- Using thread level dependencies encourages movement of data in blocks, removes data from the critical path, and encourages locality.
- Using this model reduces context-switching.

However, the model has the disadvantage of creating more tasks(fibers) in the scheduling queue compared with the use of fork/join because the fork/join model would suspend the operation of execution and re-enable it for execution. In general, most of the claims by the EARTH model are unsubstantiated: long latency operations can be done by fork/join easily, fork/join encourages locality based on the stack and arguments passed to a thread can be easily blocked, and fork/join using a voluntary-preemptive scheduler has minimal context switching. Compare Fibonacci in the EARTH model (Figure 5.24) to the fork/join (Figure 5.26) and you will see the EARTH version produces roughly 2x the amount of threads. An async version of the EARTH model is also available for Tapestry which uses data driven futures to do a partial evaluation of

thread arguments during runtime. This excludes the EARTH non-preemption property (Figure 5.25) .

5.2.1.1 Comparison to EARTH

The difference between EARTH in Tapestry is seen in the design. For the EARTH model, all signaling infrastructure is setup by the programmer or an external compiler. It must be explicit in the Threaded-C language. In addition, for EARTH signals are stored in the frame of the Threaded Procedure function, must be manually met, and shared amongst all fibers. This is different from Tapestry which is designed for simplicity for the programmer and designed for a more distributed environment. For Tapestry, the signal is stored directly in the class that creates the thread. Implicitly, a thread stores all its dependencies it must signal and does so when it finishes. The thread doesn't share the signal space among other threads nor does it explicitly pass the signal address around. In addition, Tapestry provides basic shared memory through the usage of methods and classes. Classes naturally group threads in a shared memory environment.

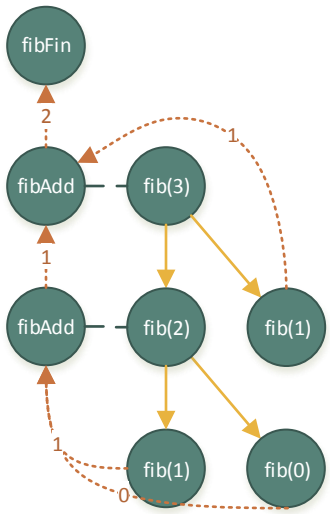
5.2.2 Fork/Join

The fork/join model of parallelism means you create a number of worker threads in parallel from a central location and then join them back to central location to get a result. Tapestry supports fork/join parallelism through the use of threads or `asyncs`. Fork/join in the threading model uses synchronization interface built into Tapestry. This interface uses the lower level thread joining mechanisms if available. Whereas, the `async` synchronization will use a synchronization mechanism separate from the thread. Fork/join does not provide data dependencies classically. Thus, thread level dependencies are not part of the model described here.

The advantage of Fork/join is seen in its simplicity and familiarity. Seen in the fact, that there is a regular convention for calling a function and getting its result returned to a central location. Furthermore, compared with the EARTH model it has

```
1  int fibAdd ( int x, int y )
2  {
3      return x + y;
4  }
5
6  int fib ( int n )
7  {
8      if( n < 2 )
9          return n;
10     else
11     {
12         Thread t1( &fib, n - 1);
13         Thread t2( &fib, n - 2);
14
15         Thread::cont( &fibAdd, t1,
16                       t2);
17
18         return 0;
19     }
20 }
21
22 int fibFin ( int fib, int sol )
23 {
24     std::cout << " Fib " << fib
25      << " is " << sol << std::endl;
26
27   return 0;
28 }
29
30 int main()
31 {
32   int FIB = 30;
33   Thread t1 (&fib, FIB);
34   Thread t2 (&fibFin, FIB, t1);
35
36   t1.start();
37
38   return 0;
39 }
```

(a) Tapestry EARTH Fib Code



(b) Tapestry EARTH Fib Diagram

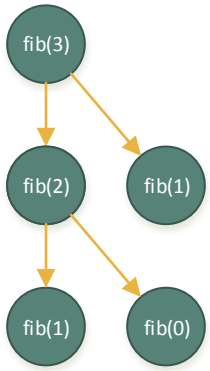
Figure 5.24: Tapestry EARTH Fib Example: For the EARTH Fibonacci (page 138 of [14]), each recursive call for the `fib` function is mapped to a thread: `t1` and `t2`. Furthermore, the addition of the values has been separated into its own function and thread using the `Thread::cont` to create a continuation of the current executing `fib`. The continuation is dependent on the recursively spawned threads, `t1` and `t2`. Remember when a continuation is called, the signaling information is copied from the current thread to the continuation, and the current threads signaling is canceled. The solid lines represent thread spawns and the dotted orange represent signaling. For Tapestry the signaling information is stored directly in the Thread class. In addition, signals are met implicitly by the runtime.

```
1  int fibAdd ( Future <int> x, Future <int> y )
2  {
3      return x + y;
4  }
5
6  int fib ( int n )
7  {
8      if( n < 2 )
9          return 1;
10     else
11     {
12         Future <int> x = async( &fib , n - 1);
13         Future <int> y = async( &fib , n - 2);
14
15         Thread::cont( &fibAdd , x, y);
16
17         return 0;
18     }
19 }
```

Figure 5.25: Tapestry EARTH Async Fib Example: Another way to do an EARTH style separation of components as opposed to Figure 5.24 is to wait on a future values passed into a thread from asynchronous functions. The difference here is that the `fibAdd` function will begin execution before all the future values are available. Whereas, with dependencies the thread waits for all the values. The `Thread:cont` function here is analogous to spawning an new thread since no signal information is needed.

```
1  int fib ( int n )
2  {
3      if( n < 2 )
4          return n;
5      else
6      {
7          Thread t1( &fib , n - 1);
8          Thread t2( &fib , n - 2);
9
10         t1.start();
11         t2.start();
12
13         result = t1.joinValue<int>()
14        + t2.joinValue<int>();
15
16         return result;
17     }
18 }
```

(a) Tapestry Fork/Join Fib Code



(b) Tapestry Fork/Join Fib Diagram

Figure 5.26: Tapestry Fork/Join Fib Example: In this example we map the recursive calls of `fib` to threads and wait on and the return values for those threads before adding the results. The solid yellow lines represent thread spawns.

a reduced amount of threads and because the model uses voluntary preemption, the context switching is minimized to only when necessary. An example for Fibonacci is provided in Figure 5.26 and using `asyncs` in Figure 5.27.

5.2.3 Static

The Tapestry model supports full chaining of dependencies allowing dependent threads to be added as dependencies to other threads. This means a static graph of threads can be built in memory and run using the model. The leaf threads at the bottom of the graph just need to be started and the runtime will discover all the dependencies and run them. A sample example is provided for Fibonacci in Figure 5.28. The example chains together a simple addition thread using a recursive function to compute Fibonacci. The programmer just needs to start the final thread returned and the code can run. The static version has large initialization time and memory requirements compared to the other models, but the run time should be significantly

```
1  int fib ( int n )
2  {
3      if( n < 2 )
4          return n;
5      else
6      {
7          Future <int> x = async( &fib , n - 1 );
8          Future <int> y = async( &fib , n - 2 );
9
10         result = x + y;
11
12         return result;
13     }
14 }
```

Figure 5.27: Tapestry Fork/ Join Async Fib Example: This example is similar to the that of Figure 5.26 except it uses asynchronous functions and future values to produce a fork/join style of parallelism.

faster because new threads are not being added during that time. Cache locality should also be increased for the threads since new work isn't be added. Lastly, context switching is non-existent.

The static graph could also be produced by higher level language such as the Tapestry Weave graphical language proposed and work in a similar manner as static dataflow. The static model doesn't seem too useful otherwise, but it could be combined with other approaches to create partial graphs so memory requirements do not become too large or the creation of a graph does not become too costly.

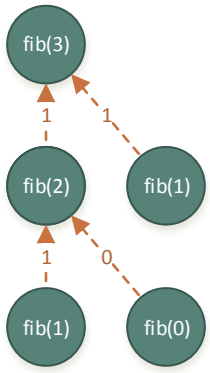
5.2.4 Hybrid

Because Tapestry allows for many different models of execution, these models and features can be combined to produce more efficient designs than one model could produce alone. The programmer can tackle the job in any way they desire. Figure 5.30 shows a hybrid example that combines both static and fork/join to reduce context switching by 1/2 and is more efficient by looking ahead at a depth of one.

In addition to mixing of models, Tapestry allows serial code to be easily mixed with parallel code by having functions as the base unit of code unlike EARTH and Cilk.

```
1  int fib ( int x, int y )
2  {
3      return x + y;
4  }
5
6  Thread link( int n )
7  {
8      if( fib > 1 )
9    {
10     Thread t1 = link ( n - 2 );
11     Thread t2 = link ( n - 1 );
12
13     Thread t3 ( &fib, t1, t2 );
14
15     return t3;
16   }
17   else
18     return Thread ( &fib, n, 0 );
19
20 }
```

(a) Tapestry Static Fib Code

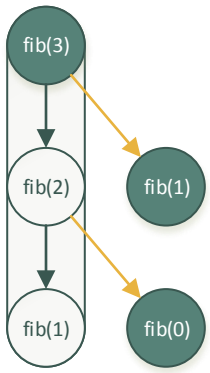


(b) Tapestry Static Fib Diagram

Figure 5.28: Tapestry Static Fib Example: This link function will return a graph of Fibonacci threads linked together in a dataflow-like graph manner. This means the values of the each `fib` call will be signaled to their parent thread. The parent thread will only start once it has all its arguments.

```
1  int fib ( int n )
2  {
3      if( n < 2 )
4          return n;
5      else
6      {
7          Thread t2( &fib , n - 2);
8          t2.start();
9
10         result = fib( n - 1 )
11        + t2.joinValue <int> ();
12
13         return result;
14     }
15 }
```

(a) Tapestry Hybrid Serial Fib Code



(b) Tapestry Hybrid Serial Fib Diagram

Figure 5.29: Tapestry Hybrid Serial Fib Example: Because Tapestry relies only on function or member function to represent a Thread, the user can mix serial and parallel calls to the same function. In this example, the left branch of `fib` recursive calls are computed serially to keep the thread continuing work while it is alive and the left calls are spawned as threads for others to work on in parallel.

This allows for code level optimizations that favor reusing a thread with locality instead of just killing its execution off and promoting parallelism. An example Fibonacci fork/join using this is provided for in Figure 5.29. This a key concept that needs to be explored in future exascale machines.

5.3 Hints and Metadata

Tapestry utilizes an extensive hint and meta system to allow for programmers to provide hints to the lower runtime to change the behavior of algorithms to favor various goals.

5.3.1 Hints

Programmers can optimize using the Tapestry framework to provide fine-grain or coarse-grain execution with any style of threading or synchronization. In particular, the model allows the programmer to choose how they wish to optimize their workload. They can use a fast dynamic fine-grain approach or a traditional static approach

```
1  int fib( int x, int y )
2  {
3      return x + y;
4  }
5
6  int fib ( int n )
7  {
8      if( n < 2 )
9          return n;
10     else
11     {
12         int result = 0;
13         Thread t2( &fib, n - 2);
14
15         if( n > 2 )
16         {
17
18           Thread t1a( &fib, n -1 -1 );
19           Thread t1b( &fib, n -1 -2 );
20           Thread t1( &fib, t1a, t1b);
21           t1.start();
22           if( n > 3 )
23           {
24             Thread t2a( n -2 -1 );
25             Thread t2b( n -2 -2 );
26             Thread t2( &fib, t2a, t2b);
27             t2.start();
28             result += t2.joinValue <int> ();
29           }
30           else
31           {
32             Thread t2( &fib, n - 2);
33             t2.start();
34             result += t2.joinValue <int> ();
35           }
36           result += t1.joinValue <int> ();
37         }
38         else
39         {
40            Thread t1( &fib, n - 1);
41            Thread t2( &fib, n - 2);
42            t1.start();
43            t2.start();
44            result += t1.joinValue <int> ();
45            result += t2.joinValue <int> ();
46         }
47
48         return result;
49     }
50 }
```

Figure 5.30: Tapestry Hybrid Fib Example: This builds a partial static graph for $n > 1$ fib calls with a join on the result of that graph. See Figure 5.31 for more information.

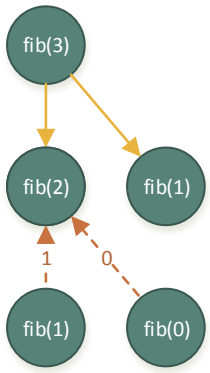


Figure 5.31: Tapestry Hybrid Fib Diagram: Notice that `fib(2)`'s leaf nodes are statically signaling it. This is because `fib(3)` created a subgraph of all 3 nodes and linked them together. Then, `fib(3)` joins on that subgraph. Thus, this example mixes static and fork/join models.

via sharing of work without changing the program. In addition, the model does not preclude optimizations for favoring locality vs parallelism. And, the model supports optimizations by mixing programming paradigms and extending them. The approach of how the underlying layer supports these optimizations varies. For instance, in shared memory systems locality enhances caching effects whereas in many-core chips it reduces system wide traffic. At the moment these features can be dynamically varied during runtime.

The runtime hint system allows switching on or off features during runtime. By default, the system will favor locality and dynamic load balancing (work stealing). However, a number of hints are available to change how the system works:

```
//Favors locality over parallelism
Runtime::Hint(Hint::LOCAL);
//Favors parallelism over locality
Runtime::Hint(Hint::PARALLEL);
//Favors dynamic load balancing over static scheduling
Runtime::Hint(Hint::BALANCE);
//Favors static scheduling over load balancing
Runtime::Hint(Hint::STATIC);
```

Figure 5.32: Tapestry Hint API

During the execution you can also set the total number of threads for the current problem or block size of threads during the current execution using the special hint interface:

```
//Current Problem Size
Runtime::Hint(Hint::SIZE, int size );
//Size of thread chunks
Runtime::Hint(Hint::BLOCK, int size );
```

Figure 5.33: Tapestry Hint Size API

For static scheduling, block size specifies how you wish to chunk the threads to other processors and allows a combination of dynamic and static scheduling. If no block size is given the scheduler will set the block size equal to the Hint::Size divided by the processor size. This will produce a completely static schedule. If neither are given, scheduler will set the size of blocks to two times the core count.

For dynamic load balancing (work stealing), changing the block size will change the number of threads stolen by other processors at a time, but varying the problem size will not affect the scheduler.

Further, for more fine-grain control to parallelism, the programmer can change the parallelism factor:

```
Runtime::Hint(Hint::PFACTOR, int amount);
```

Figure 5.34: Tapestry Parallelism Factor API

A higher parallelism factor means more parallelism with zero being the lowest and one hundred the highest. You can think of it as a percentage.

Furthermore, because these features can be dynamically varied, the programmer can easily build algorithms that varies these features during execution and build even more optimal solutions.

Depending on the lower level runtime these keywords may not do anything. For instance, if the lower layer uses Pthread or Windows threads these hints will not do anything since neither can utilize the information.

5.3.2 Metadata

The system also employs simple metadata wrappers to describe threads and data more extensively to the scheduler. The scheduler can use this information to determine how the system will handle the data passed into threads.

```
Thread::Metadata(Thread & threadToWrap, std::String data, std::String
    data1, ... std::String dataN);

Metadata::start( Metadata& dataToStart );
```

Figure 5.35: Tapestry Metadata API

5.4 Modular Components

Tapestry is modularized with an extensive interface layer built on top of Warp and Fibers that is flexible and allows hint information to be passed to the lower layer.

5.4.1 Scheduling

Scheduling is implemented within the lower layer runtime.

5.5 Implementation

Tapestry has been implemented in C++ using the current standard and is portable to all major operating systems including Windows, Linux, and SYS/BIOS. Tapestry's current runtime supports the dominant shared memory systems, but its modular nature allows it to be ported to other systems with optional ability to turn off features of the model not easily supported by other architectures. The interfaces provided by Tapestry just need to be implemented on other architectures to support these features.

5.5.1 Tapestry Fibers Shared Memory

Tapestry Fibers is a very bare shared memory implementation of basic threads that is designed for speed and efficiency, with minimal interfaces, simple and fast queues, and simple scheduling. Tapestry Fibers threads support a void argument pointer, joining, and voluntary preemption.

Tapestry Fibers provide interfaces to add a thread, join on a thread using a join value passed in the corresponds to a thread, or wait on a thread using your own variable to signal that waiting should end. These interfaces are simple. Both the join and wait interfaces simply take a pointer to a boolean value. The join interface will free the boolean as it is assumed it came from the scheduler and the wait will not. Adding the thread just takes in a reference to the thread which contains a function pointer and argument pointer.

In addition the implementation utilizes work stealing in conjunction with work sharing using a two level queue system allowing for many types of scheduling. More details will be further explained in Chapter 6.

Chapter 6

TAPESTRY FIBERS

Tapestry's lower level runtime is designed to be a portable and a configurable implementation of scheduling for threaded work. It doesn't use any external libraries except the standard threading packages available for Linux or Windows operating systems. Its design uses a number of features to allow for configuration between dynamic work stealing and static scheduling with block level granularity being allowed to change. The runtime is modular allowing for various schedulers or queues implemented by users. Furthermore, the design proposes the use of pushing a new scheduler on the stack to allow for work stealing via voluntary preemption without the need for a compiler.

6.1 Design

Tapestry Fibers employs a two level queue system if desired with interfaces to push threads into the queue as seen in Figure 6.1. The queue system implemented employs a fixed size local queue on each core that can only be accessed by the local core and a secondary queue that shares work with other cores. The local queue is lock free and does not use any lock-free algorithms since only the core it is on can access it. A minimal overhead lock-free queue is important for fine-grain execution. The other queue uses a locking mechanism or some algorithm to guarantee it can be stolen from by multiple other cores. This locking mechanism only occurs on pops, because writes are ordered on X86-64. The scheduler can be tuned to share some amount of initial work. Primarily it utilizes work stealing to load balance the system. The local queues are much faster to access than the shared queues and promote locality. However, the local queue can be turned off if parallelism is more important than locality. Such as in the case of many-core architectures where locality could create starvation.

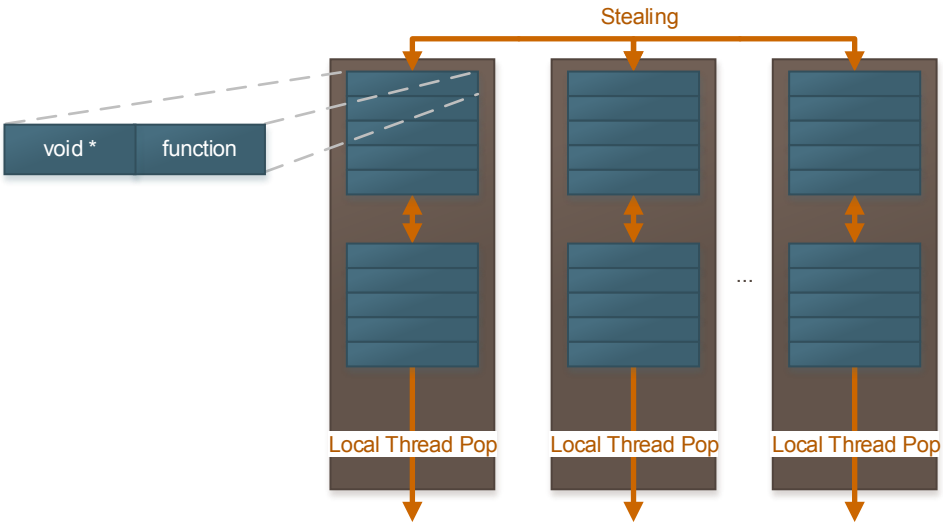


Figure 6.1: Tapestry Fibers Framework: Each core contains a sharing and local queue. Both queues together can be thought of as a one contiguous queue. The top is locked and the bottom is free of any synchronization. Local threads take work from the bottom and work stealing at the top.

Initially, the runtime will push work into the share queue until it becomes full, and then it will push work into its local queue. The size of the share queue was empirically decided to be two times the number of cores. As long as work is available in the local queue the local core scheduler will use work from there otherwise it will get work from the share queue, and if no work is available the local scheduler will steal work from another queue.

Queues are designed as simple stacks to promote locality and reduce overhead associated with deque maintenance. However, deques are available to be used to allow for better divide and conquer parallelism a la Cilk style. Work is pushed into the share queue until the share size is reached. Then, work is pushed into the local queue. If the share queue ever drops below the share size, the local core will push new threads into the share queue. In the deque version, both queues are a deque, but the other cores steal from the top. In addition, the local core will push its local top deque data onto the share queue when the share size falls below the minimum requirement. Local threads push and pull from the bottom, and other cores pull from the top. This allows other cores to steal larger workloads near the top of the tree. Stealing from the top enhances

efficiency for divide and conquer algorithms. Starvation is not a problem because tasks apply only to one application and are not scheduled with a time quantum.

In the shared memory system the optimizations flags and hints change how the implementation occurs or what happens during runtime. If locality is favored, the system will share data equal two times the local processor count before storing the data in its local queue. If the share queue ever falls below this amount, the scheduler will put new work from its stack into the local queue. On the other hand, if sharing is favored, the processor will only place work in its share queue and never use its local queue. The parallelism factor is used to override these settings, and set the share percentage of work equal to the PFACTOR. E.G., if 50 is used, every other thread added will go into the share queue. If 100 is used, all the threads will go into the share queue.

If dynamic scheduling is favored, the runtime will not divide work and share it to others, but let them steal. Stealing load balances the system; however, if a static schedule is favored, the system will share its work evenly to others by using a simple procedure of giving each processor a chunk of two times the core count of threads. The block size design follows that described in Chapter 5.

Finally each queue item is designed to be small as possible. So, the system employs an argument void* and a void (void *) function pointer to allow for minimal space and overhead using a partial direct queue without a free list.

6.2 Modularity

Tapestry Fibers is divided into interfaces for executing, stealing, waiting on threads, and hint systems. Each of these interfaces executes work on the queue using a standardized interface. This means queue types can be easily swapped out without recoding the implementations of the Fibers' interfaces. Furthermore, they do not have any external library dependencies except the standard operating system thread libraries such as Pthreads. Tapestry Fibers' interfaces use the underlying modular components and interfaces to interact with queues and wait for threads to finish.

Fibers has a few basic interfaces for passing hints, creating threads, and waiting on threads. Most of the work for Tapestry is done in the thread wrapper layer. These interfaces provide standard level thread creation seen in typical C style thread libraries.

To add a simple thread use the following api:

```
struct ThreadItem
{
  void* ( * function )( void * );
  void *  argument;
};

addThread ( struct ThreadItem & thread )
```

Figure 6.2: Tapestry Fibers Thread Creation API

The following will wait on bool value to be true before it returns. It will cause a new scheduler to be pushed on the stack until join is true.

```
void joinThread( volatile bool * join )
```

Figure 6.3: Tapestry Fibers Join API

Hints can be simple passed to the runtime using the hint system built in:

```
void addHint( hint::HintType hint )
```

Figure 6.4: Tapestry Fibers Hint API

6.2.1 Connecting Fibers and the Wrapper

Tapestry's uses a SystemThread level class to implement various interfaces for varying thread libraries. This SystemThread class is abstract, so each implementation can inherent it and implement their features. In theory Tapestry could switch between runtime libraries on the fly and use the best features of all the libraries. However it is

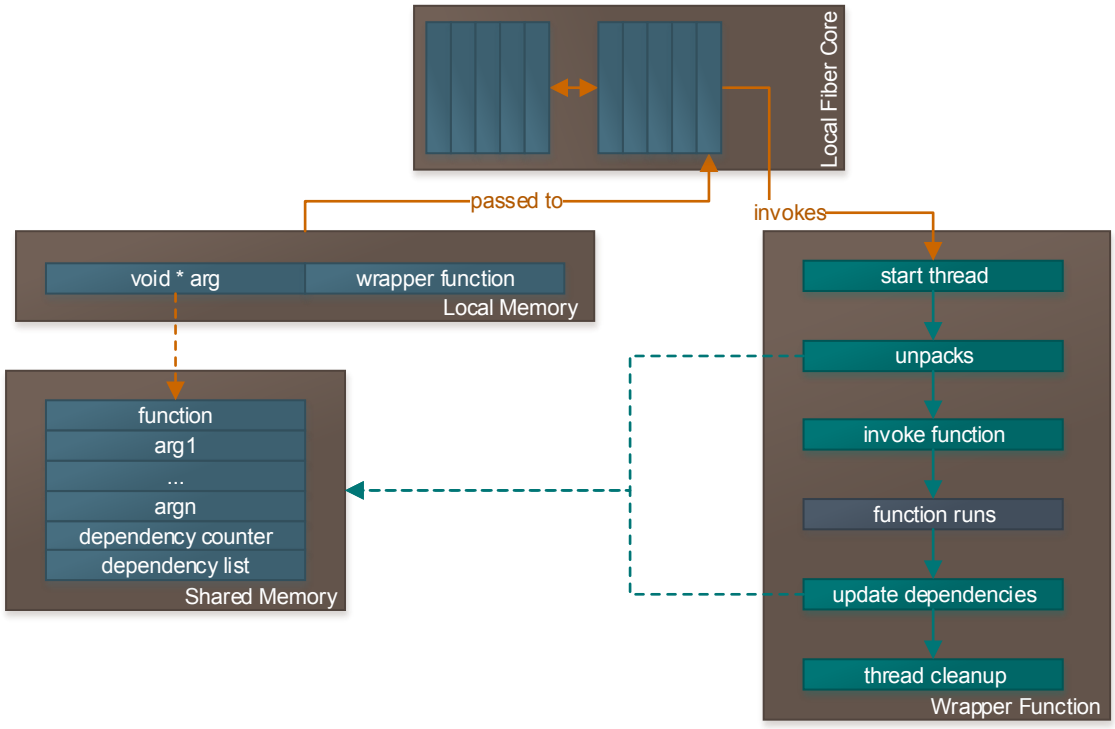


Figure 6.5: Tapestry Warp to Fibers: Using the interfaces provided by Fibers, Tapestry warp pushes the wrapper thread into Fiber's queue at the bottom. If the thread is not stolen by another worker, the wrapper will eventually execute and start the thread.

configured to only run one library at once currently. The top most layer in Tapestry just uses void * (void*) wrapper thread and passes that to the SystemThread class. It runs the system thread which causes the wrapper to be passed to Fibers as seen in Figure 6.5. When a Thread joins another thread, Tapestry will invoke join on the system thread which just invokes the lower level layer to wait on the current threads return value.

6.3 Fine-grain Optimizations

To provide a fast execution of fine-grain tasks, the spawn and synchronization features must have minimal overhead. For Tapestry Fibers spawning into local or share queue requires no locking mechanism because X86-64 does not reorder writes and the last write is used to indicate the item is available in the queue. Furthermore, pulling data from the local queue for execution does not require locking either. Removal of

locks is essential for performance when the task execution is so small that locking takes longer than executing the task. Another optimization is that Tapestry Fibers employs a partial direct task queue where there is only zero levels of indirection to the arguments and function pointers instead of the normal one. This allows for caching effects to occur on stealing because arguments and function to the thread are stored in the task. However, further arguments implemented in the upper level thread layer to implement the variable arguments requires a level of indirection. We hope to move this into the task queue to find further benefits.

Another important implementation is that Tapestry only runs one thread per core to maximize cache benefits. This guarantees tasks will not be switched out by the runtime during execution unless they are waiting which means they will stay in the cache will executing. Additionally, it is guaranteed that if a thread begins on a certain core it will finish executing on that core thus minimizing memory movement and maximizing cache effects. To accomplish this, the thread is kept in the stack and when stored it is pushed onto the last suspended task from that core. This means the execution benefits from temporal locality when restoring tasks.

6.4 High Throughput Queue

Tapestry allows you to swap in a high throughput input-restricted deque (algorithm seen in Figure 6.6) that has been designed for promoting locality on cache-based systems. The deque uses the well known principles that make spin locks fast on X86-64 in addition to provided new ideas about promoting locality. The algorithm stores a value to synchronize on at the location of every element in the deque. The queue will perform an atomic swap on this values when popping from either side. This leaves the front and back pointers not being locked on, but the local items in the queue. In principle reading these locks from memory will cause pulling the values stored nearby into the queue into memory. And in principle because x86-64 locks the cache-line we will only be locking on the value stored in the line which includes the local data we want to execute. Thus, this data is inactive and doesn't matter if it is locked on. This

is contradictory to known information that says to make the lock wide enough to fit in the cache-line to reduce false sharing and locking on unrelated data causing unintended blocks and serialization of code.

Because this queue is designed for dynamic work stealing systems you can only push data into one side only for the local core. The deque is lock and synchronization free for pushing due to write ordering on X86-64. In addition, reads on the local lock values use dirty reads before atomically swapping to increase performance. On failure the queue will back off and check other queues.

6.5 Work Stealing via Stack Pushing

Work stealing without a cactus stack and a compiler requires various implementations. In general, runtimes without compilers will use leap frogging or switch to a new thread when it suspends. However, Tapestry implements a new methodology without the drawbacks of leap frogging or spawning a new thread. Tapestry when it encounters a join point will cause the system to stop executing the current scheduler, by pushing a new scheduler on the execution stack as seen in Figure 6.5. The scheduler accesses the same queues as the original and will stop executing once the thread that it is waiting to join on is ready. If a scheduler waits on a join that waits on other joins this creates a problem where the scheduler stops working and does not do any useful work because it waits for the buried joins to complete. Thus, Tapestry does not suffer from the buried join problem because a new scheduler is placed on the stack that steals work and begins executing that work. This means Tapestry can steal from any executing core unlike leap frogging which is limited to stealing from it is stealer of a join. Furthermore, Tapestry won't incur any overhead by context switching between multiple threads: i.e. the waiting threads and the currently executing one. The drawback of Tapestry's stack pushing implementation is that it increases space requirements by having multiple schedulers stored on the stack in addition to the thread stack. However, this is limited for fork/join parallelism that the thread we want to join on will be

```
void push_front(ExecutionItem & value)
{
  while(*((volatile int *)&front->lock)!=0U);
  *front = value;
  front->lock=1U;
  ++front;
  if(front == top)
    front = bottom;
}

bool pop_front( ExecutionItem & item )
{
  temp = front-1;
  if(*((volatile int *)&temp->lock)==1U)
  {
    item = *temp;
    old = OS::Atomics::swap(&temp->lock, 0U);
    if(old == 1U)
    {
      if(front != bottom)
        --front;
      else
        front=top;
      return true;
    }
  }
  return false;
}

bool pop_back( ExecutionItem & item )
{
  while(1)
  {
    temp = back;
    if(*((volatile int *)&temp->lock)==1U)
    {
      item = *temp;
      old = OS::Atomics::swap(&temp->lock, 0U);
      if(old == 1U)
      {
        if(temp != top)
          ++back;
        else
          back=bottom;
        return true;
      }
    }
    else
      //Break and back off check other queues or do other work...
  }
  return false;
}
```

Figure 6.6: Tapestry High Throughput Queue

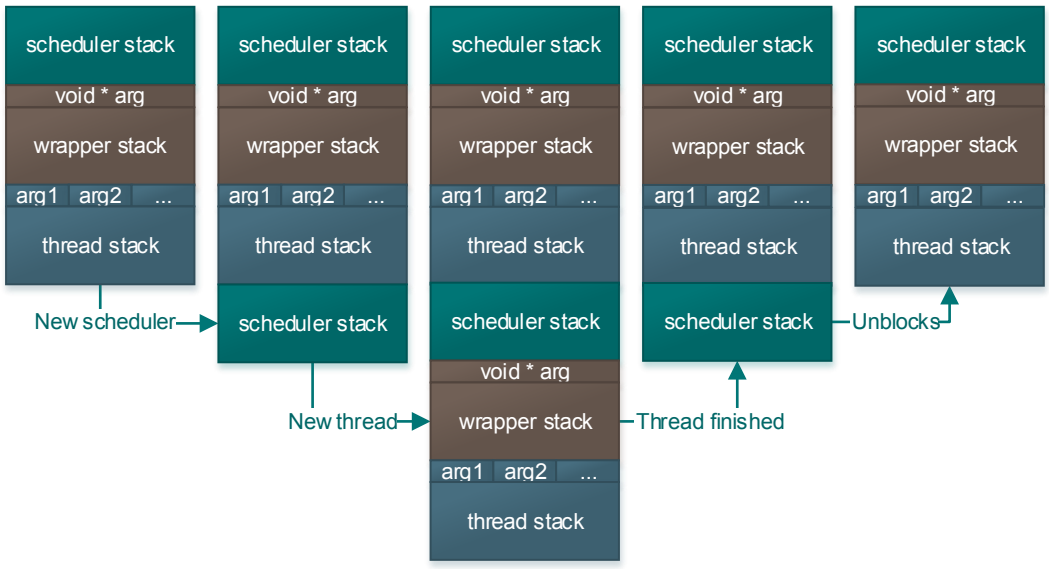


Figure 6.7: Tapestry Work Stealing via Stack Pushing: During a join operation Tapestry Fiber's will block using voluntary blocking. This picture shows how stack pushing works to resolve work stealing with blocking operations. On blocking, a new scheduler is pushed on the stack. The scheduler with access to the original thread queues will begin working on the work. Once a thread completes that the scheduler is waiting on, the scheduler will be pulled off the stack.

at the top of the stack. If it is stolen this means that the thread's queue is empty and needs to steal work.

6.6 TI C66x Port

The Texas Instruments' C66X DSP has a number of issues to consider when porting Tapestry to it. First and foremost is that there is no shared memory by default. Second, the DSP does not have cache consistency model and the programmer most invoke calls to a special API to maintain consistency. Third and finally, the DSP has no atomic operations and a limited set of locks available.

To handle shared memory allocation on the TI DSP, Tapestry configures the DSP on boot to use 4MB of L2 MSM SRAM as a shared region. Any structures that need to be shared will use the shared SRAM such as queues and Tapestry threads. In general we can swap in the 512 MB of off chip DDR3 to be used as shared memory if needed for bigger problem sizes. To further complicate matters, the TI DSP can use

various memory managers for allocating memory. The four managers provided by the SYS/BIOS operating system are:

- HeapMem: allocates various size blocks.
- HeapBuf: allocates fixed size blocks.
- HeapMultiBuf: internally uses fixed size blocks to allocate, but has variable size allocation.
- HeapTrack: for detecting memory allocation and deallocation errors.

For Tapestry Fibers HeapBuf would perform the best with little internal fragmentation because each item in the queue is a fixed size. However, because Tapestry's threads can be various sizes or a fixed size that is bigger than the queue this would cause a high level of internal fragmentation. HeapMem allows for various size blocks, but causes a high level of external fragmentation after blocks become free and as the link list is traversed the allocation time becomes non-deterministic. HeapMultiBuf uses multiple HeapBufs internally with various block sizes for each. It combines the speed of HeapBuf with the flexibility of HeapMem. This is good for Tapestry when using one thread size coupled with a different size for each queue item. So Tapestry employs HeapMultiBuf for fixed applications where only one type of thread is used. It uses two HeapBufs with sizes set to the local queue and the thread size.

Another problem, is that the TI chip only has cache coherency between L1D and L2 cache within the same core. There is no coherency between L1P and L2 within the same core or L1 and L2 across cores. Nor any coherency between L1, L2, and L2 Shared or external DDR3. Because of this the users most manually control cache line write-backs, invalidations, and write-back invalidations when using shared memory. Tapestry Fibers manages this by only write-back invalidating the cache line for other cores' shared queues on steals. It does a write-back on its own shared queue when pushing or popping from it. For the upper layer, a dependent thread's memory only needs to be invalidated when dependencies need to be met after a thread runs. The current running thread needs to be write-back invalidated to indicate it has run and returned a value.

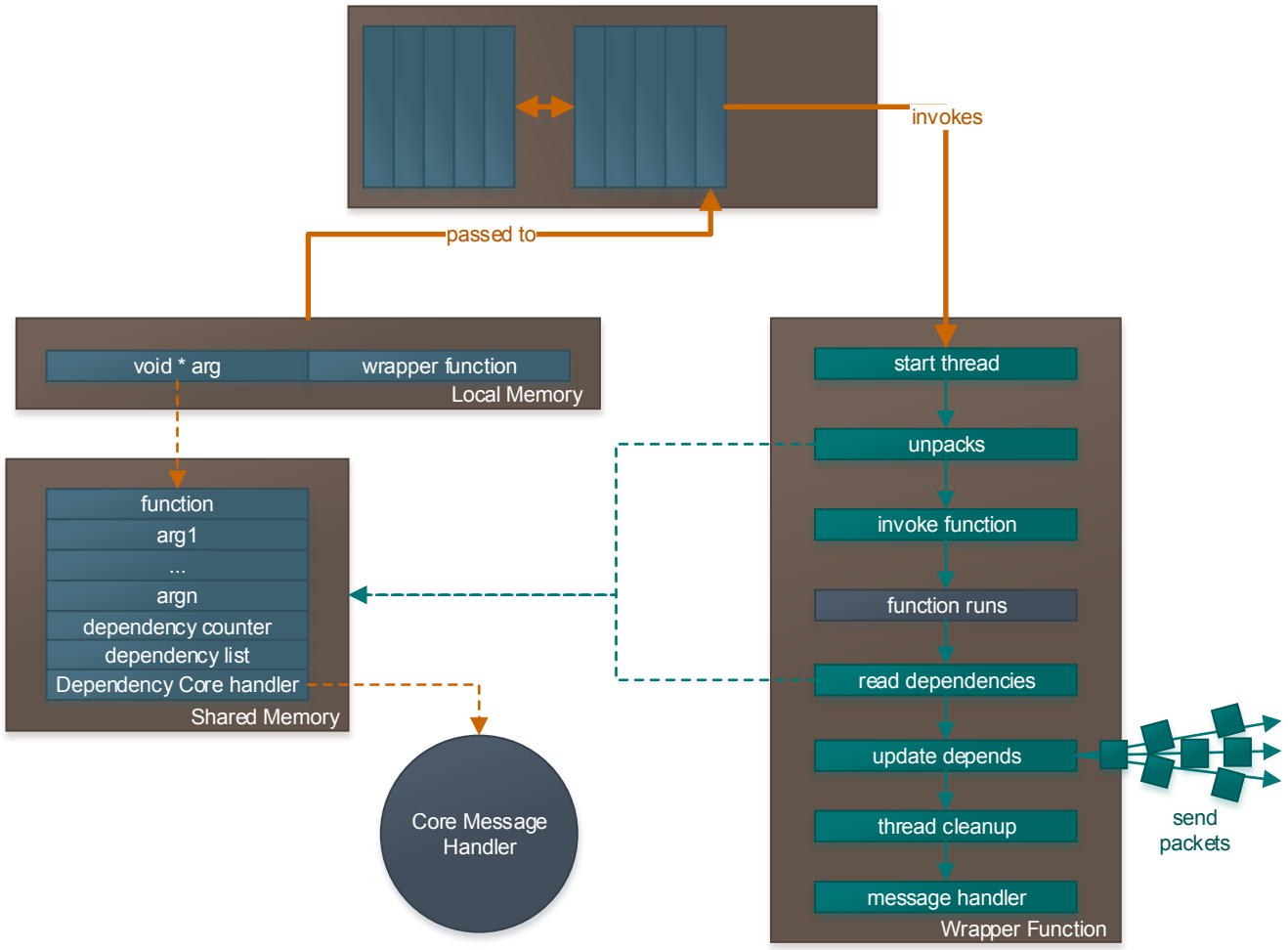


Figure 6.8: Tapestry TI C66X Port: Tapestry employs a messaging system for TI C66X. In the system, because of a limitation on locks, and the need to meet dependencies, Tapestry randomly assigns a core to handle dependency filling for each thread. In the figure, you can see how Tapestry will read which core will handle dependencies for said thread and send packets to each core that needs dependencies updated. Once all dependencies are met, the core will push the thread into its queue.

Finally, because dependencies need to be met by Tapestry and there are not any atomics and a limited number of locks, Tapestry employs a messaging system to send dependency updates to a managing core for a given thread. When a thread is created, the managing core is chosen at random. The dependency handling is handled within the wrapper layer whenever any thread runs. Tapestry will send messages to the handling core for any given thread that is dependent on currently finished thread. After which, it will check to see if there are any messages waiting in its message pump to handle. At which point, it meets the dependency information for that thread using the message pump. Figure 6.8 shows this. We chose this methodology primarily because the other solution we designed would virtualize the locks for each thread mapping it to a limited number of hardware threads. However, this would cause lots of contention on the locks for fine-grain parallelize tasks with a large number of dependencies being pushed through.

6.7 NUMA Considerations

When running Tapestry Fibers on a NUMA system, there are number of considerations that need to be met. First, what is the type of algorithm running on this machine? If we store our queues in a local node memory for each core, this will benefit algorithms with good load distribution. Fork/join parallelism works well with this type of configuration because most of the work is done by the local core and cores who steal choose cores closer before moving further away. However, for highly imbalanced work where only one thread is spawning the work, interleaving the queue memory across the nodes provides increased throughput because of more memory controllers. This in turn provides better access to the thread queues. Furthermore, certain data structures benefit from sharing them whereas others need only be local. Tapestry Fibers in this sense is NUMA aware. It can be setup to use local memory for the queues with interleaved memory for the rest of the data structures, just interleaved memory for the whole system, or just preferably local memory if possible.

Another consideration, his how to pin the cores to the machine. Tapestry was

designed for consecutive pinning, but this makes all the threads reside in only a few nodes when the system only uses a few cores. This limits the bandwidth available to these threads, but improves locality. Thus, Tapestry is NUMA aware in this sense also. It can be configured to pin threads consecutively, every other node in a round-robin fashion, or not all.

Chapter 7

EVALUATION

This section presents results for the performance of Tapestry on x86-64 on three different x86-64 architectures when it is configured to use the Fiber runtime. I perform micro-benchmarks and specific applications comparing against Intel OpenMP and Cilk Plus.

For my methodology, I average each point on each graph 10 times unless otherwise noted. I do not warm up programs before benchmarking because this introduces caching effects and produces false results. Furthermore, for each run I unload and reload programs to stop runtimes or compilers from caching subsequent results to the same call. For runtimes, I start them up before collecting data because I am interested in their performance on algorithms not startup costs. Finally, it is noted that I choose random data wherever applicable.

All benchmark code can be found in Appendix B. In addition, more case studies other than Bulldozer one presented here are provided in Appendix A. They provide more details and analysis on Tapestry dependencies, performance of Tapestry dependencies when used as glue for OS threads, performance of Tapestry in a hyper-threading environment, and redundant graph elimination performance benefits.

7.1 Benchmarks

Tapestry was evaluated with six different benchmarks. Most of the benchmarks are very fine-grain. It is compared against Cilk Plus and Intel OpenMP. The memory requirements are given for Help-First and Work-First thread spawning.

7.1.1 Fibonacci

Fibonacci numbers are integers given by the following series: $1, 1, 2, 3, 5, 8, 13, ...$ This can be modeled with the recurrence relation: $F_n = F_{n-1} + F_{n-2}$, and seed values occur at $F_1 = 1$ and $F_2 = 1$. Naturally, this recurrence relationship can be easily calculated with a recursive function call. The branch factor is 2. To parallelize the benchmark, recursive calls to F_{n-2} are asynchronously parallelized while calls to F_{n-1} are recursively called.

The maximum memory requirements for the serial program are given by n $\times$ stack frame where stack frame is the size of Fibonacci function stack frame. When parallelized, the requirements are the same if the asynchronous calls are immediately done by the current thread (Work-First) or left to be completed by other threads (Help-First). For these cases the requirements are (thread space $\times$ cores $\times$ n) + (stack frame $\times$ cores $\times$ n) because traveling to a leaf node will need stack space while spawning threads at each recursive call. This benchmark is very regular.

7.1.2 N-Queens

For the N-Queens benchmark, the goal is to place n queens on a n $\times$ n board without having two queens attack each other. The goal of this problem is to find every possible solution for a given board size. The benchmark utilizes recursion to place queens in position on the board. The benchmark has been heavily optimized with bit-fields to reduce the memory requirements and to know which positions have already been visited. Furthermore, it only calculates half the board and mirrors the rest. In the worst case the branching factor is n, but this is never reached because our benchmark marks off places where the queen cannot be placed. To parallelize the benchmark, the recursion for the first branch of each node is done recursively, but all other branches are done asynchronously.

The maximum memory requirements for the serial program are given as n $\times$ stack frame. If the program immediately executes a thread the memory requirements are (thread space $\times$ cores $\times$ n) + (stack frame $\times$ cores $\times$ n) otherwise it will

be (thread space $\times$ cores $\times$ $(n^2 - n)$) + (stack frame $\times$ cores $\times$ n).

7.1.3 N-Puzzle

N-Puzzle is a benchmark that is a classical search problem. For the problem, there exists a n $\times$ n board with $n - 1$ pieces on the board that represent a picture if placed in the correct order. These pieces can only be moved by sliding them into the empty slot on the board. The goal of the game is to form a picture by ordering the pieces correctly through sliding. This problem can be solved in a number of ways, but for the benchmark we find the optimal solution (least number slides) from a given state using iterative deepening depth-first search (IDDFS). IDDFS visits nodes in each level of a tree iteratively, but in a depth-first manner. First, it visits level 0 using depth first, then level 1, so forth up to level n. This benchmark does a tree search, but can be easily turned into a graph search with Tapestry's redundant graph elimination optimization. At most the branching factor is 4 and at least it is 2. To parallelize the benchmark, the recursion for the first branch of each puzzle node is done recursively, but all other branches are done asynchronously.

The maximum memory requirements for the serial program are given as d $\times$ stack frame. If the program immediately executes a thread the memory requirements are (thread space $\times$ cores $\times$ d) + (stack frame $\times$ cores $\times$ d) otherwise it will be (thread space $\times$ cores $\times$ d $\times$ 3) + (stack frame $\times$ cores $\times$ d) where d is the depth of the solution in the graph.

7.1.4 Quicksort

Quicksort is a divide and conquer sorting algorithm. Given a random array of integers of size n, the algorithm chooses a random pivot point in the array. It swaps the numbers greater than or equal to the pivot point to the right with those on the left effectively creating two lists. Then, it recursively sorts each list on the left and right with same idea until it reaches the leaf nodes. This algorithm can be easily parallelized by allowing one branch each node to be done in parallel.

This leads to an effective memory requirement of the serial program to be: (n $\times$ stack frame) + ($n \times$ integer size). The parallel version has a memory requirements are (thread space $\times$ cores $\times$ n) + (stack frame $\times$ cores $\times$ n) + ($n \times$ integer size)

7.1.5 Monte-Carlo

Monte-Carlo simulation is an embarrassingly parallel algorithm to produce a distribution of possible outcomes for use in risk analysis. It works by using a probability distribution for each factor that is uncertain. It then calculates results with different random variables from the probability function. Each iteration of these outcomes are typically calculated with a loop. Thus, each outcome value because it is independent of previous calculations can be calculated in parallel. Hence, the parallel version can easily break up each calculation to be done in parallel.

The memory requirements for the serial version is just stack frame size since it only expands one iteration at a time. The parallel version requires cores$\times$stack frame size+ thread space for Work-First, but requires for Help-First: cores $\times$ stack frame size + thread space $\times$ n where n is the number of steps in the algorithm.

7.1.6 Matrix Multiplication

Matrix multiplication is a standard mathematical computation that is done which takes two matrices and produces a resulting matrix. This computation is used in a number of various scientific applications. The standard parallelization technique for this is to break the matrix into smaller blocks and compute each block in parallel. This can be done by griding the matrix in memory or by breaking each matrix into small blocks of memory and storing consecutively block by block. The latter will increase inter-block locality; whereas, the former increase intra-block locality. In addition, the block size should be chosen to increase caching effects. Furthermore, multiple levels of blocks or grids can be used for each cache level to enhance performance. Finally, a register tile with vectorization enhances performance significantly.

The memory requirements for the program are $3\times n\times n$+cores×stack frame size+ thread size × block count for all the matrices and the execution of the threads.

7.2 Platforms

Tapestry was compared on three x86-64 platforms: Core 2, Core i7, and Bulldozer for various applications. In addition, it was ported to C6678 and verified for correctness.

Three different application suites for x86-64 machines are summarized in Figure 7.1 and Figure 7.2. The benchmarks show that its performance is generalizable to any x86 machine independent of application or machine and better or on par to Cilk Plus.

7.2.1 x86-64: Core 2

The particular model used in the benchmarks is the Core 2 Duo E6600. The architecture has 2 cores clocked at 2.4 GHz, contains a 32 KB L1 data and program cache per core, a 4 MB shared L2 cache, and a front-side bus that is clocked at 1066 MHz. The machine contains 6GB of memory and has a peak performance of 19.2 GFLOPS and Composite Theoretical Performance (CTP) of 37600 Millions of Theoretical Operations Per Second (MTOPS)[55].

7.2.2 x86-64: Core i7

The particular model used in the benchmarks is the Core i7-2600k. The architecture has 4 Hyper-threaded cores clocked at 3.4 (3.8 Max) GHz, contains a 32 KB L1 data and a 32 KB L1 instruction cache per core, a 256 KB L2 cache per core for data and instructions, and a 8 MB of shared L3 cache. The machine contains 8GB of memory and has a peak performance of 108.8 GFLOPS and CTP of 136000 MTOPS[56]. With turbo, the max peak is 122 GFLOPS and the CTP is 152000 MTOPS.

7.2.3 x86-64: Bulldozer

The processor is the 6234 with 12 cores clocked at 2.4 (3.0 Max) GHz, 16 KB L1 data cache per core, 64 KB L1 instruction caches shared by every two cores, 2 MB

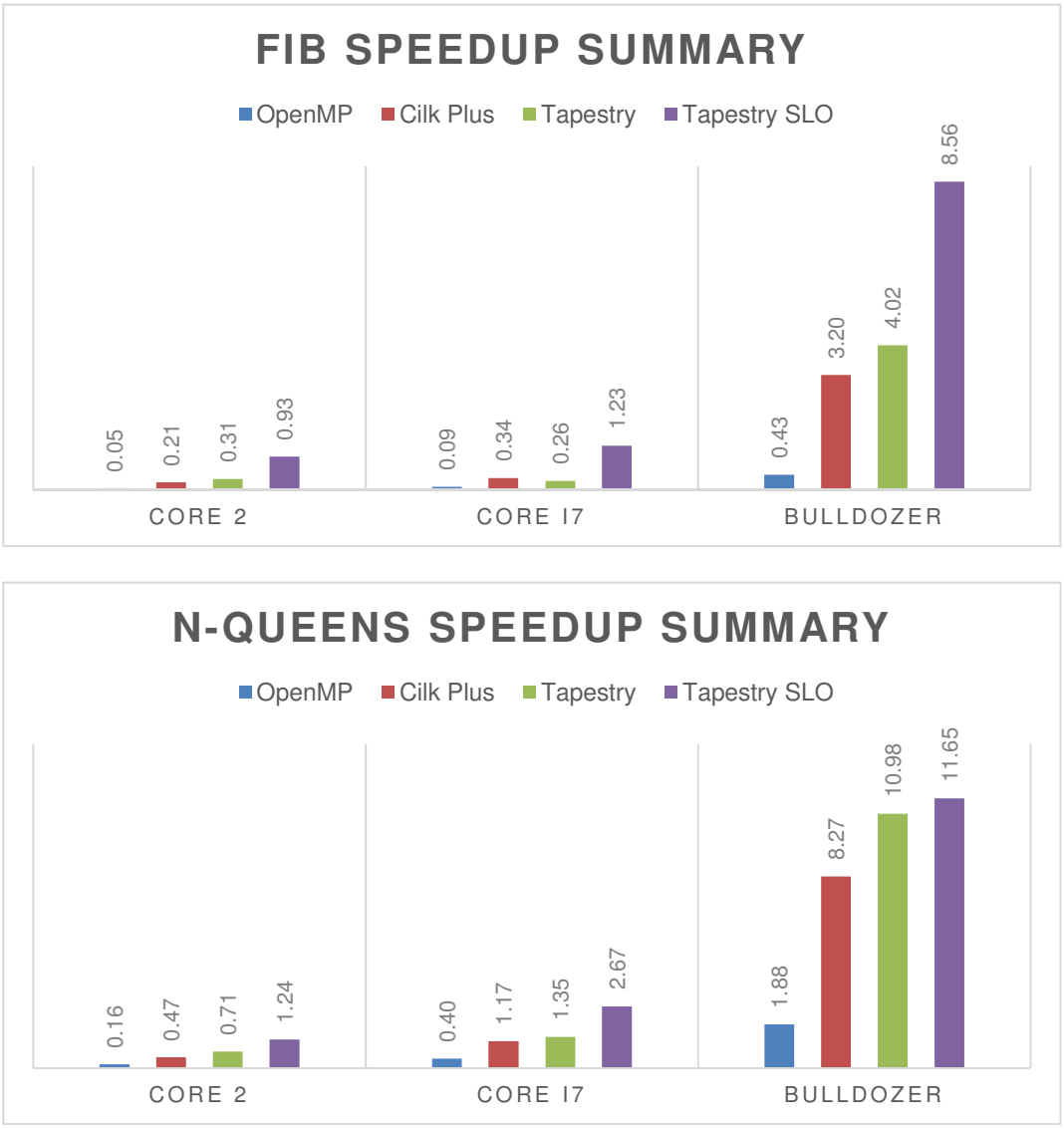


Figure 7.1: Performance Summary Part 1: The baseline is the sequential kernel. Two threads were used on Core 2. Eight threads were used on the Core i7. Forty-eight threads were used on Bulldozer.

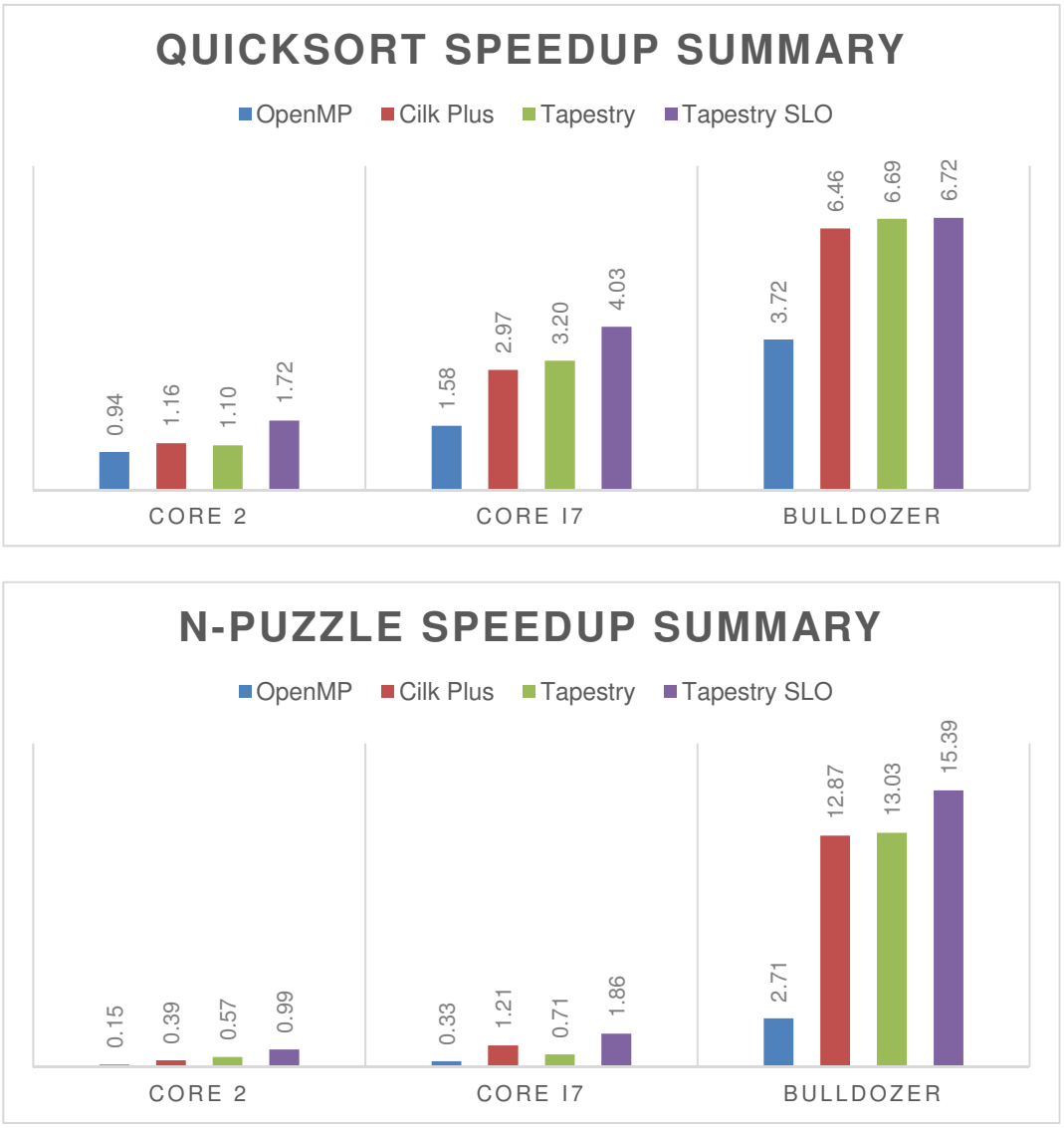


Figure 7.2: Performance Summary Part 2: The baseline is the sequential kernel. Two threads were used on Core 2. Eight threads were used on the Core i7. Forty-eight threads were used on Bulldozer.

of L2 caches shared by every two cores, and 8MB of L3 caches shared by every 6 cores. The board I use has 4 6234 for 48 cores and 8 NUMA nodes with 128 GB of memory. The CTP per processor is 217866 MTOPS and I calculated the peak performance to be 2.4X8X24 = 460.8 GFLOPS. With turbo, this is 576 GFLOPS. The architecture is described in more detail in Section 2.6.2.

7.2.4 TI C6678

The TI processor has 8 cores and was configured to be clocked at 1.25 GHz, have 32 KB L1 data and a 32 KB L1 instruction cache per core, 512 KB of L2 cache per core, 4 MB of shared memory on chip, and 512 MB of off-chip memory. It has a peak performance of 160 GFLOPS. The architecture is described in more detail in Section 2.6.1.

7.3 Case Study on Bulldozer

These tests compare the various components of the Tapestry High Throughput Input-Restricted deque in addition to that Cilk Plus' scalability on a 48 core AMD Opteron 64-bit 6234 machine at 2.4 GHz with 128 gigabytes of ram. The machine is unique in the fact it has cache-coherent Non-Uniform Memory Access (ccNUMA) with 4 sockets and 8 NUMA nodes. The system uses Scientific Linux 6. For all the test I use Parallel Studio XE 2011 with O3 optimizations on.

7.3.1 Runtime Micro-benchmarks

Tapestry when configured using a runtime could be less efficient than compiler-based runtimes because those runtimes can use static optimizations. Furthermore, Tapestry's decoupling of layers could also introduce overhead in the execution of tasks.

7.3.1.1 One Thread Overhead

In the first test, I compute the overhead of spawning and finishing a thread. I compare this to Cilk Plus and OpenMP. The test spawns an empty thread individually

	Spawn	Join	Total
Tapestry	31.73 ns	10.86 ns	42.59 ns
Intel Cilk Plus	41.46 ns	02.31 ns	43.77 ns
Intel OpenMP	29.39 ns	14.70 ns	44.10 ns

Table 7.1: Overhead of One Thread: In this test I spawn empty threads on one core using task parallelism and calculate the overhead spawning and joining on 1 thread. Cilk Plus most likely executes the spawned thread immediately and executes the continuation later. Whereas Tapestry spawns the thread and executes during the join. These numbers were averaged over 100 million iterations.

and joins on it. I compute this 100 million times and average the results. It is worth noting I use OpenMP's task framework and not data parallel for loops for this test.

Tapestry's overhead is on par to both OpenMP and Cilk Plus as seen in Table 7.1. From this information, I can extrapolate Tapestry has about three times as much overhead to create a task as executing it.

7.3.1.2 Parallel Scheduling Overhead

In this test I spawn a number of empty threads in a loop and join on all of those. I look at the serial execution vs using all the cores on the machine. This test shows the overhead of load balancing work from one core across all cores on Tapestry, Cilk Plus, and OpenMP. The results are shown in Figure 7.3. Tapestry and OpenMP provide parallel for constructs to provide better load balancing for this type of work.

The results show that Cilk Plus performs significantly better. Perhaps, Cilk Plus is using a back-off on stealing which results in less movement of the work which improves performance for fine-grain workloads that are less balanced.

7.3.1.3 Dependency Overhead

The final two benchmarks, test the performance creating 100 million parallel tasks with dependency logic and the time it takes to execute 100 million threads that form a chain of dependencies in serial. The results show that I can spawn one dependency task in 163.75 ns and execute a dependent thread every 523.09 ns. This means I can spawn dependency tasks 3 times faster than I can execute them.

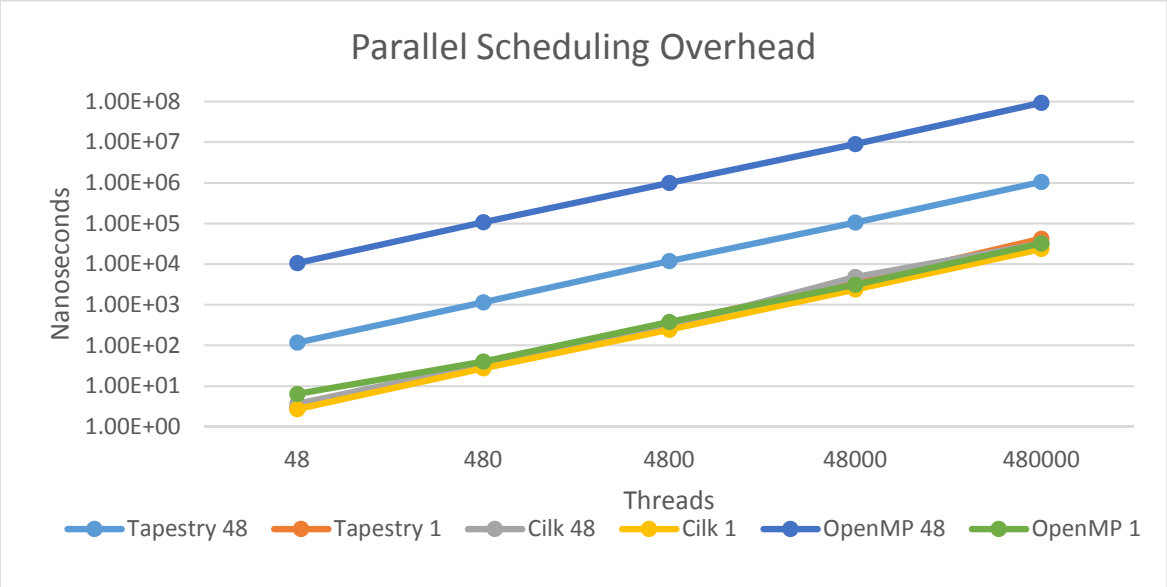


Figure 7.3: Parallel Spawn: This test differs from Table 7.1 by spawning the number of threads on the x-axis all before joining on them. Whereas, the other test only spawns 1 thread at a time and joins it. The much higher overhead on Tapestry for 48 cores vs Cilk Plus is most likely due to Cilk Plus doing a back-off on steals.

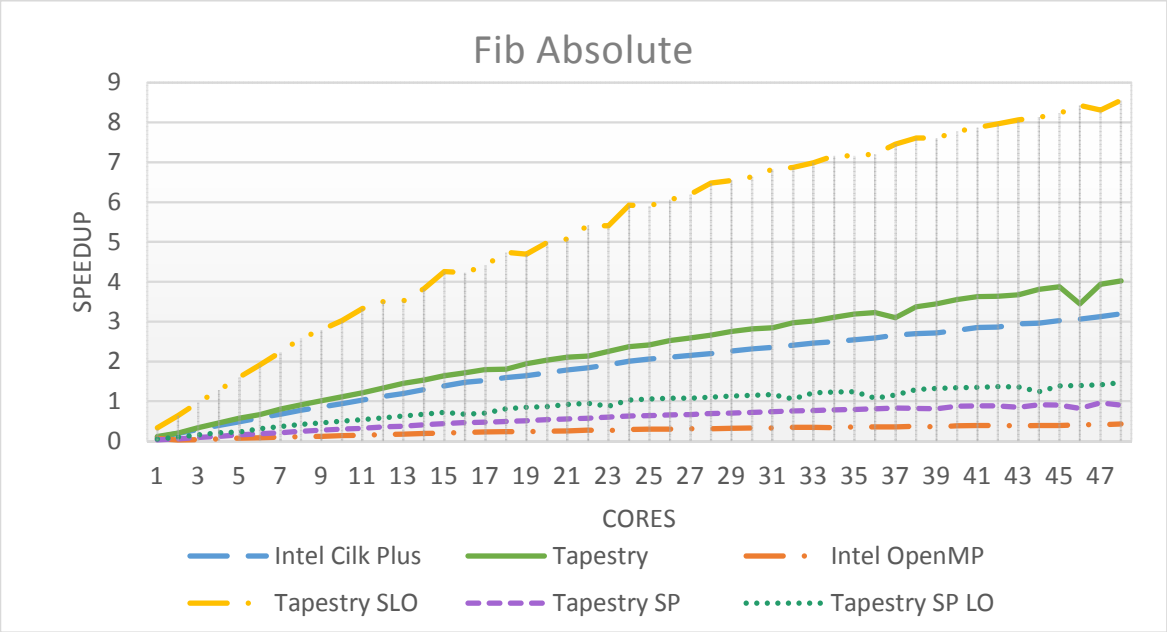


Figure 7.4: Fibonacci Scalability: A Fibonacci number of 44 was calculated.

7.3.2 Runtime Benchmarks

In these benchmarks I test the performance of the runtime using various applications.

7.3.2.1 Fibonacci Scalability

I compute Fibonacci of 44 on this test. The results seen in Figure 7.4 show that Tapestry's performance exceeds that of Cilk Plus. It appears Tapestry's speedup is function of a line whereas Cilk Plus' speedup is only a function of a line up to 16 cores and starts to fall off. The fall off appears to get worse as cores are added and be asymptotic. OpenMP is also linear up to around 16 cores and falls off like Cilk Plus. In addition, the linear property of Tapestry's deque does not appear to fall off up to 48 cores. Fibonacci is very fine-grain which bods well for fine-grain applications.

I note that Tapestry's results appear to have two outliers toward the upper core count. Finally, Tapestry's SLO implementation achieves pretty good performance in this test, but the performance is not linear and appears to fall off as core count

increases. This is expected because SLO limits parallelism in favor of locality and should create more starvation as core count increases if the problem size is not big enough. In addition, I note the performance of using split-phase transactions on the code produces results that are much slower than the fork/join model. This is most likely caused by the combination of the overhead of more memory to represent dependency logic and overhead caused by the smaller size of tasks when split into finer-grain parts.

7.3.2.2 N-Queens Scalability

I use a board size of 16 for these tests. Like the Fibonacci test, the results in Figure 7.5 show that Tapestry's performance exceeds that of Cilk Plus. Like before, Tapestry's speedup is linear whereas Cilk Plus' speedup is only linear up to 16 cores and starts to fall off. The fall off appears to get worse as cores are added and be asymptotic. OpenMP is also linear up to around 16 cores and falls off like Cilk Plus. In addition, the linear property of Tapestry's deque does not appear to fall off up to 48 cores for this coarser task. Here SLO scales well again, but falls off. In addition, the performance difference between the two is much smaller. N-Queens is pretty fine-grain and less balanced than Fibonacci. However, comparatively N-Queens is more coarse than Fibonacci. The split-phase transaction versions appear to be about half the speed of Cilk Plus. Once again, the difference in performance is most likely due to additional memory overhead and overhead caused by having finer tasks.

7.3.2.3 Quicksort Scalability

I sort on a size of 55 million for these tests. Figure 7.6 shows that the scalability of quicksort is pretty limited. This is due to the fact that quicksort has to reach a depth of 7 before 48 cores can work on it. Another limiting factor is that the amount of work grows finer and finer the deeper the graph is traversed. Which means that the amount of work for the first thread at the top is equal to the amount of work divided among its children at a depth of seven. This creates a sequential bottleneck and multiple bottlenecks till a depth of 7 is reached. Here the results indicate that Cilk

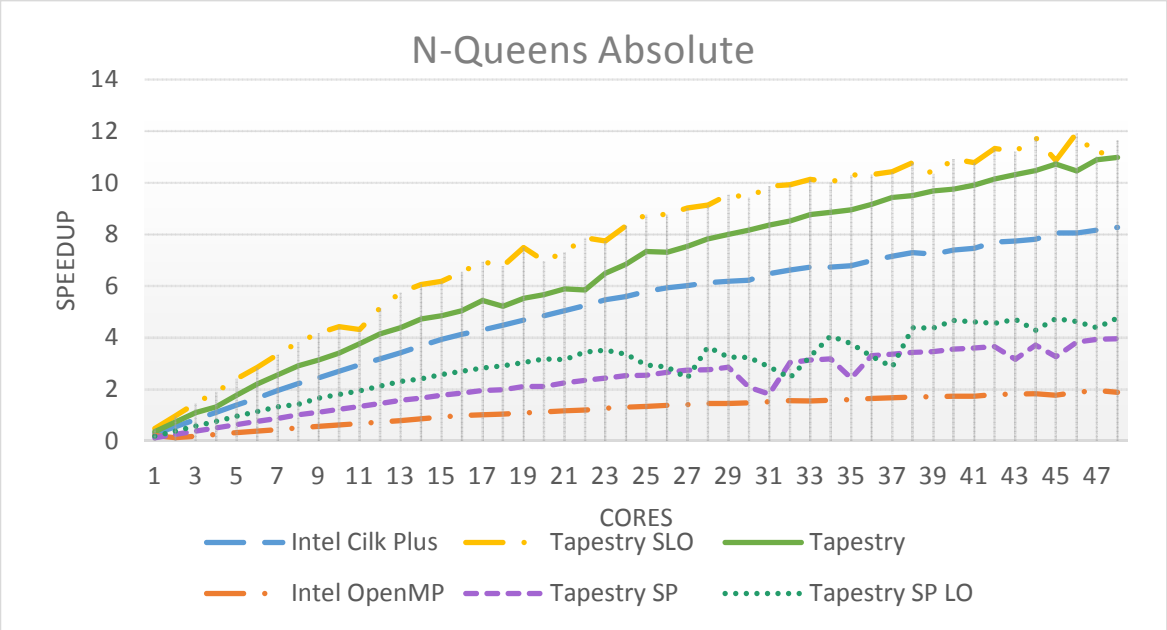


Figure 7.5: N-Queens Scalability: A board size of 16 was used.

Plus' performance is worse than Tapestry. SLO performs well, but falls off as more cores are added. Our implementation for dependencies requires a barrier thread to indicate all threads are finished which require 55 million threads register to the barrier. This has significant overhead due to contention on the dependency met variable. So our results took too long too finish. Results indicated it appeared to be around 20X slower than the fork/join version. Performance most likely improve if I utilized a split-phase transactions to create a tree-like barrier that is very similar to the Fibonacci and N-Queens split-phase addition operation, but does no useful work.

7.3.2.4 Monte Carlo Scalability

Here our number of paths is 192. Like the other two tests, for this embarrassingly parallel algorithm with very coarse thread work, Tapestry begins off slow, but it is linear whereas Cilk Plus starts to fall off at 16 cores. Tapestry's performance exceeds that of Cilk at 38 cores. Tapestry again is slower initially due to the fact its threads are pinned. See Figure 7.7.

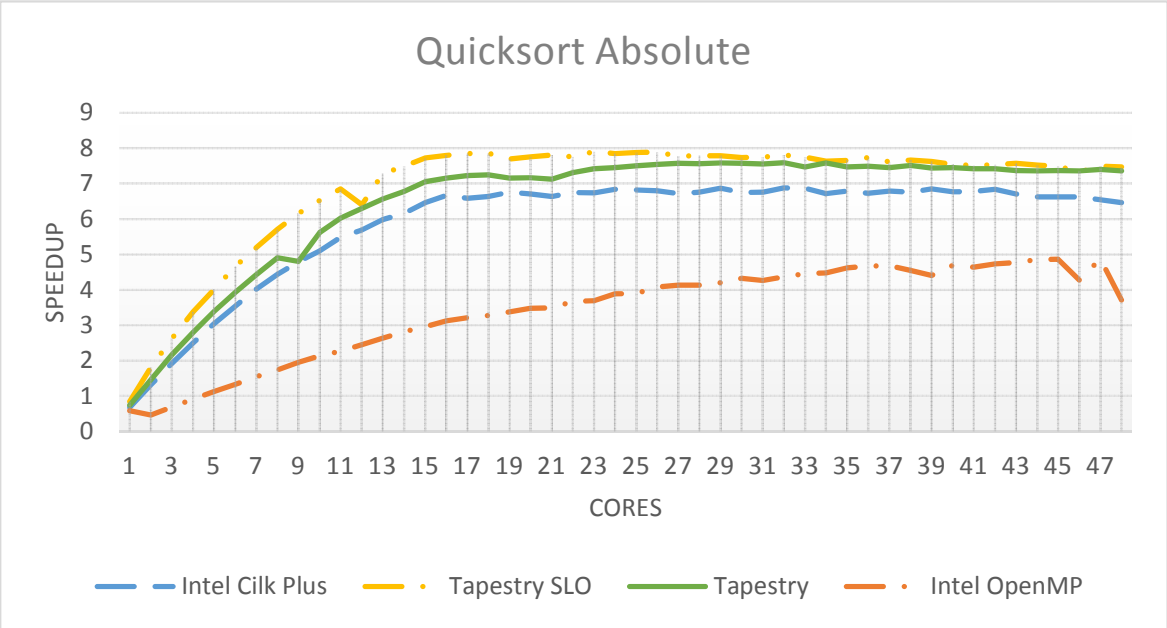


Figure 7.6: Quicksort Scalability: 55 million integer numbers were sorted.

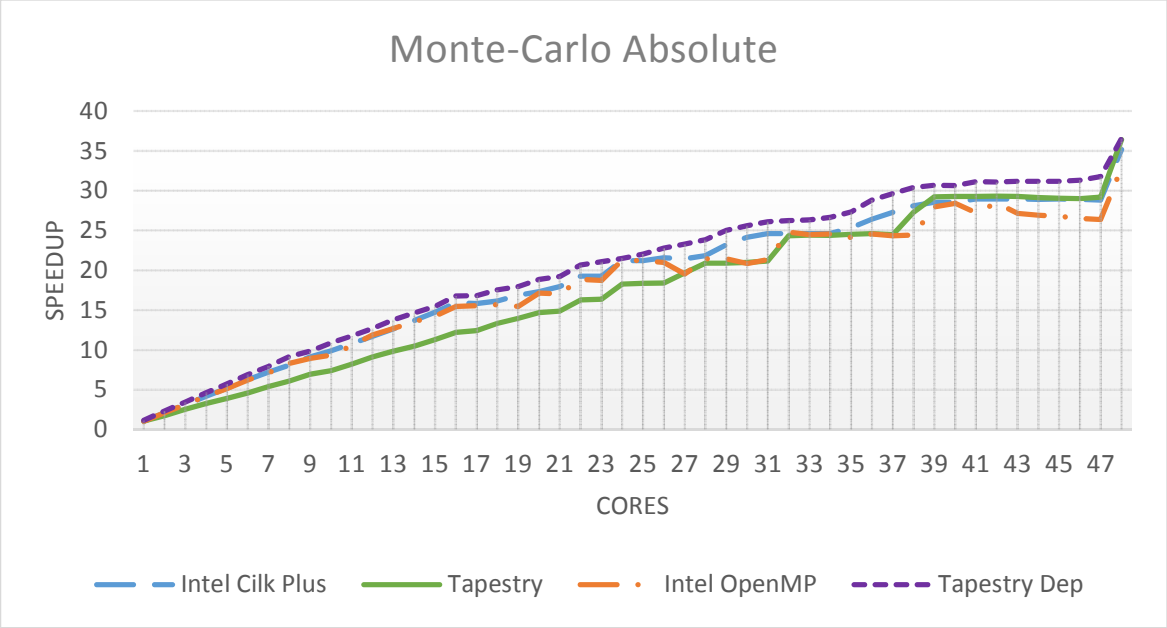


Figure 7.7: Monte Carlo Scalability: 192 paths were calculated. The Tapestry dependency version uses a barrier thread to end the program. Whereas, regular Tapestry joins on each thread.

7.3.2.5 N-Puzzle Iterative Deepening Scalability

The solution to the puzzle is at a depth of 20 for a 3x3 board. The N-Puzzle test (Figure 7.8) uses Iterative Deepening Depth-First search to expand a graph and search for the solution to a N-Puzzle. Iterative Deepening effectively visits the nodes in breadth-first order, but with space bounds equal to that of depth-first search. In this search I am building a graph and searching it together in one step. The test is pretty fine-grain since the nodes for the N-Puzzle do not do much work. In the best case, each node has two synchronizations. Similarly, in the worst case there are four synchronizations.

The test shows that Tapestry and Cilk Plus' performance is much more scalable that of Intel OpenMP. The jagging in the graph is caused by the fact the test will cancel the search once a solution is found and more threads does not guarantee at a depth of 20 that a solution will be found quicker. In terms of performance SLO does well. For this test we did not produce a threaded dependency version. It would require a split-phase transaction or barrier thread.

7.3.2.6 Matrix Multiplication Kernel Static Scalability

The problem size for this test is 16640x16640 double precision matrices. This allows the machine to gain caching effects due to alignment and is big enough to guarantee enough work across 48 cores. For my matrix multiplication tests on the AMD machine, I choose a custom kernel after evaluating the scalability and performance of it compared with the Intel Math Kernel Library (MKL) BLAS dgemm and the AMD Math Core Library (ACML) dgemm using FMA4 instructions. My kernel on one core gets around 10 GFLOP/s, the MKL library produces only around 5 GFLOP/s, and AMCL does about 16 GFLOP/s. Both are linked to serial libraries. In addition, my kernel with Pthreads gets better scalability than MKL using a static schedule up to 24 cores, but worse than AMCL. Whereas, the MKL library flat lines when linked with OpenMP at 24 cores and the results for AMCL become inconsistent. I note here that the scalability of my kernel is better initially if I don't pin the threads to cores instead

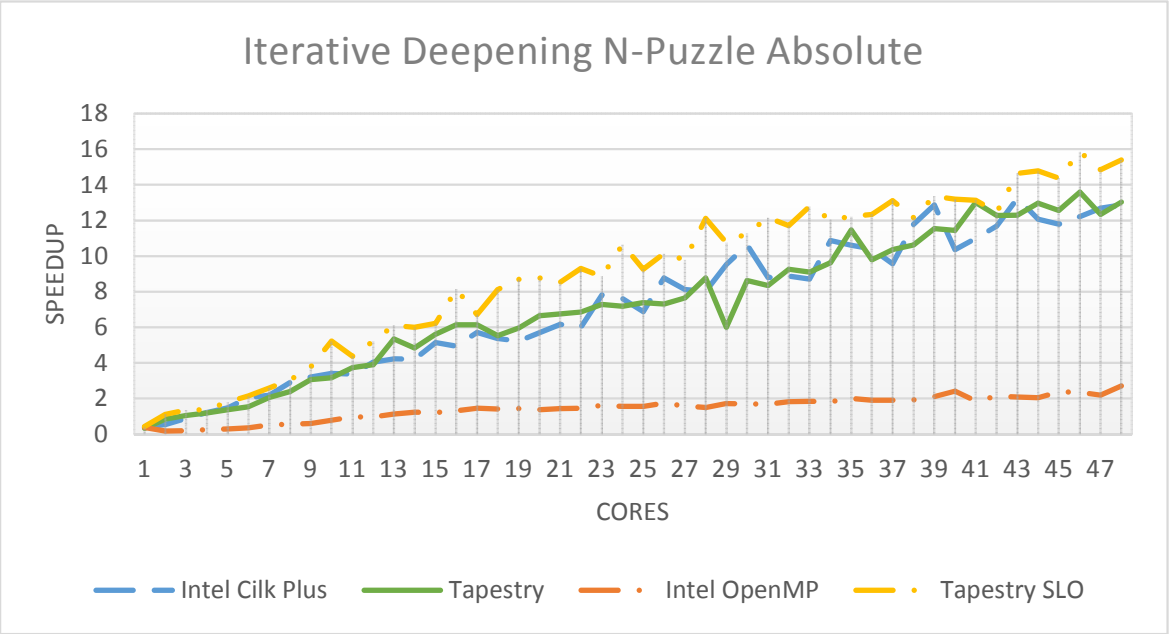


Figure 7.8: N-Puzzle Iterative Deepening Scalability: Solution was at a depth of 18 for a 3x3 puzzle.

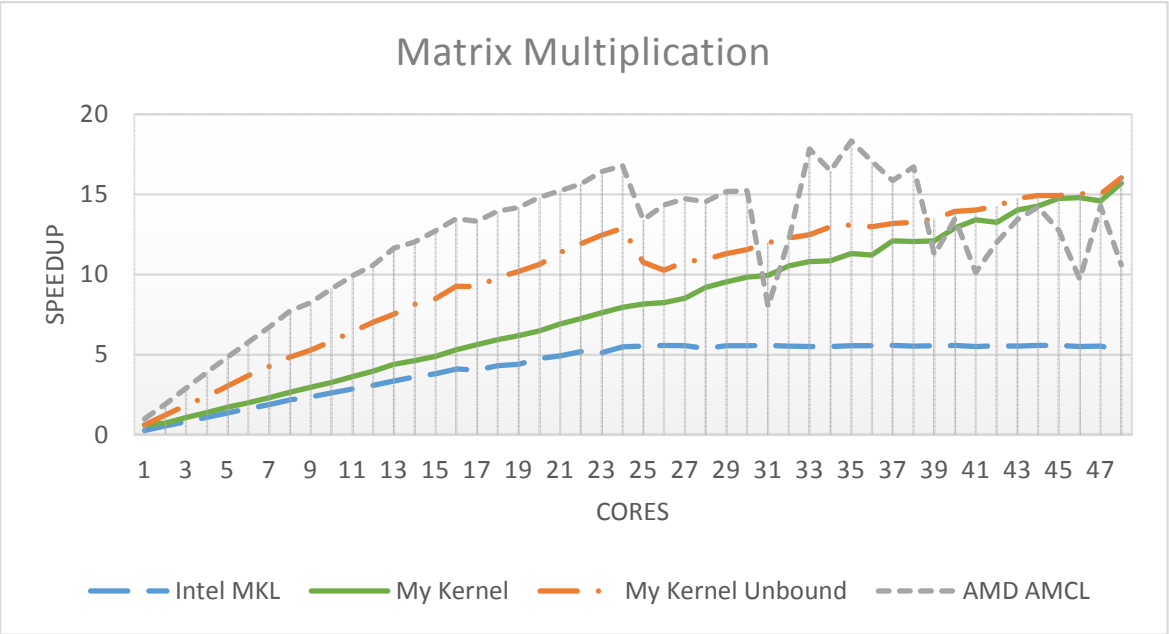


Figure 7.9: Matrix Multiplication Kernel Static Scalability: 16640x16640 matrices were used.

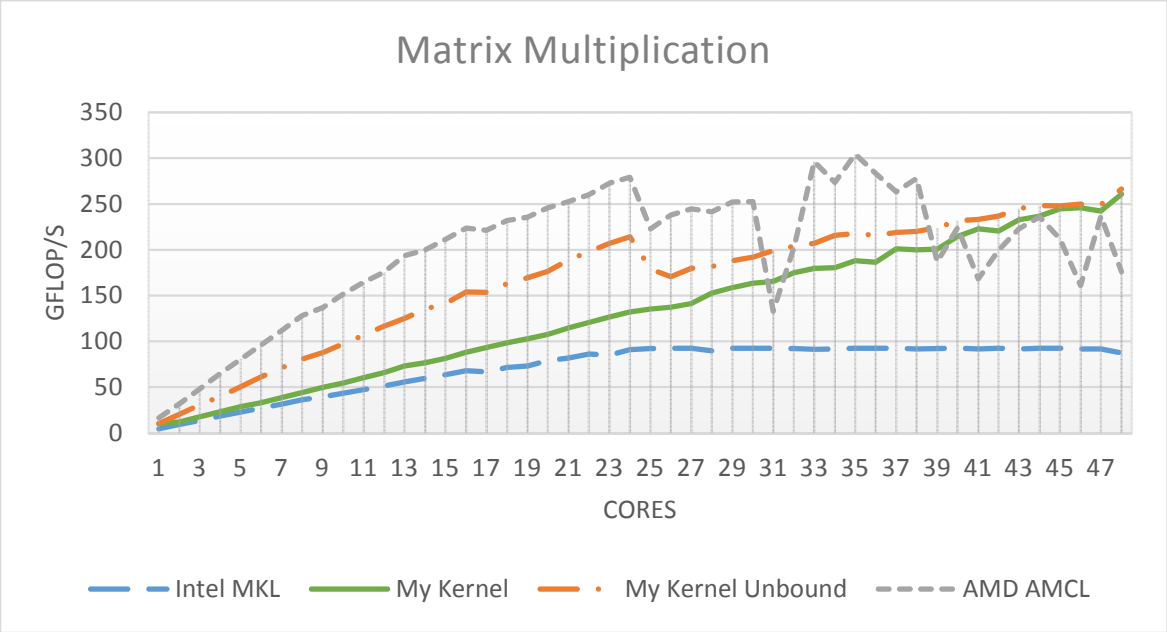


Figure 7.10: Matrix Multiplication Kernel Static FLOP/s: 16640x16640 matrices were used.

of pinning to consecutive cores. This is due to the NUMA layout of the node which allows greater bandwidth when not pinning. The baseline for the scalability graph seen in Figure 7.9 is the 16 GFLOP/s AMCL sequential version. In these tests my kernel utilizes the 128-bit wide FMA4 instructions is compiled with PGI's pgcc 12.9 with O3 and fastsse optimizations on.

My kernel performs griding with a size set 256x256 doubles for the grid width and height. In addition, I register tile each grid block. I chose a tile size of 2x4 empirically. It was the best performing. Furthermore, the tiled multiplication is vectorized with FMA4 instructions. Figure 7.10 shows the GFLOP/s per core. My kernel achieves a performance of 267 GFLOP/s on 48 cores whereas MKL only achieves 95 GFLOP/s. AMD's reaches 302 GLOP/s but only at 35 cores and inconsistently. The max scalability of these tests is around 24X due to the machine having only one FPU per 2 cores and AMD achieving near peak on 1 core because it utilizes the 256-bit wide instructions.

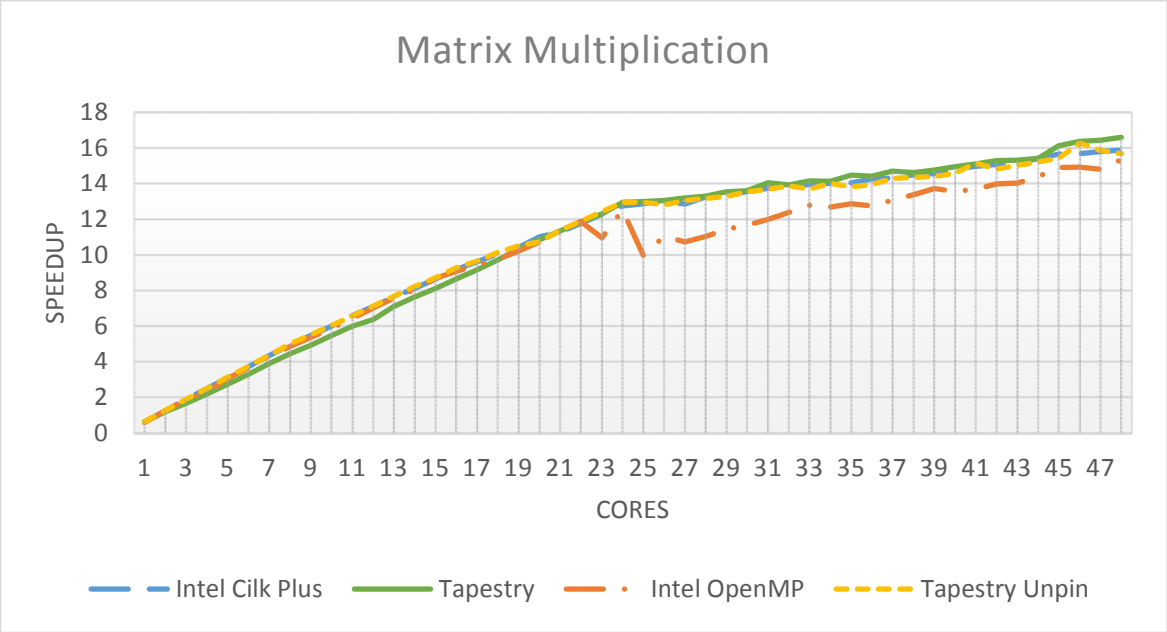


Figure 7.11: Matrix Multiplication Scalability: 16640x16640 matrices were used.

7.3.2.7 Matrix Multiplication Scalability

Using my matrix multiplication kernel presented in the last section, I took the kernel and applied it to Intel OpenMP, Cilk Plus, and Tapestry using the same problem size. Here I am using a fully dynamic schedule with Cilk Plus and Tapestry, but OpenMp uses a completely static schedule. The first thing to note is the scalability of Tapestry when compared with Cilk Plus, OpenMP, and Tapestry when not pinned as seen in Figure 7.11. Tapestry has linear speedup. However, Cilk Plus, OpenMP, Tapestry Unpin have almost perfect speedup till 24 cores where they take a significant dive in performance. This is because none of these libraries pin threads to cores; so, the threads can be scheduled by the every other core by the OS. The performance hit is probably due to sharing a FPU and the L1 instruction cache by every two cores. Tapestry when pinned is doing so to every other core to achieve a similar speedup. Cilk Plus gets around 263 GFLOP/s and Tapestry gets around 272 GFLOP/s as seen in Figure 7.12. Finally, OpenMP only achieves about 250 GLOP/s.

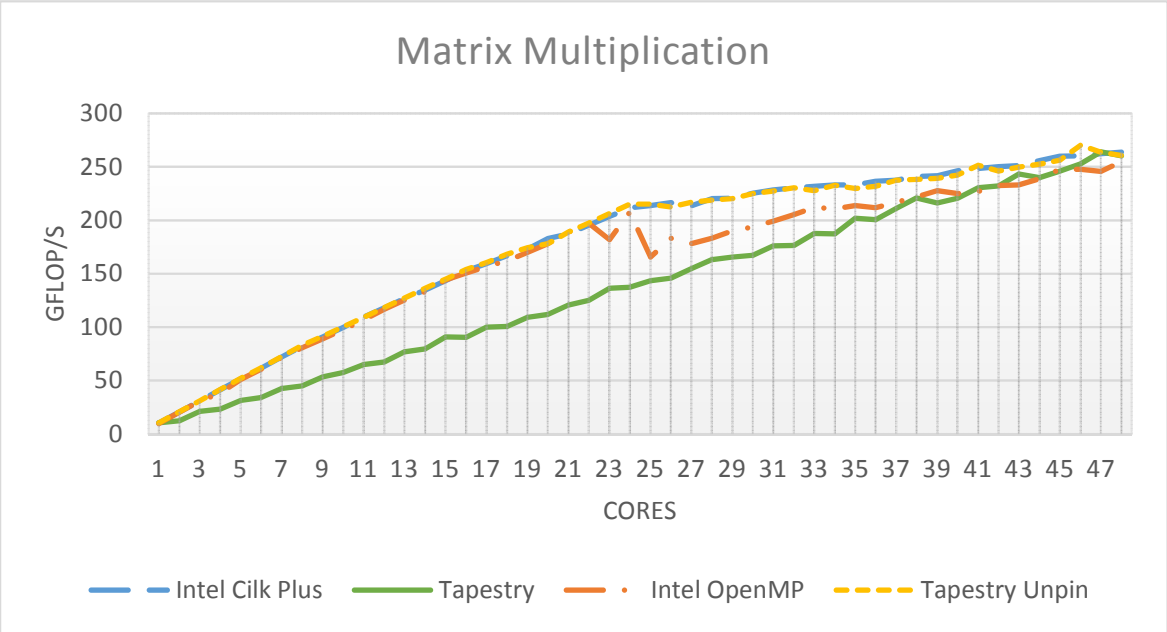


Figure 7.12: Matrix Multiplication with Cache-based Atomic Deque FLOP/s: 16640x16640 matrices were used.

Chapter 8

RELATED WORK

8.1 Microsoft Task Parallel Library

The Task Parallel Library (TPL) [57] is a library to provide APIs and parallel types (such as a concurrent stack) to Microsoft .NET Framework 4 System Threading and System Threading Tasks namespace to simplify adding parallelism and concurrency to applications. These are much in the same vein as Intel's TBB for C or C++, but for Microsoft .NET. TPL's underlying runtime will scale the concurrent work dynamically striving for the most efficient use of processors and resources. To accomplish this the TPL handles work partition, thread scheduling, state management, and other low-level details. However, it is noted that the TPL, like all .NET code is run in a virtual machine akin to Java's. Language support includes VB.NET, C#, and F#.

Microsoft is advocating that TPL is much better way to write parallel code instead of using their standard System Threading and System Threading Tasks interfaces to manually handle things. Furthermore, to effectively use the TPL they recommend that users have a basic understanding of locks, deadlocks, and race conditions. The TPL is divided into three areas: Data Parallelism, Task Parallelism, and Dataflow.

Data Parallelism is similar to the type of parallelism traditional OpenMP exploits where the same operation is performed on different data. To accomplish this, TPL provide parallel `for` and `foreach` statements that can be overloaded to control/-monitor loop states, maintain thread-local state, control the degree of concurrency, et cetera. The data of these loops are partitioned so multiple threads can operate on different segments concurrently.

Task parallelism is similar to OpenMP's task framework or Cilk and is designed for more independent tasks that are running concurrently. Tasks run on top of regular

threads. The framework uses work stealing [57] for load balancing of threads and hill climbing[58] to adjust the number of threads to maximize throughput of the tasks. Furthermore, the framework supports nesting or creating of children tasks as well as continuing tasks a la EARTH style as opposed to the use of callbacks to accomplish this.

Lastly, the Dataflow Library promotes an actor based programming model with coarse-grained dataflow or pipelined tasks. The model allows for the linking of operation blocks together to communicate asynchronously and perform tasks as data becomes available. For instance, they could be used to process images frame by frame as they come in from a camera using a pipeline to increase throughput.

Tapestry differs from this because Tapestry does not use a virtual machine, works for more fine-grain tasks, utilizes C++, and aims to work on distributed machines.

8.2 Intel Concurrent Collections

Intel Concurrent Collections (CNC) [59] is a library for C++ programs where semantic dependencies of an algorithm are defined to give an order. This is akin to dataflow, where the user expresses high-level computational steps including inputs and output, but not where these steps should be executed. These dependencies indicate what steps can be done in parallel. The underlying runtime and tuning layer will handle where and when these steps will be completed.

Code within each computational step is written with C++ standard constructs. Data is local to a computational step or is explicitly produced, received, and consumed. The CNC model supports the big three levels of parallelism: data, task, and pipeline. The point of CNC is to keep the interface the same across many different architectures such as shared memory and distributed systems while the underlying runtime may change or support various scheduling dynamics such as static or dynamic.

Like the TPL, the point of CNC is to ease the transition to parallel architectures for programmers not familiar with parallelism. This is seen in the fact that the user

does not worry about parallelism, but just expressing their codes dependencies through separation of the application and the optimization of said application.

CNC uses three constructs to achieve algorithmic construction: tag-collections, steps, and item-collections. Tags indicate what items are needed for a step to begin. Steps are the actual computation. And items are what are produced or taken in for each step. Dependency information between steps is gleamed by tag prescriptions.

Tapestry differs from CNC because Tapestry is a true actor based model that works for more than just consumer/producer style applications.

8.3 OpenMP

OpenMP [53] is extension to the C language using only preprocessor directives to parallelize code. Specifically, pragmas are used to instruct the compiler to generate parallel regions. Specifically, the parallelism uses fork/join concept or parallelism where the master thread forks a number of slave threads to complete work. The slave threads will join back into a master thread once the work completes. Then, the master will create more slave threads. It requires a special library and compiler to achieve these effects otherwise the code will run in serial. OpenMP is especially known for its data parallelism constructs to achieve good performance on shared memory machines. These allow for parallel loop constructs, parallel blocks, sharing data, expressing if conditions, reductions, critical sections, locks, barriers, atomic operations, scheduling, et cetera. Some of these constructs are only available when using the runtime library routines and types instead of the compiler directives. These include locks, timing functions, scheduling, and setting up the environment.

OpenMP 3.0 introduced task level parallelism to its model. In general, tasks can be spawned and joined on using the pragmas OpenMP is popular for. This style of parallelism is similar to Cilk.

OpenMP has number of well-known drawbacks that include the requirement of a compiler, no compare and swap, lack of mechanisms to the map threads to processors, and a large startup time.

Tapestry differs from OpenMP because it does not require a compiler and supports dependencies.

8.4 Habanero C

Habanero-C (HC) is a language developed by Rice University the follows closely to the design of Habanero-Java. The language is very similar to Cilk except it adds a flexible barrier called phasers, Data-Driven Futures, and Hierarchical Place Trees. The futures are very similar to those provided in C++ and Tapestry. They can be used to simulate dataflow-like dependencies by indicating that a thread should wait to start based on those futures; However, those futures are not part of the thread and require explicit signaling. This means the signaling could be aliased, is dependent on branch conditions, and could be met anywhere at any time by any thread. In addition, these futures require waiting.

Tapestry is different from HC because Tapestry doesn't require a compiler and supports dependencies on threads without the need for preemption. I addition, because the dependencies are tied to the thread the scheduler can best choose how to run the thread. Tapestry does not require the runtime to wait for the threads to be met, but utilizes direct signaling to start a thread.

8.5 Unified Parallel C

Unified Parallel C (UPC) [60, 61] is an extension of ANSI C to allow for parallelism that is based on the partitioned global address space (PGAS) programming model [62] (distributed shared memory programming model). UPC extends the features of C to include parallelism using SPMD with global memory access that can be local and remote. To accomplish this UPC virtualizes the address space across processors into one space. Writing and reading to memory is done with simple statements like in C, but the `shared` keyword must be used to indicate that the variable resides in or points to the global address space. UPC uses a simple parallel loop construct to parallelize workloads and allows for synchronization control via barriers, locks, and

memory consistency control. It builds upon the experiences of Split-C, AC, and PCP. Because UPC is pretty simple and the performance has shown to be good, vendors have developed UPC compilers for commercial purposes.

Specifically, HP, SGI, Sun, and Cray platforms have compilers. Also, an open-source implementation and compiler known as MuPC is available from Michigan Technological University for X86 Intel Architecture. Additionally, BUPC for various platforms including X86-64 Intel Architecture are available from the University of California Berkley. Furthermore, GNU UPC (open-source) is available and supports various architectures including the Cray T3E.

UPC key features include efficient mapping from language to machine architecture, minimization of thread communication overhead by exploiting data locality, and simplified remote memory access.

UPC has a distributed shared memory programming model that is similar to a regular shared memory model, but also has the ability to make use of data locality. Specifically, the distributed shared memory model divides its address space into various partitions where each memory partition has an affinity to a certain thread. This means, that a shared object that has an affinity for a certain thread will reside in that thread's local shared memory space.

The UPC view of memory is partitioned into private and shared spaces. Each thread has its own private space, as well as a portion of the shared space. The shared space is divided into a number of partitions per thread of which each has an affinity to a particular thread i.e. the partition is located in the thread's logical memory space. The use of the new `share` keyword for shared data allows for easy blocking of shared data and arrays across these threads.

A UPC shared pointer can reference any location in the shared address space. Whereas a private pointer may reference only addresses in its local portion of the shared space or its private space. Both static and dynamic memory allocations occur via mallocs or by using special keywords in the compiler. Both are supported for shared and private memory.

Data distribution in UPC is pretty simple. Because of the distributed shared memory programming model, UPC can share data among its threads using simple typing information. For instance, to share an array of size N equally among the threads, the user simply defines the array as shared using the `share` keyword, and UPC will distribute the array elements in a round robin fashion.

Finally, because shared memory is accessible by all threads, synchronization features are present in UPC to give a sequence or order to the access of memory. UPC allows the user to switch between strict or relaxed memory consistency models at the variable level, in code sections, or for the scope of the entire program. In addition to this, users can use locks to guarantee access of certain data by only one thread. Lastly, barrier are available to synchronize all threads at certain points in the program before new steps can be done.

Tapestry differs from UPC because it does not require a compiler and supports dependencies.

8.6 X10

The X10 programming language [63] designed for parallelism using a PGAS similar to UPC, but is far more complex and unique allowing for multiple styles of programming. It came from out of the DARPA-led High Productivity Computing Systems program (HPCS). In general, X10 borrows its style from a number of languages, but focuses on asynchrony, locality, atomicity, and order. Specifically, order is created through type-safe, class-based, and object oriented design. Furthermore, it supports fork-join programming, GPU programming, SPMD computations, phased computations, MPI-style communication, active messaging, and cluster programming [63]. X10 implementations are available for Power and x86 based architectures as well as all major operating systems including Linux, AIX, MacOS, Cygwin, and Windows.

X10 couples a more abstract version of the PGAS model with parallel programming. In X10, a computation is divided among a set of places. Each place holds a part of the data and contains one or more operations that act on that data. The type system

is constrained in the form of dependent types [64] for object-oriented programming. A dependent type depends on a value. These dependent types come in the form of parent and child relationships for operations. These relationships are used to prevent deadlocks that can occur when two or more process wait for each other to finish before they can complete. Operations can spawn children operations who themselves may have children. Parents can wait on children, but children cannot wait on their parents. Thus, waiting is cycle free. X10 also features global distributed arrays, ways to do unstructured and structured parallelism, and user defined primitive structure types.

Tapestry differs from X10 because it does not require a compiler and supports dependencies.

8.7 Chapel

Chapel [65] is parallel programming language developed by Cray Inc. as part of HPCS. Chapel was designed to improve the programmability of large-scale parallel computers while matching or exceeding the performance and portability of current programming models. It is open-source with contributions from academia, industry, and scientific computing centers. Chapel is designed to improve the productivity of high-end programmers while also being a portable parallel programming model. That means Chapel can be used on desktop multi-core machines as well as commodity clusters.

Chapel supports using high-level abstraction for task parallelism, data parallelism, and nested parallelism to create a multithreaded execution model. Like X10, Chapel allows for reasoning and specifying placement of data and tasks on architectures using local types in order to tune for locality. Chapel, like other parallel programming models, has a global-view of data with user defined aggregate implementation operations on distributed data structures allowing them to be expressed in a natural manner. Chapel has a multi-resolution code allowing users to write very abstract code, and they then can add more detail incrementally until the code is as close to the machine as they need. To help facilitate rapid prototyping and code reuse, Chapel supports object-oriented design, type inference, and features for generic programming.

Cray Inc. designed Chapel from the ground up rather than extending an existing language. Chapel was made to be easy to learn for users of C, C++, Fortran, Java, Perl, Matlab, and other popular languages because it is an imperative block-structured language. However, Chapel's parallel features are most directly influenced by ZPL, High-Performance Fortran (HPF), and the Cray MTA/Cray XMT extensions to C and Fortran.

Tapestry differs from Chapel because it does not require a compiler and supports dependencies.

8.8 Fortress

Fortress [66] like X10 and Chapel was designed for HPCS and is a programming language for high-performance computing designed with for high programmability like all programming languages. Like Chapel, Fortress has been designed from scratch without ties to legacy language semantics or syntax. Like other HPC languages, parallelism is built into the core of the language as well as the ability to specify locality and do transactions. Fortress also enables compiler optimizations across library boundaries with a component system and a test framework that facilitate program assembly and testing. Fortress has a small core language with much of the language encoded (e.g. arrays and other basic types) in libraries atop the core for extendibility and future proofing. That means future additions to the language can be easily supported. Additionally, Fortress supports mathematical notation, static checking of properties including physical dimensions or properties, static type checking of multidimensional arrays and matrices, and definitions of domain specific libraries.

Fortress is a general-purpose programming language. Its support for large-scale parallelism and management of data locality, its use of mathematical notation for syntax, and its static checking of units and dimensions (among other things) make it particularly well suited to high-performance computing. Fortress is also designed to be both highly parallel and have rich functionality contained within libraries, drawing from Java but taken to a higher degree. For example, the for loop is a parallel operation,

which will not always iterate in a strictly linear manner depending on the underlying software and hardware. However, the for loop is a library function and can be replaced by another for loop of the programmer's liking rather than being built into the language.

Fortress' syntax emulates mathematical notation as closely as possible. It has a novel type system to better integrate functional and object-oriented programming. It allows for specification of data distribution through the use of special data structures, it supports static checking of physical units and dimensions, it supports embedding of domain-specific language syntax in programs, and it includes a component system to facilitate the process of compiling, linking, and deploying programs. Syntactically, it most resembles Scala, Standard ML, and Haskell. Fortress is being designed from the outset to have multiple syntactic stylesheets. Source code can be rendered as ASCII text, in Unicode, or as a prettied image. This will allow for support of mathematical symbols and other symbols in the rendered output for easier reading.

Tapestry differs from Fortress because it does not require a compiler and supports dependencies.

8.9 Coarray Fortran & Coarray Fortran 2.0

Co-array Fortran [67, 68] (CAF) extends Fortran 95 for Single Program Multiple Data (SPMD) parallel processing. It adds architecture-independent syntax that can be used for distributed memory, shared memory, and cluster machines. It is an explicit notation for data decomposition that strives to be Fortran-like.

Specifically, the CAF SPMD parallel programming model uses a small set of extensions to Fortran 90 to support access to non-local data using lightweight and flexible synchronization primitives, pointers, and dynamic allocation of shared data. Executing CAF programs because they are SPMD are a static collection of asynchronous process images. CAF programs explicitly manage locality, data and computation distribution like MPI programs, but it is a shared-memory programming model based upon one-sided communication. The idea is that, CAF programs can directly reference off-processor values using extensions for subscripted references rather than explicitly

coding messages to exchange off-processor data. Because remote data access and synchronization are part of language, communication and synchronization are amenable to compiler-based optimizing transformations. The model has the following shortcomings according to Rice: there is no support for processor subsets, coarrays must be declared as global variables, the coarray extensions lack any notion of global pointers, reliance on named critical sections for mutual exclusion hinders scalable parallelism, Fortran 2008's sync images statement doesn't provide a safe synchronization space, there are no mechanisms to avoid or tolerate latency when manipulating data on remote images, and there is no support for collective communication [69].

To address the shortcomings of the model, Rice University is developing a clean-slate redesign of the Coarray Fortran programming model [69]. Rice's design for Coarray Fortran, which they call Coarray Fortran 2.0, is an expressive set of coarray-based extensions to Fortran designed to provide a productive parallel programming model. Compared to the emerging Fortran 2008, Rice's new coarray-based language extensions include some additional features:

- The ability to process subsets known as teams, which support coarrays, collective communication, and relative indexing of process images for pair-wise operations.
- Support for topologies, which augment teams with a logical communication structure.
- Allow for dynamic allocation/deallocation of coarrays and other shared data.
- It allows for local variables within subroutines: declaration and allocation of coarrays within the scope of a procedure is critical for library based-code.
- Team-based coarray allocation and deallocation is added.
- Global pointers are added in support of dynamic data structures.
- There is enhanced support for synchronization for fine control over program execution.
- Safe and scalable support for mutual exclusion, including locks and lock sets.
- Addition of events, which provide a safe space for point-to-point synchronization.

Tapestry differs from CAF because it does not require a compiler, supports dependencies, and utilizes C++.

8.10 Global Arrays

The Global Arrays (GA) [70] toolkit is designed to be an efficient and portable shared-memory programming interface for distributed-memory computers. Each MIMD program consists of a number of processes distributed across any number of computers. Each process can asynchronously access logical blocks of distributed dense multi-dimensional arrays without explicit cooperation or communication. The GA model exposes programmers to the non-uniform (NUMA) characteristics of high performance computers indicating local shared data is faster to access than remote. The primary target architectures for which GA was developed are massively-parallel distributed-memory and scalable shared-memory systems.

GA can complement the message-passing programming model rather than explicitly replacing it (i.e. the strengths of both can be utilized to enhance performance), and GA supports legacy libraries which means existing message passing libraries can be taken advantage for use in GA programs. This includes the Message Passing Interface (MPI).

The GA toolkit provides the typical shared memory style programming environment with a distributed array data structure known as a global array. A global array can be used as if it was stored in shared memory. The global array object encapsulates all the details of data distribution, addressing, and data access from the programmer. However, users can obtain information about the data distribution and locality if data locality is important.

The basic shared memory operations supported include the typical `get`, `put`, `scatter`, and `gather`. These operations are complemented by atomic read-and-increment, accumulate (reduction operation that combines data in local memory with data in the shared memory location), and lock operations. These operations are only supported when access data in global arrays rather than arbitrary memory locations. Thus, at least one global array has to be created before data transfer operations can be used. These GA operations are one-sided and will complete regardless of actions taken by the remote processes or processor that own the referenced data. In particular, GA does

not offer or rely on a polling operation or require inserting any other GA library calls to assure communication progress on the remote side. The GA library supports two different programming styles: task-parallel and data-parallel.

Tapestry differs from UPC because it adds dependencies to threads these dependencies do not need global memory to communicate. However, Tapestry can be complemented by the model.

8.11 HPX

The HPX runtime system is a modular, feature-complete, and performance-oriented representation of the ParalleX execution model. The model is targeted at conventional parallel computing architectures such as SMP nodes and commodity clusters. HPX incorporates routines to manage lightweight user-threads in addition to providing an Active Global Address Space (AGAS). HPX is implemented in C++11 and utilizes over 20 Boost and candidate Boost libraries.

The four principal properties exhibited by ParalleX are:

- Exposure of intrinsic parallelism, especially from meta-data, to meet the concurrency needs of scalability by systems in the next decade.
- Intrinsic system-wide latency hiding for superior time and power efficiency.
- Dynamic adaptive resource management for greater efficiency by exploiting runtime information.
- Global name space to reduce the semantic gap between application requirements and system functionality both to enhance programmability and to improve overall system utilization and efficiency.

HPX provides an experimental conceptual framework to explore the goals of ParallelX. The evolution of ParalleX has been, in part, motivated and driven by the needs of the emerging class of applications based on dynamic directed graphs. Such applications include scientific algorithms like adaptive mesh refinement, particle mesh methods, multi-scale finite element models and informatics related applications such as graph explorations. They also include knowledge-oriented applications for future web search engines, declarative user interfaces, and machine intelligence.

Tapestry differs from HPX because it does not need a global address space, and it does not need external libraries.

Chapter 9

CONCLUSION & FUTURE WORK

9.1 Conclusion

Overall Tapestry provides many contributions to software runtime design. Using it as a standard thread runtime without considering its plethora of features, Tapestry's performance and scalability are on par or exceed that of industry level software. In addition, Tapestry provides an intuitive simple extension to the standard C threading model allowing for data driven threads and synchronization of work at many different levels. These features include the control of the runtime and all the features of codelets. Tapestry also allows for a new features called dynamic split-phase transactions and multiple continuations on these codelets. Lastly, Tapestry is modular, portable, compiler free, and can be run on top of any type of threading model using any type of synchronization features.

Additionally, Tapestry's design allows for extension and exploration of new models of execution that are not limited to traditional thread, work stealing, or Codelet Model of execution. This means Tapestry can adapt and change where other inflexible models of execution will break as new architectures are introduced in the future.

9.2 Future Work

For the future, I would like to provide a simulation framework to run Tapestry on to explore various future exascale architectures effects on features provided by Tapestry. Furthermore, I would like to provide a graphical language so users can build codelet graphs within their programs without the use of language semantics. I would like to perform more fine-grain optimizations to Tapestry Fibers, provide a framework to pipeline Threaded Dependencies, explore using Tapestry to run on top other models,

allow for dynamically switching between runtimes in real time, provide a tool to analyze sequential program performance to provide information of candidate functions that can benefit from Tapestry's fork/join fine-grain performance, and another tool translate sequential programs automatically into Tapestry using the Fork/Join model.

9.2.1 Threaded Dependencies

Although much of the framework for codelets comes from meshing dataflow with EARTH, more concepts are still need to be explored more thoroughly to further differentiate the model. These are: how to ensure dataflow pipelining (token level), codelet pipelining, and finally how these two optimizations interact.

9.2.1.1 Data Pipelining

The movement of tokens through the execution of loops, for merging, and in many other cases will benefit execution time immensely if pipelined correctly. A simple but interesting question arises, how should the model handle pipelining of data to benefit the overall execution of program assuming no codelet duplication?

Previous work in static dataflow pipelining shows pipelining loops in data-driven architectures does not need a scheduler for operations. It can be simply achieved by creating a structure connecting the operations of the loop body and the hardware structures responsible for the loop iterations [26, 71, 72]. In the codelet model a similar scheme should allow efficient execution of parallel loops that are data-driven. Such a structure would in theory even allow for dynamic pipelining of loops. For the codelet model consideration, codelets connected together in a loop need to be mapped to a module that would facilitate the reuse of codelets without duplication and allow the connection mapping to be pipelined ideally. A simple and easy way to avoid this problem however, is by duplicating codelets for consecutive iterations. But this doesn't facilitate the reuse of data structures between iterations and could be costly for loops with many codelets by resulting in too much parallelism. This solution would only allow

for static software pipelining because special signaling instructions would be needed to indicate another codelet can run and would require scheduling.

Another possible solution would require the use of centralized module wherever each codelet is executing. This does not need scheduling to be achieved. In this scheme the module would be a special hardware structure used to map addresses of codelets to executing units, but it could be created in the executing software. This model would allow within a close area of processors to dynamically execute each iteration of a loop with data reuse. This would facilitate software pipelining similar to ones found in static dataflow schema. The module would most likely need to map dependency arcs to special FIFO queues to be able to enable these features. As long as there were enough modules per area of processors the communication costs would be small and not congested. Overall this solution has the benefit of only needing to be implemented at the system code level. This means it does not require compiler support. However, the downside is the use of a specific module of code in the system may be slower than a static solution due to the overhead of the module.

9.2.1.2 Codelet Pipelining

Pipelining can be also tackled at the codelet level without thinking about how to meet the data-driven demands, but at a scheduling level to facilitate codelet parallel execution as opposed to data reuse. Such solutions would reuse traditional instruction level software pipelining and expand their concepts to many-core. These solutions are static only and would require duplicating actors and would not have reuse of signaling slots. When codelets execute in a loop schema, if the number of codelets per iteration is increased and reordered, the number of codelets per iteration execution increases. Thus, parallelism increases in theory. However, codelets should work like dataflow, and thus each iteration of codelets should be able to execute as long as dependencies are met.

It is important for a many-core system to have much parallelism, and we advocate that parallelism could be increased with these techniques. And these technique

should help traditional code loops that create codelets. The main idea being explored here is Single-dimension software pipelining at the multi-core level as done by Douillet [40]. In general these ideas require Lamport clocks and should run well across many cores. Basically the pipeline schedule produced by SSP is broken up and scheduled across many cores with the synchronization being done by the Lamport clocks. It appears this idea can be generalized to other software pipelining algorithms other than SSP. This is the only known multi-core software pipelining solution at the time and is very coarse grain. So a very important question to answer is, will this solution provide a benefit when instructions are coarsened in size like in codelets? I.E. will enough parallelism exist for benefits? Or is locality more important?

Perhaps a hybrid solution the combines Douillet's ideas with the features of dynamic data-driven pipeline will prove even more beneficial. Code could be reorganized statically using an SSP-like algorithm and given a schedule. Then, we chunk the iterations to be run in parallel. However, instead of using the Lamport clocks to synchronize we could establish a centralized module to execute N amount of chunks at a time on N number of cores that are connected in pipelined loop. This would allow the benefits of parallelism with the locality and reuse of data-driven pipelining. This would reduce parallelism amount of a maximum parallel schedule though. It's my personal belief that data should be kept local as much as possible and parallelism should be focus on using processors close to data as possible.

9.2.2 Additional

Finally, I would like to provide a detailed analysis of performance per optimization to detail how each optimization contributes to performance.

BIBLIOGRAPHY

[1] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, E. R. M. Ramanathan, V. Thomas, I. Corporation, and S. S. Pawlowski, "Intel Processor and Platform Evolution for the Next Decade Executive Summary," 2005. [Online]. Available: http://epic.hpi.uni-potsdam.de/pub/Home/TrendsAndConceptsII2010/HW_Trends_borkar_2015.pdf

[2] Nvidia, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[3] ——, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[4] Intel, "The Intel Xeon Phi Coprocessor 5110P," 2012. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/high-performance-xeon-phi-coprocessor-brief-2.pdf

[5] ——, "Transforming Mission-Critical Computing," 2011. [Online]. Available: http://www.intel.com/content/dam/doc/product-brief/xeon-e7-transforming-mission-critical-computing-brief.pdf

[6] Tilera, "TILE64 Processor Product Brief," 2008. [Online]. Available: http://www.tilera.com/sites/default/files/productbriefs/PB010_TILE64_Processor_A_v4.pdf

[7] Texas Instruments, "TMS320C66x Multicore DSPs for High-Performance Computing," 2011. [Online]. Available: http://www.ti.com/lit/ml/sprt619/sprt619.pdf

[8] AMD, "AMD Opteron 6000 Series Platform," 2012. [Online]. Available: http://www.amd.com/us/Documents/AMD_Opteron_6000_Comparison.pdf

[9] ARM, "ARM11MPCore Processor," 2012. [Online]. Available: http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php

[10] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams, "Ibm power7 multicore server processor," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 1:1 –1:29, may-june 2011.

[11] J. Duato, F. Silla, B. Holden, P. Miranda, J. Underhill, M. Cavalli, S. Yalamanchili, U. Bruning, and H. Froning, "Scalable Computing: Why and How," 2010. [Online]. Available: http://www.hypertransport.org/docs/uploads/HNC_WP_33976512.pdf

[12] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. Gao, "A Study of the On-Chip Interconnection Network for the IBM Cyclops64 Multi-Core Architecture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, april 2006, p. 10 pp.

[13] Intel, "An Introduction to the Intel QuickPath Interconnect," 2009. [Online]. Available: http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf

[14] K. B. Theobald, "Definition of the EARTH Model," 1999, technical Memo.

[15] G. R. Gao, J. Suetterlein, and S. Zuckerman, "Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond," April 2011, technical Memo.

[16] J. B. Dennis, J. B. Fosseen, and J. P. Linderman, "Data flow schemas," pp. 187–216, 1974. [Online]. Available: http://dl.acm.org/citation.cfm?id=646795.704865

[17] J. B. Dennis, "Programming Generality, Parallelism, and Computer Architecture," pp. 484–492, 1969.

[18] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966. [Online]. Available: http://link.aip.org/link/?SMM/14/1390/1

[19] J. E. Rodrigues and J. E. Rodriguez Bezos, "A Graph Model for Parallel Computations," Cambridge, MA, USA, Tech. Rep., 1969.

[20] D. A. Adams, "A Computation Model with Data Flow Sequencing," Ph.D. dissertation, Stanford, CA, USA, 1969, aAI6913919.

[21] P. Whiting and R. Pascoe, "A history of data-flow languages," *Annals of the History of Computing, IEEE*, vol. 16, no. 4, pp. 38 –59, winter 1994.

[22] J. M. P. Cardoso, "Dynamic Loop Pipelining in Data-Driven Architectures," pp. 106–115, 2005. [Online]. Available: http://doi.acm.org/10.1145/1062261.1062283

[23] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *SIGARCH Comput. Archit. News*, vol. 3, pp. 126–132, December 1974. [Online]. Available: http://doi.acm.org/10.1145/641675.642111

[24] A. Arvind and K. P. Gostelow, "The U-Interpreter," *Computer*, vol. 15, pp. 42–49, February 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=1318726.1319193

[25] G. M. Papadopoulos, "Monsoon: an Explicit Token-Store Architecture," pp. 82–91, 1990.

[26] Z. Paraskevas, "Code Generation for Dataflow Software Pipelining," Master's thesis, Montreal, Quebec, Canada, 1989.

[27] G. R. Gao, "A Pipelined Code Mapping Scheme for Static Data Flow Computers," Ph.D. dissertation, Cambridge, MA, USA, 1986.

[28] G. Gao, H. Hum, and Y.-B. Wong, "Parallel Function Invocation in a Dynamic Argument-Fetching Dataflow Architecture," in *Databases, Parallel Architectures and Their Applications,. PARBASE-90, International Conference on*, mar 1990, pp. 112 –116.

[29] J. Rumbaugh, "A Data Flow Multiprocessor," *Computers, IEEE Transactions on*, vol. C-26, no. 2, pp. 138 –146, feb. 1977.

[30] G. M. Papadopoulos, *Implementation of a General-Purpose Dataflow Multiprocessor*. Cambridge, MA, USA: MIT Press, 1991.

[31] D. E. Culler, K. E. Schauser, and T. v. Eicken, "Two Fundamental Limits on Dataflow Multiprocessing," in *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, ser. PACT '93. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, pp. 153–164. [Online]. Available: http://dl.acm.org/citation.cfm?id=647025.714362

[32] J. Ramanujam, "Optimal Software Pipelining of Nested Loops," pp. 335–342, 1994.

[33] B. R. Rau and J. A. Fisher, " Instruction-Level Parallel Processing: History, Overview and Perspective," 1992.

[34] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," pp. 63–74, 1994.

[35] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software Pipelining," 1995.

[36] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-dimension Software Pipelining for Multidimensional Loops," *ACM Trans. Archit. Code Optim.*, vol. 4, March 2007. [Online]. Available: http://doi.acm.org/10.1145/1216544.1216550

[37] H. Rong, A. Douillet, R. Govindarajan, and G. R. Gao, "Code Generation for Single-Dimension Software Pipelining of Multi-dimensional Loops," pp. 175–186, 2004.

[38] H. Rong, A. Douillet, and G. R. Gao, "Register Allocation for Software Pipelined Multi-Dimensional Loops," pp. 154–167, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065030

[39] A. Douillet and G. R. Gao, "Register Pressure in Software-Pipelined Loop Nests: Fast Computation and Impact on Architecture Design," 2005.

[40] ——, "Software-Pipelining on Multi-Core Architectures," pp. 39–48, 2007. [Online]. Available: http://dx.doi.org/10.1109/PACT.2007.64

[41] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Principles and Practice of Parallel Programming*, 1995, pp. 207–216.

[42] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," in *IEEE Symposium on Foundations of Computer Science*, 1994, pp. 356–368.

[43] Intel, "Intel Cilk Plus Language Extension Specification Version 1.1," 2011. [Online]. Available: http://software.intel.com/sites/default/files/m/4/e/7/3/1/40297-Intel_Cilk_plus_lang_spec_2.htm

[44] "Habanero Java," 2009. [Online]. Available: http://habanero.rice.edu/hj

[45] Intel, "Intel Thread Building Blocks Reference Manual," 2012. [Online]. Available: http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf

[46] J. Reinders, *Intel Threading Building Blocks - Outfitting C++ for Multi-Core Processor Parallelism*, 2007.

[47] A. Kukanov, "The Foundations for Scalable Multicore Software in Intel Threading Building Blocks," *Intel Technology Journal*, vol. 11, 2007.

[48] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors," in *International Symposium on Symbolic and Algebraic Computation*, 2007, pp. 15–23.

[49] "SWARM Beta." [Online]. Available: http://www.etinternational.com/index.php/products/swarmbeta/

[50] "SWARM Case Studies." [Online]. Available: http://www.etinternational.com/index.php/products/swarmbeta/swarm-case-studies/

[51] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron Processor for Multiprocessor Servers," *Micro, IEEE*, vol. 23, no. 2, pp. 66 – 76, march-april 2003.

[52] V. Sarkar and J. Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," in *Proceedings of the 1986 ACM conference on LISP and functional programming*, ser. LFP '86. New York, NY, USA: ACM, 1986, pp. 202–211. [Online]. Available: http://doi.acm.org/10.1145/319838.319863

[53] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1," May 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[54] D. Orozco, "TIDeFlow: A Dataflow-Inspired Execution Model for High Performance Computing Programs," Ph.D. dissertation, 2012.

[55] Intel, "Intel Core2 Duo Processor E6000 Series," 2011. [Online]. Available: http://download.intel.com/support/processors/core2duo/sb/core_E6000.pdf

[56] ——, "Intel Core i7-2600 Desktop Processor Series," 2012. [Online]. Available: http://download.intel.com/support/processors/corei7/sb/core_i7-2600_d.pdf

[57] D. Leijen, W. Schulte, and S. Burckhardt, "The Design of a Task Parallel Library," in *24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA'09), Orlando, FL*, Oct. 2009, also appeared in Sigplan Not., **44**(10) 227–242, 2009.

[58] Microsoft, "Task Parallelism (Task Parallel Library), year = 2011, url = http://msdn.microsoft.com/en-us/library/vstudio/dd537609(v=vs.100).aspx."

[59] Frank Schlimbach, "Intel Concurrent Collections for C++ 0.8 for Windows and Linux," 2012. [Online]. Available: http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc

[60] UPC Consortium, "UPC Language Specications V 1.2," 2005. [Online]. Available: http://upc.gwu.edu/docs/upc_specs_1.2.pdf

[61] W. W. Carlson, J. M. Draper, and D. E. Culler, "S-246, 187 Introduction to UPC and Language Specification."

[62] "Partitioned Global Address Space," 2012. [Online]. Available: http://www.pgas.org/

[63] Vijay Saraswat and Bard Bloom and Igor Peshansky and Olivier Tardieu and David Grove, "X10 Language Specication Version 2.2," 2012. [Online]. Available: http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf

[64] T. Altenkirch, C. Mcbride, and J. Mckinna, "Why Dependent Types Matter," in *In preparation, http://www.e-pig.org/downloads/ydtm.pdf*, 2005.

[65] Cray Inc, "Chapel Language Specication Version 0.9.1," 2012. [Online]. Available: http://chapel.cray.com/spec/spec-0.91.pdf

[66] Eric Allen et al., "The Fortress Language Specication," 2007. [Online]. Available: http://labs.oracle.com/projects/plrg/Publications/fortress1.0beta.pdf

[67] International Organization for Standardization, "Fortran 2008 Draft International Standard," 2010. [Online]. Available: ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf

[68] R. W. Numrich and J. Reid, "Co-Array Fortran for parallel programming," *ACM FORTRAN FORUM*, vol. 17, no. 2, pp. 1–31, 1998.

[69] J. Mellor-Crummey, L. Adhianto, and W. N. S. III, "A new vision for coarray Fortran," pp. 1–9, 2009.

[70] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apr, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *International Journal of High Performance Computing Applications*, vol. 20, pp. 203–231, 2006.

[71] J. B. Dennis, "First vVrsion of a Data Flow Procedure Language," pp. 362–376, 1974. [Online]. Available: http://dl.acm.org/citation.cfm?id=647323.721501

[72] G. R. Gao, H. H. J. Hum, and Y.-B. Wong, "Towards Efficient Fine-Grain Software Pipelining," *SIGARCH Comput. Archit. News*, vol. 18, pp. 369–379, June 1990. [Online]. Available: http://doi.acm.org/10.1145/255129.255177

Appendix A

ADDITIONAL BENCHMARK RESULTS

This section benchmarks Tapestry with many different configurations. It provides more details and analysis on Tapestry dependencies, performance of Tapestry dependencies when used as glue for OS threads, performance of Tapestry in a hyper-threading environment, and redundant graph elimination performance benefits.

For all these tests I benchmark only on X86-64. Initially, I compare Tapestry dependencies when it is configured to use the Tapestry Fibers runtime for Core 2 and Core i7. During the Core 2 tests, I compare Tapestry Runtime System using dependencies to the fork/join model. I vary for Tapestry thread count and core count showing that Tapestry is fully scalable. Lastly, I evaluate Tapestry dependencies when used with operating system threads compared to just operating system threads

A.1 Case Study for Core 2

These benchmarks are run using Core 2 platform and test every optimization available to Tapestry instead of just he best to see how these optimizations affect performance.

A.1.1 Runtime Micro-benchmarks

In this section we compare the performance of dependencies to that of threads for the same code. We note the performance of the dependency version to that of the none dependency version and show its slowdown based on work relativity.

A.1.1.1 Serial Overhead for Dependencies

In this figure we show that slowdown compared to the none dependency version comparing Locality Overdrive to Locality Overdrive dependencies and regular to

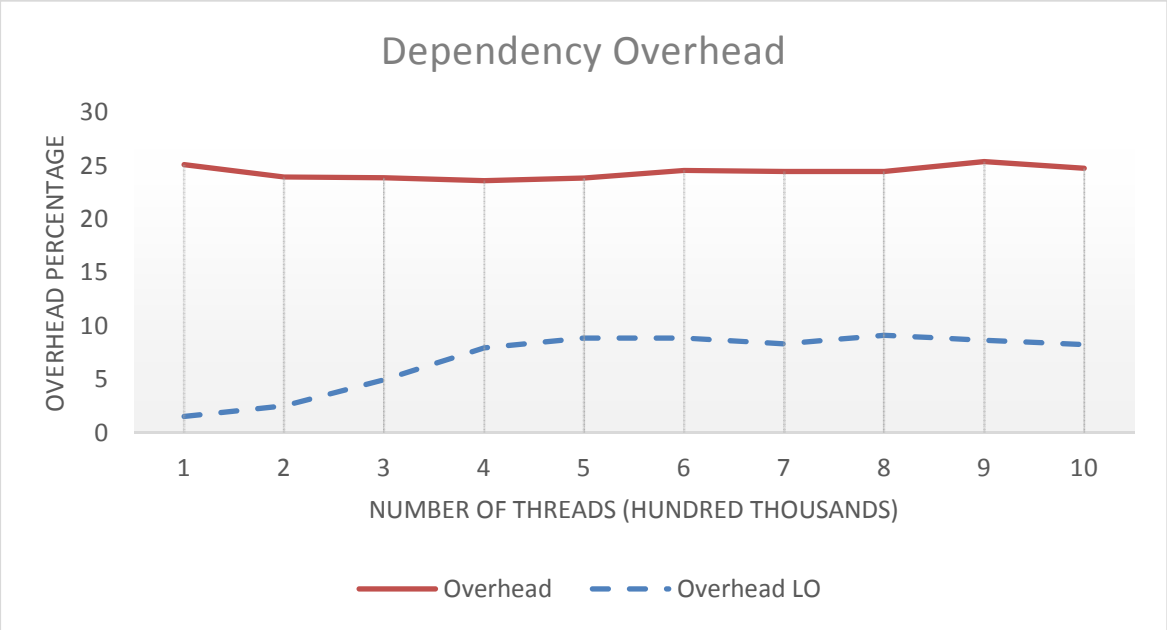


Figure A.1: Serial Overhead for Dependencies

regular dependencies with memory optimizations on. In this test, I do thread creation where recursion is used to create dependent threads vs using dependencies to do the same. This test is designed to compare only the overhead without the extra work factors due to split-phase transactions. We can see the overhead of dependencies is very low for LO and not terribly bad without any optimizations.

A.1.1.2 Fibonacci Overhead for Dependencies

In this Fibonacci test we expand upon the recursive algorithm by breaking each operation into two which makes the work even finer. As expected, the dependency version is about 1.5 to 2.5x slower (Figure A.2) due to having to execute two times the amount of threads at a even finer scale than the recursive version.

A.1.1.3 N-Queens Overhead for Dependencies

Similar to the Fibonacci test, we break the recursive algorithm into finer parts and do about double the amount of work. We note that N-Queens is much coarser

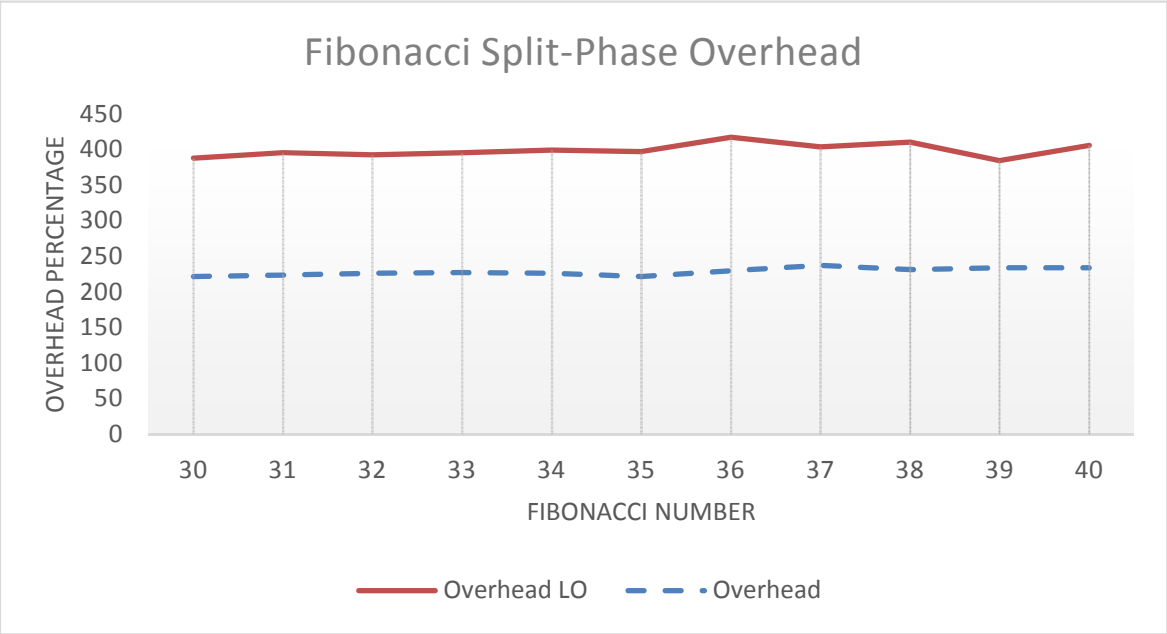


Figure A.2: Fibonacci Overhead for Dependencies

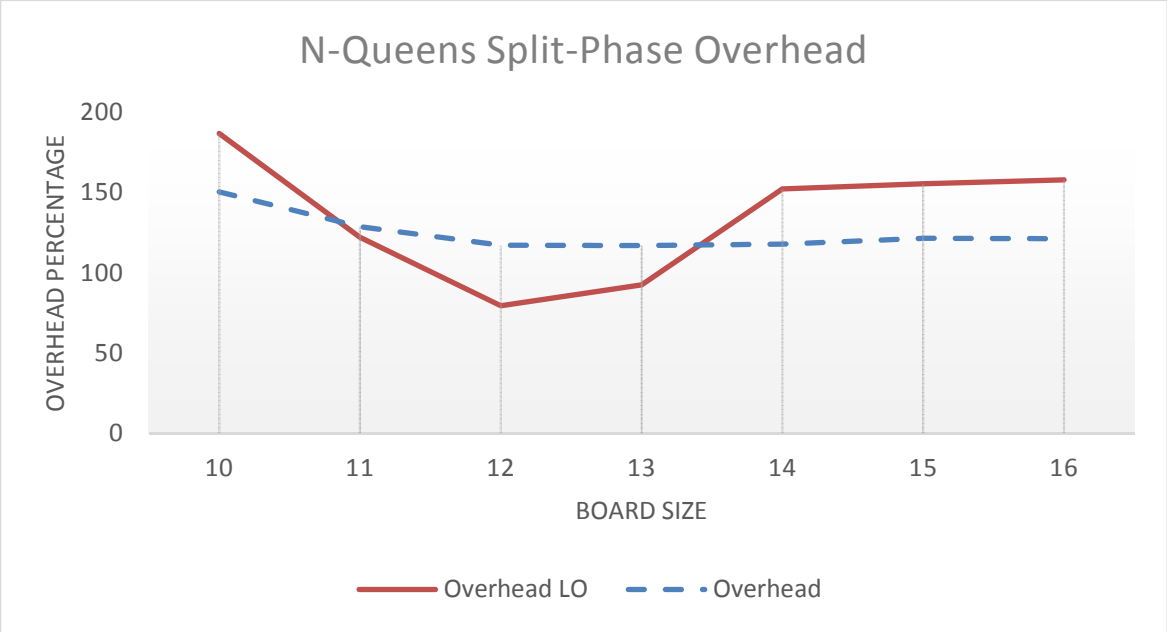


Figure A.3: N-Queens Overhead for Dependencies

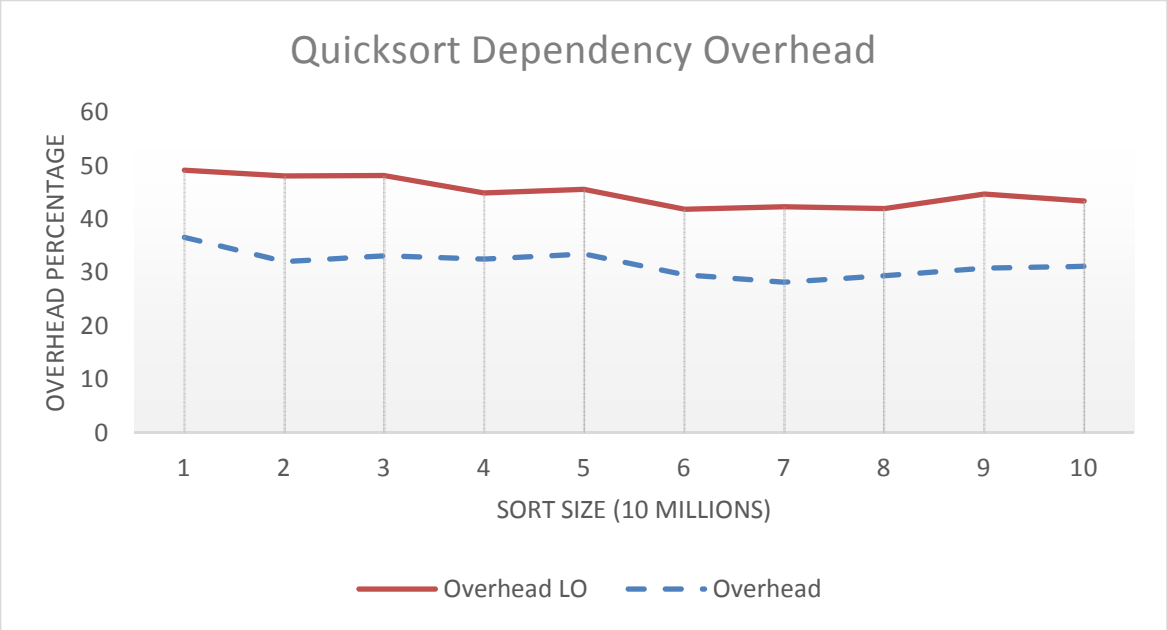


Figure A.4: Quicksort Overhead for Dependencies

compared to Fibonacci. As expected the results show that N-Queens performs around 1.5 to 2x slower as indicated in Figure A.3.

A.1.1.4 Quicksort Overhead for Dependencies

For this test, the Quicksort is 1.3 to 1.5x slower using the aggregate thread as shown in Figure A.4. Each thread has to signal the thread which creates lots of contention on the aggregate thread. If we switched the implementation into a tree barrier, the performance probably would be much better.

A.1.1.5 Monte Carlo Overhead for Dependencies

In Figure A.5, the results show that overhead is minimal and in some cases dependencies out perform threads. This is because when finishing, the first available core can execute the finish thread to report the results whereas in the non-dependency version, the core that created the threads waits to be woken and told everyone is done.

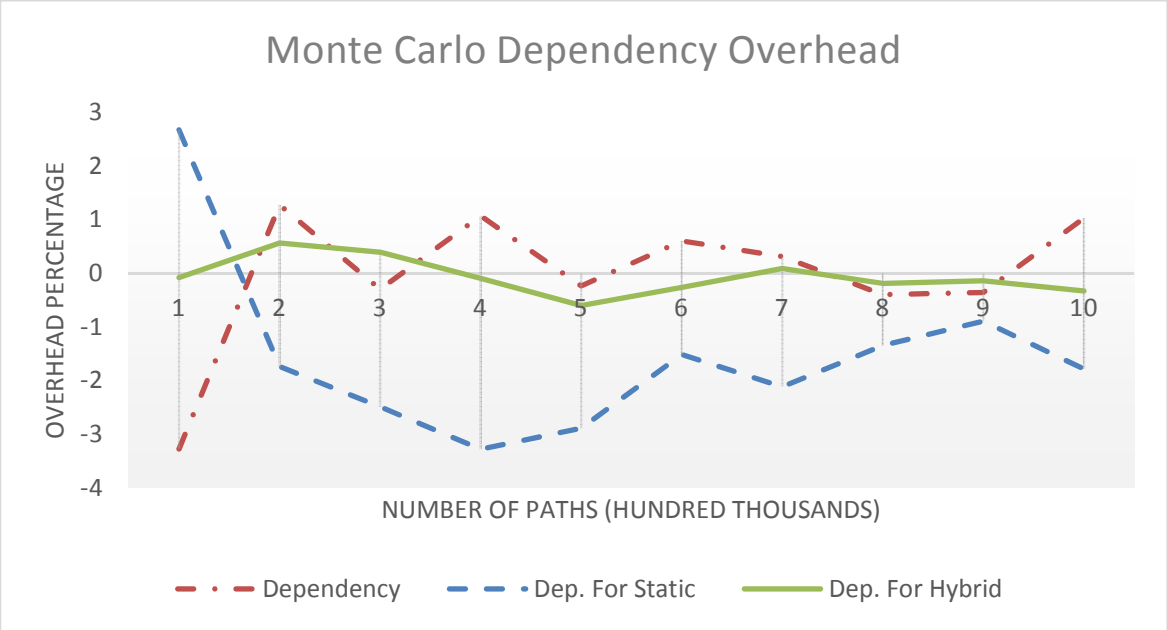


Figure A.5: Monte Carlo Overhead for Dependencies

Furthermore, the Monte-Carlo test is very macro scaled so the overhead of the signaling is minimized to the amount of work.

A.1.2 Runtime Benchmarks

In this section we benchmark the Super Locality Optimization (SLO) and Locality Optimization (LO) for fork/join algorithms on a dual core and quad core machines to show the advantage of our optimizations for multi-core machines. We compare well known runtime systems against Tapestry when it is configured with the Tapestry Fibers runtime system. The two runtime systems we compare against are Intel Cilk Plus and Intel OpenMP. These tests only use the Fork/Join model here. All the algorithms are exactly the same across all three runtime systems. All these test use the fast stack based memory optimizations. All the test in this section run Windows 8 using a Core 2 Duo at 2.4 GHz with 6GB of ram except the scalability tests near the end of the section which use a Core I7. For both systems, I use the Parallel Studio XE 2011 compiler for Windows on these tests with O2 optimizations.

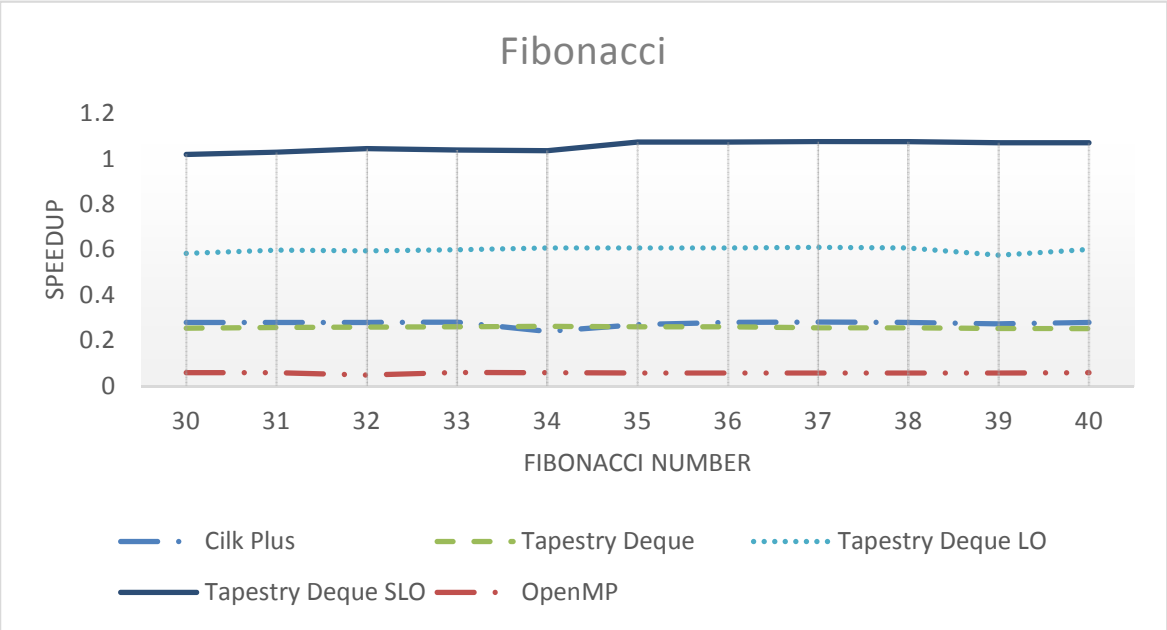


Figure A.6: Fibonacci Fork/Join

A.1.2.1 Optimizations for Dependencies

Memory optimizations refer to the use of a memory manager instead of the system memory manager to allocate memory for the dependencies. A dynamic split-phase transaction means that one branch is completed recursively for the operation using dynamic continuations, and the other is spawned as a thread.

A.1.2.2 Fibonacci Fork/Join

The Fibonacci test is a recursive very fine grain test with little data passed between threads. Here the results show that Tapestry and Cilk Plus are significantly faster than OpenMP's task framework for recursive tasks. We can see that by bypassing the lower runtime and binding the current threads to running core with a Locality Optimization (LO), we are able to achieve significant relative speedup compared to Cilk Plus and OpenMP. Additionally, when forcibly binding even further and skip the Tapestry Thread creation using a recursive call to implement the function with Super

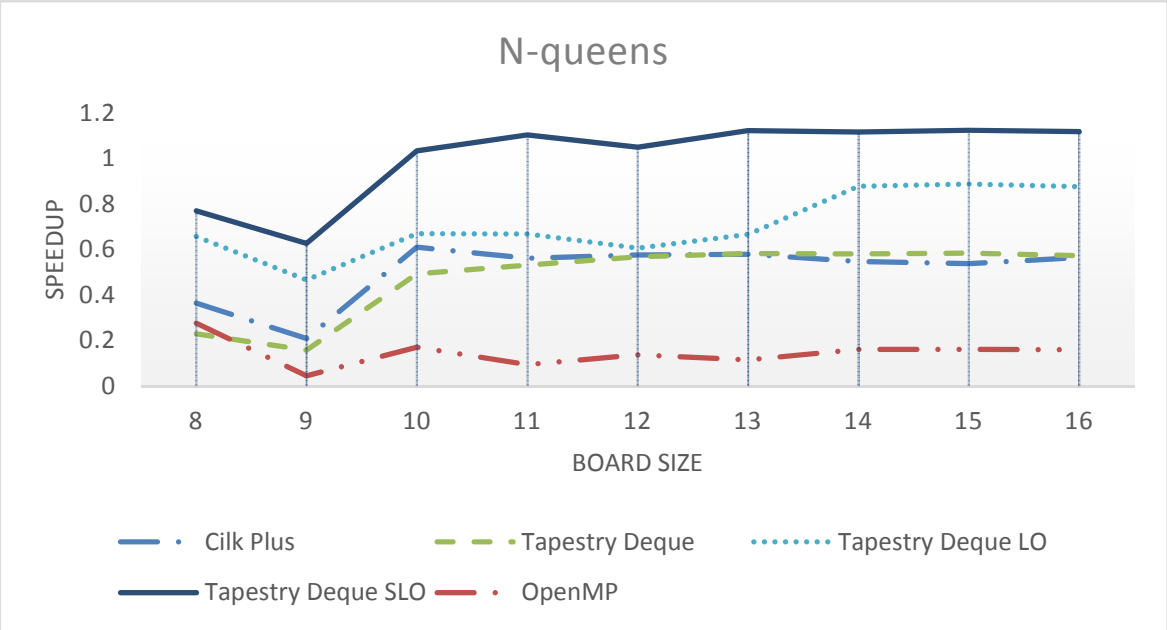


Figure A.7: N-Queens Fork/Join

Locality Optimization (SLO) we nearly double the speed again and achieve a nearly 5% gain over the sequential implementation.

The results seen in Figure A.6 show that Tapestry's performance is on par with Cilk Plus for fork/join Parallelism for just a small task. The results indicate that skipping the scheduler using LO increases performance over Cilk Plus and skipping memory management using SLO increases further performance. Since our model of parallelism is similar to Cilk Plus' these optimizations could be employed to improve Cilk Plus' performance for very fine grain tasks.

A.1.2.3 N-Queens Fork/Join

Here the benchmarks as seen in Figure A.7 indicate that for coarser tasks OpenMP is still significantly slower than Tapestry and Cilk Plus for a natural divide and conquer algorithm. Furthermore, Tapestry is still able to achieve around a 10% speedup over the sequential for relatively small and large problem sizes.

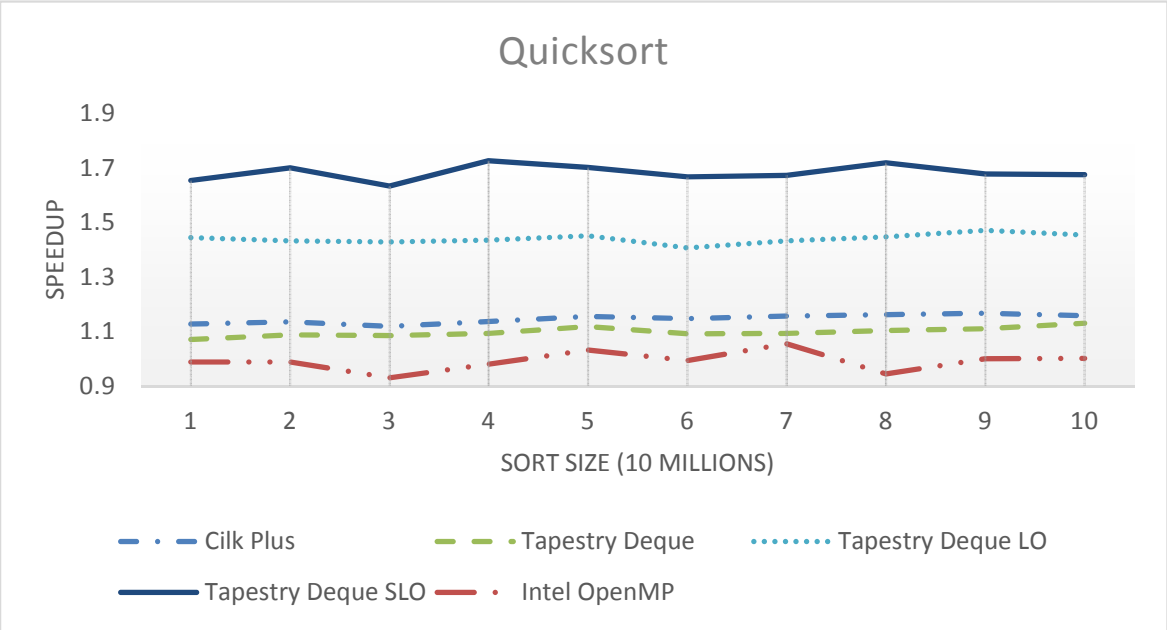


Figure A.8: Quicksort Fork/Join

A.1.2.4 Quicksort Fork/Join

Quicksort, with its natural mixture of coarse and fine-grain tasks, interestingly shows that OpenMP, Cilk Plus, and Tapestry works as well as a sequential implementation. However, Tapestry's performance with optimizations is impressive with around 1.7 speedup over the sequential task for all problem sizes as seen in Figure A.8.

Tapestry out performs Cilk Plus on a mix of coarse and fine grain tasks using the SLO and LO optimizations.

A.1.2.5 Monte Carlo Fork/Join

Our static scheduler for OpenMP already performs on par with Cilk Plus as indicated in Figure A.9, but the Hybrid scheduler that uses macro threads with various sizes for loop work per thread outperforms all the schedulers.

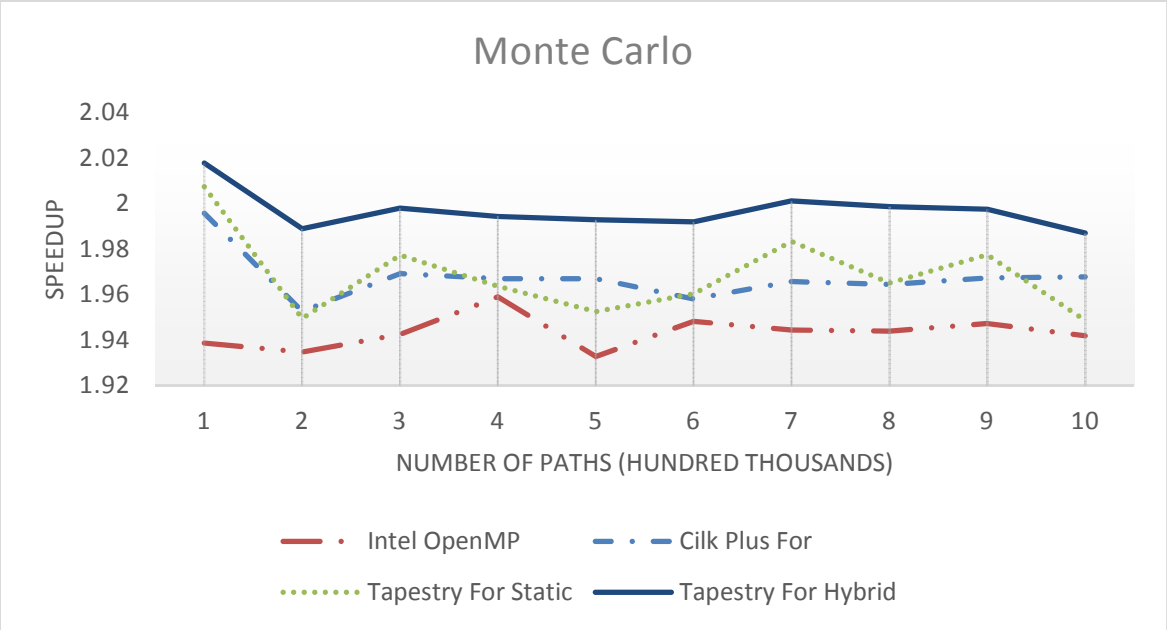


Figure A.9: Monte Carlo Fork/Join

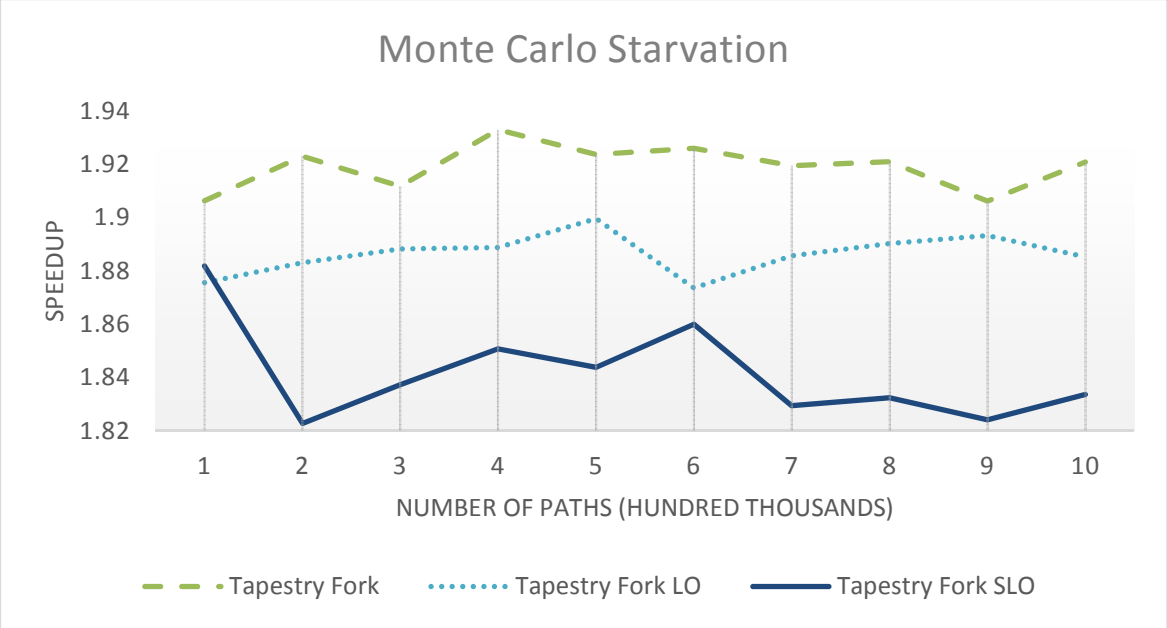


Figure A.10: SLO Starvation Monte Carlo

A.1.2.6 Monte Carlo Starvation

As referenced in Figure A.10, Tapestry LO and SLO performs well in this task taking 2-5% hit over Tapestry for these embarrassingly parallel large tasks despite the potential for starvation introduced by the optimizations. Tapestry dependencies also perform well. For this task we see a 5-8% slowdown over our for loop versions.

In the following sections, we compare dependencies in runtime threads to the slowest runtime (OpenMP). We note that runtime threads are much faster than operating system threads. In all these algorithms we are compare fork/join to a dependency divide a conquer approach that uses split phase transactions or a special aggregate threads. We can apply three static optimizations to dependencies: LO, Memory, and Dynamic Split Phases. SLO is not applicable because it is implemented by replacing thread glue with recursive calls which is not possible when dependencies need to be met.

A.1.2.7 Fibonacci Dependencies

As seen in Figure A.11, the two most important optimization here are the Dynamic Split Phase which reduces the amount of work significantly and LO which reduces scheduler overhead. If memory management is not used, the optimizations have little affect on the performance alone. However, they increase the performance by 2x. We are producing about two times the amount of threads as the thread only version which speaks volumes on the performance.

A.1.2.8 N-Queens Dependencies

Figure A.12 shows the performance is still very good for a more coarse grain task like N-Queens with a board size of 16. We see a similar pattern to which optimizations are the best compared with the Fibonacci. In this benchmark, we are still producing about double the amount of work due to the Dynamic Split Phase transaction compared with the thread only version.

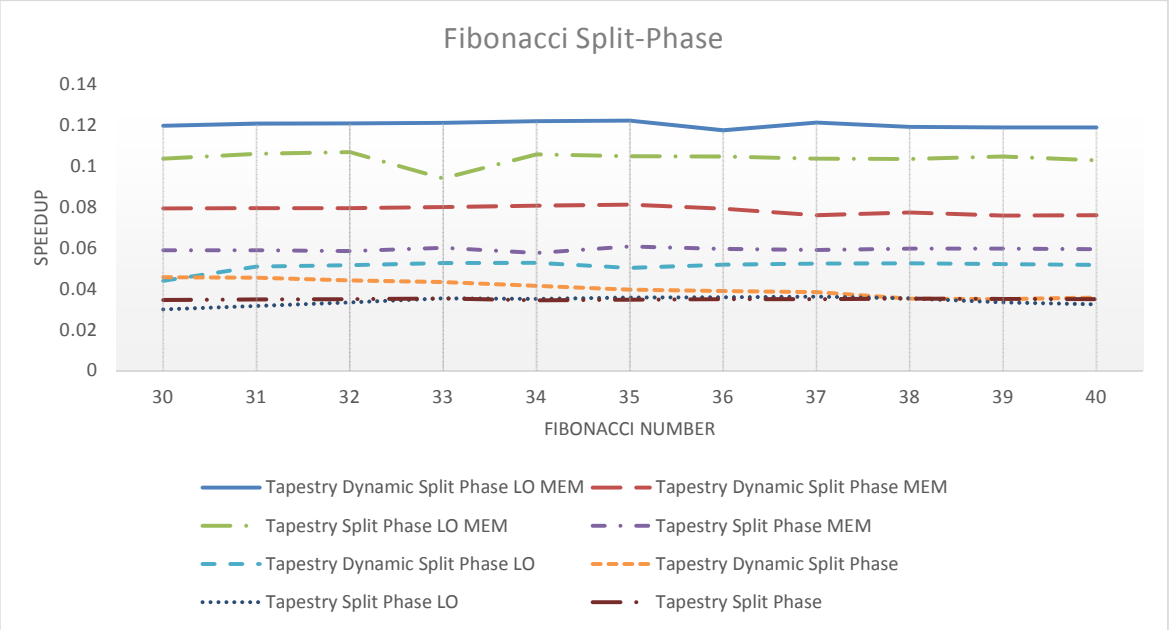


Figure A.11: Fibonacci Dependencies

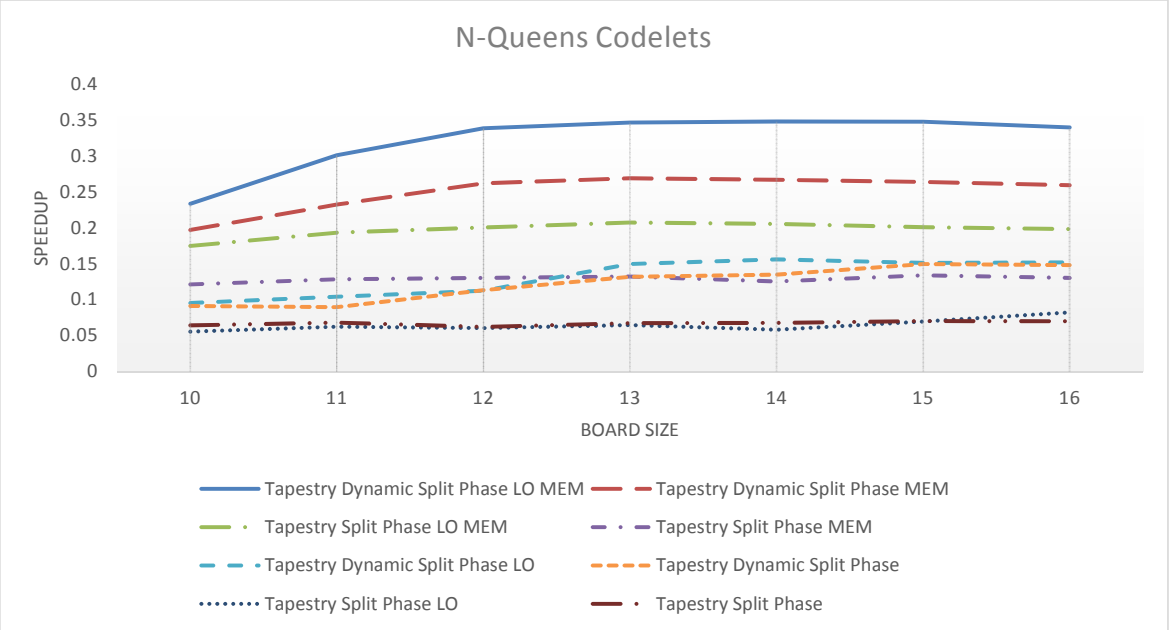


Figure A.12: N-Queens Dependencies

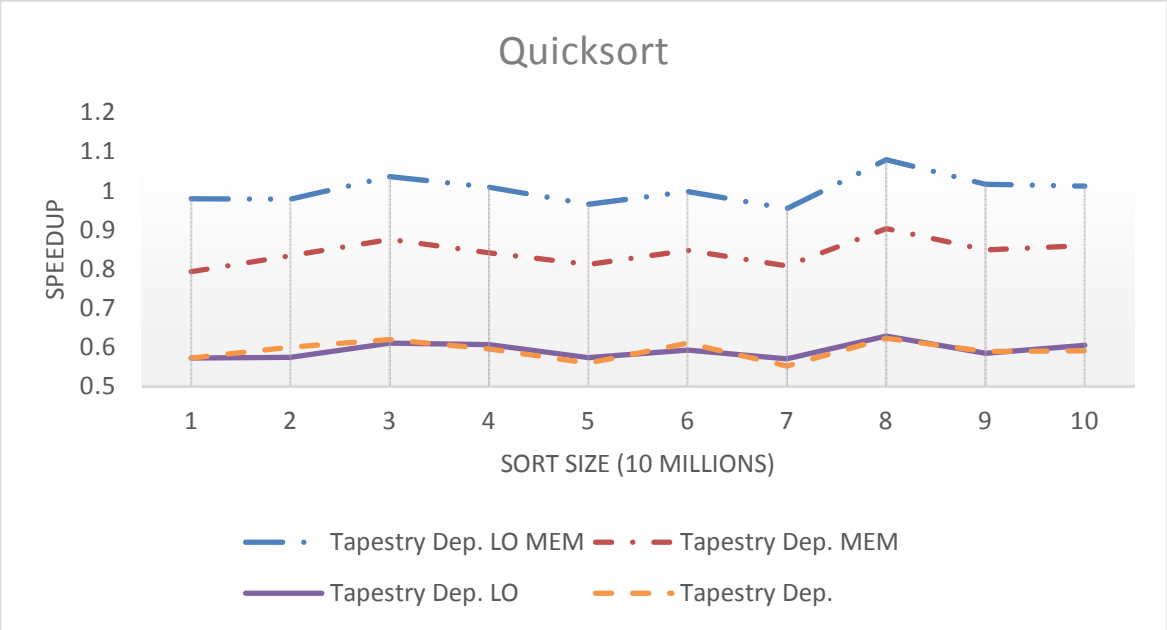


Figure A.13: Quicksort Scalability for Dependencies

A.1.2.9 Quicksort Dependencies

The dependency quicksort uses an aggregate thread to indicate the work is finished. It produces the same amount of work as the threaded version, but with less memory requirements because the threads do not need to stick around unless we use the stack based memory management with hints. The performance shown in Figure A.13 is on par with the sequential version and OpenMP. It is within 10% the performance of Cilk Plus and 70% of Tapestry Threads only.

A.1.2.10 Monte Carlo Dependencies

In the Monte Carlo test we compare an embarrassingly parallel task using dependencies. The Tapestry dependencies are created in a loop and use an aggregate thread to indicate the work is finished. The Static Dependencies For version creates a number of macro threads equal to the number of cores with the for work evenly split between cores. The Hybrid Dependencies For version creates more dependencies with less work allowing for some to be stolen. A completely dynamic version would

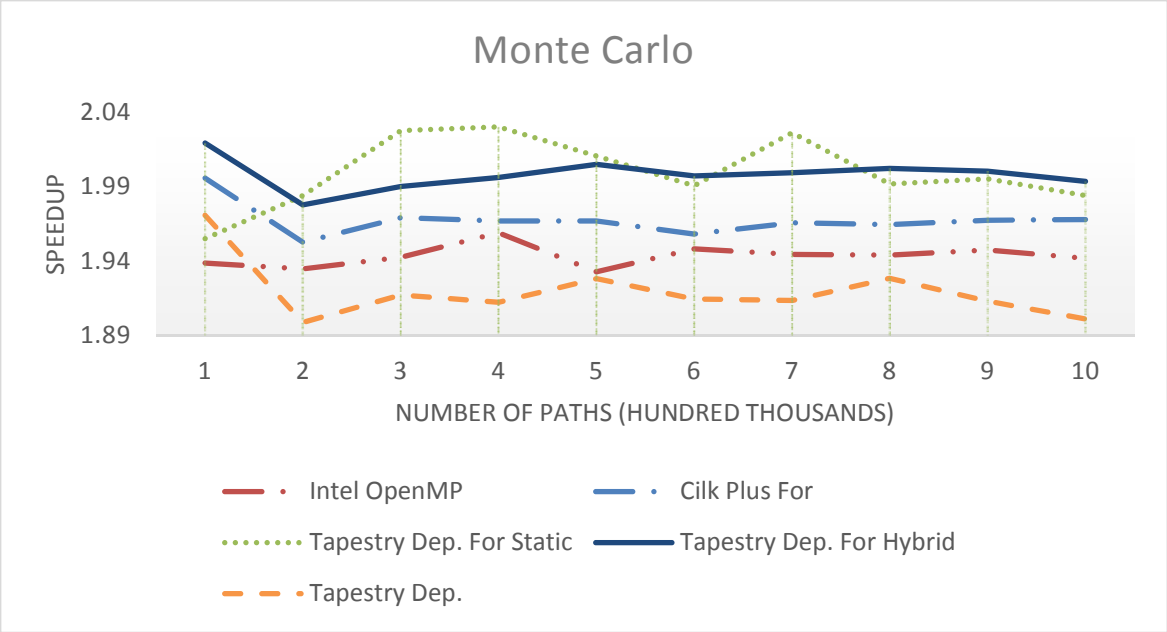


Figure A.14: Monte Carlo Dependencies

equivalent to the Tapestry dependency version. The results are good as seen in Figure A.14.

A.1.3 OS Benchmarks

These tests are configured to use Tapestry as an over all layer to operating system threads. Tapestry's layer provides new features to the strict C style threads including the use of dependencies. We evaluate the performance of using fork/join synchronization in comparison of using dependencies. These unique tests are designed to show the advantage of using dependencies in a context switching environment for a large number of threads. The baseline is fork/join for all the operating system tests. I run these tests in Windows 8 64-bit using an Intel Core 2 Duo at 2.4 GHz with 6GB of ram. I use the Parallel Studio XE 2011 compiler for Windows on these tests with \O2 optimizations.

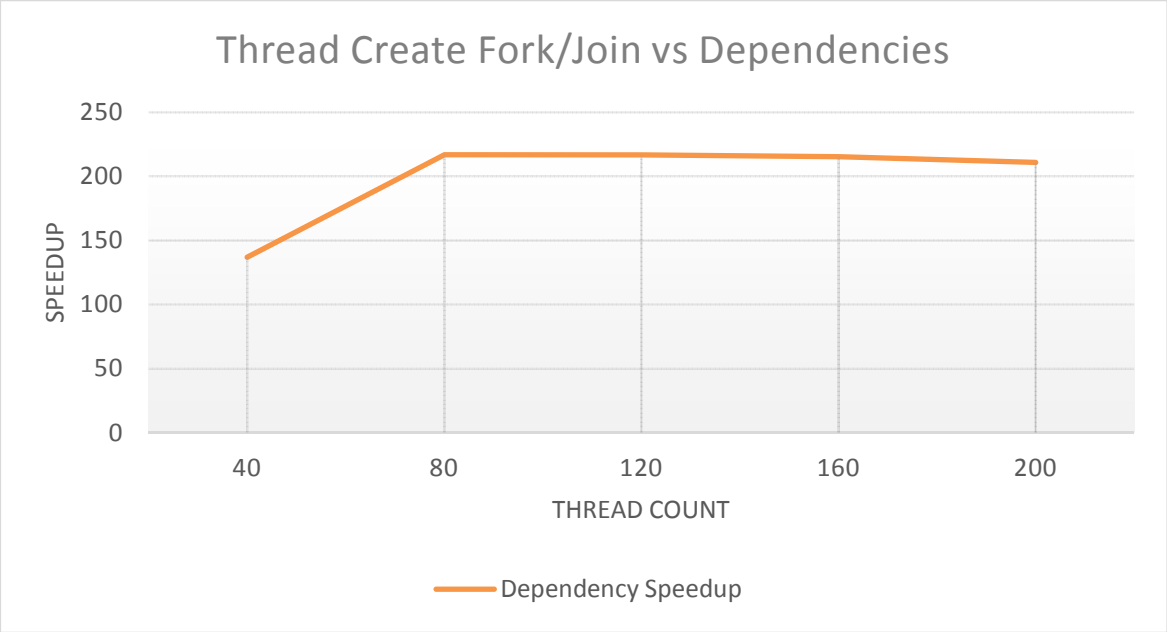


Figure A.15: Spawn Test: The baseline is fork/join.

A.1.3.1 Spawn Test

This test spawns threads from a loop. Each thread computes a random number, then spawns an additional worker thread that updates the value. The random number threads wait for its results back from the worker, updates it, and print the result. For the dependency version of the test, we create three threads: the random number thread, the update thread, and the print thread. A print thread depends on a update thread which depends on a random number thread. For this dependency synchronization pattern the dependency version produces 33% more threads, but guarantees it will have at most only half the number of threads alive at once. This reduces the memory requirements in a half. The results for this test are seen in Figure A.15.

A.1.3.2 Fibonacci

This test performs a recursive Fibonacci with each thread being a separate recursive call. So for fib = N, there will be two threads spawned with fib=N-1 and N-2. However, we optimize here and call one thread using recursion without spawning.

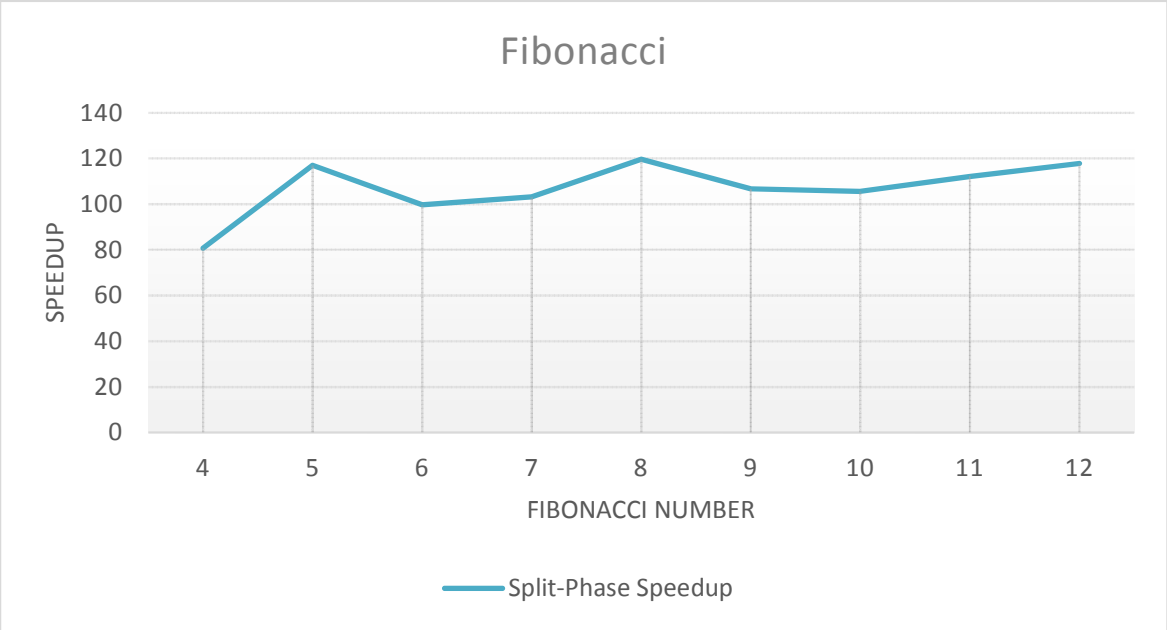


Figure A.16: Fibonacci: The baseline is fork/join.

For the dependency version, the addition and spawning are split into two separate threads. The addition thread is dependent on the two spawned threads. The addition thread can be thought of as a continuation of the spawning thread. Because of this, the thread signaling information needs to be copied over via the dynamic split-phase transaction. Additionally, the dynamic split-phase transaction allows for one recursive call to be done without spawning. Due to this fact, the dependency version is spawning twice the number of threads, but at most requires one half the memory. This test is very fine grain, with barely any operations performed. The results for this test are seen in Figure A.16.

A.1.3.3 N-Queens

The N-Queens benchmark is similar to Fibonacci. It is a recursive back off algorithm. We only do half the board to take care of rotations and reflections. For each position we mark off where a queen is and has already been placed and recursively place more queens until we find a solution or not. When a solution is found: a 1 or 0

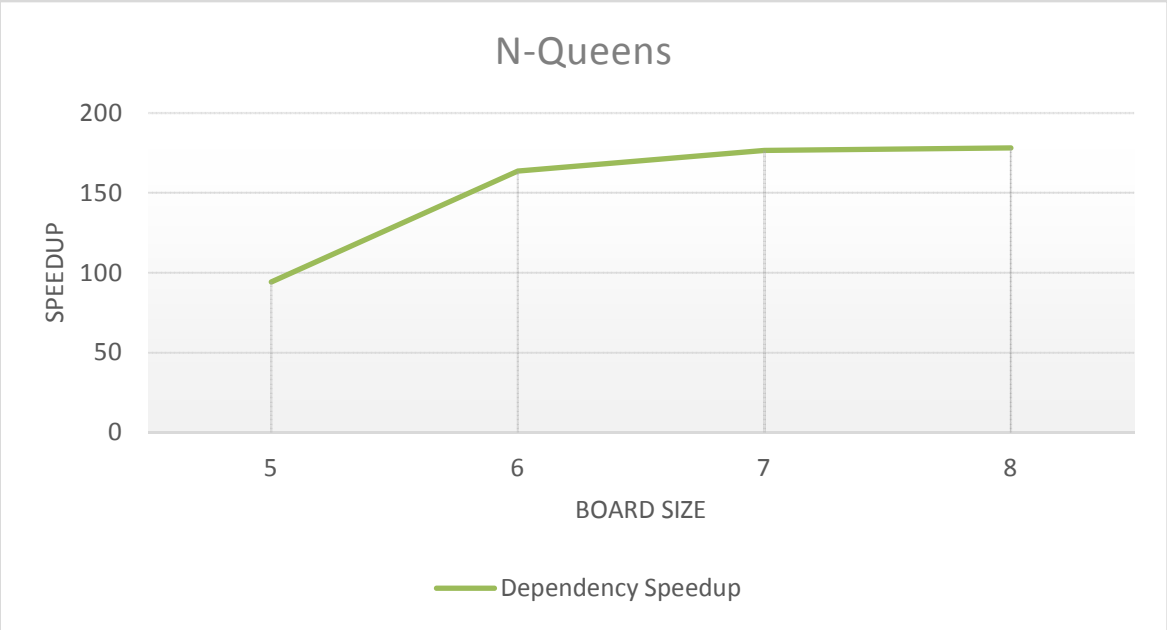


Figure A.17: N-Queens: The baseline is fork/join.

is returned, and those values are added up similar to the Fibonacci application. The difference here is that N-Queens can spawn a variable and larger number of threads for each thread which reduces in size as you move toward the leaf nodes. Furthermore, the test is less balanced meaning certain threads can produce more work than the other threads. Also the implementation uses bit-fields and is far more coarse grain than Fibonacci due to the thread spawning work and board updating. The results for this test are seen in Figure A.17.

A.1.3.4 Quicksort

The last test done is a quicksort recursively. Each block of the sort is done by a thread. So initially the work is very coarse, but as the algorithm proceeds toward the leaf nodes, it becomes much finer. Each thread waits for their child to finish using a join. The dependency version uses an aggregate thread which all threads signal to indicate their work is finished; thus, they do not need to wait. The results for this test are seen in Figure A.18.

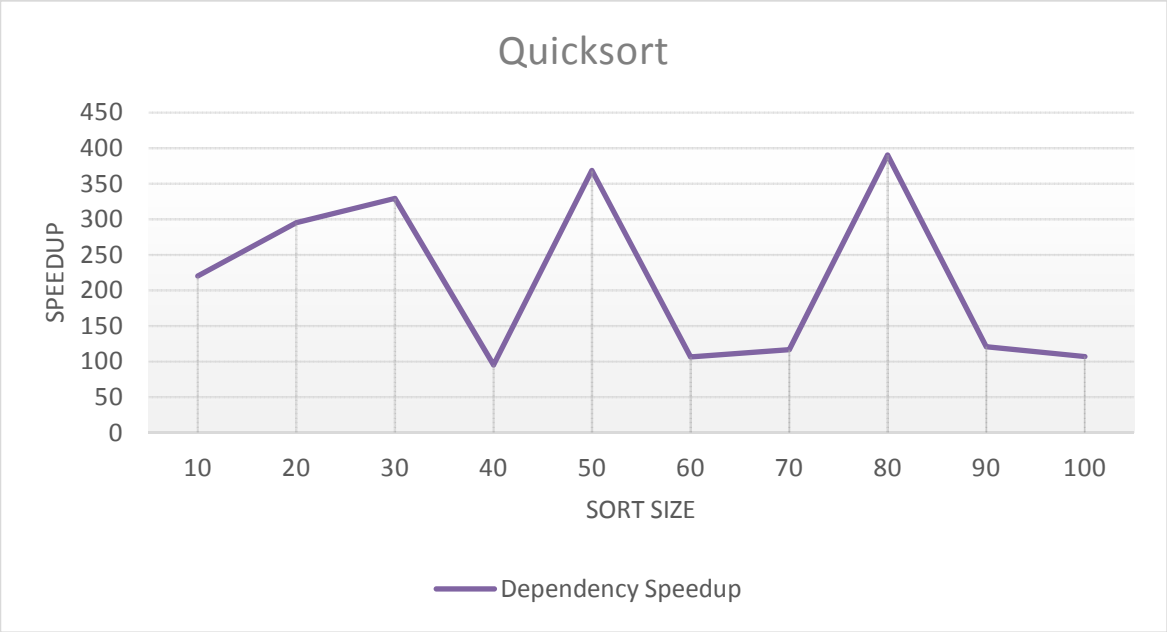


Figure A.18: Quicksort: The baseline is fork/join.

All the results show a superior advantage for using dependencies vs the fork/join model for finer-grain parallelism in preemptive operating system environment with speedups in the range of 80 to 400 times with much less memory requirements. Analysis indicates, the advantage comes from the fact that the fork/join model keeps parent threads alive while children work thus increasing the overall number of threads alive at any given one time which increases the amount of time spent context switching between threads. However, in the dependency mode parent threads will exit after completing all useful work leaving the operating system to context switch only on threads doing useful work. In general, the fork/join models are much easier to code because one doesn't need to split threads to handle end states.

A.1.4 Tree Reduction Tests

These tests show the benefit of reducing a tree based search into a graph one automatically by the Tapestry runtime.

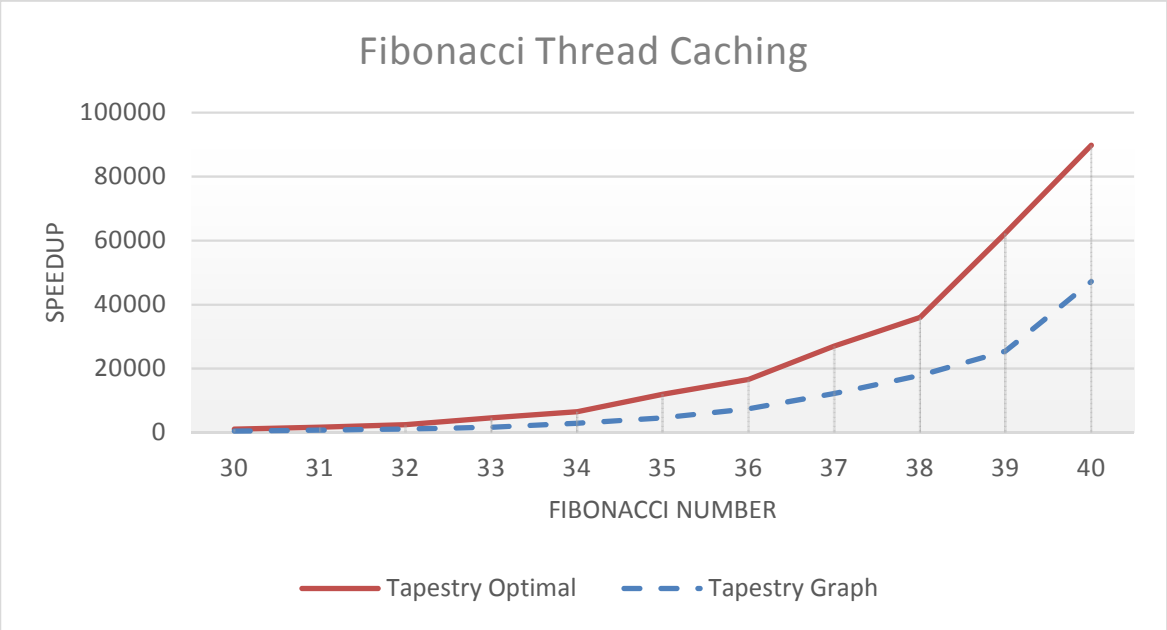


Figure A.19: Fibonacci Automatic Tree to Graph Reduction Speedup

A.1.5 Fibonacci Automatic Tree to Graph Reduction Speedup

Using Tapestry's redundant graph elimination optimization so that calls to the same thread with same data are only called once and connected to other states i.e. redundant states are eliminated, we see significant reduction in Fibonacci's workload over the sequential and thus a significant improvement in speed as shown in Figure A.19. Optimal is speedup achieved by the user employing manual reduction of states by creating a manual graph vs the Tapestry runtime doing this for them.

A.2 Case Study for Core i7

These were benchmarks run on the Core i7 using hyper-threading. We only run application specific tests on these machines.

A.2.1 Runtime Benchmarks

The next scalability tests use a Core i7 using up to 8 threads on the 4 core machine because the machine has hyper threading support.

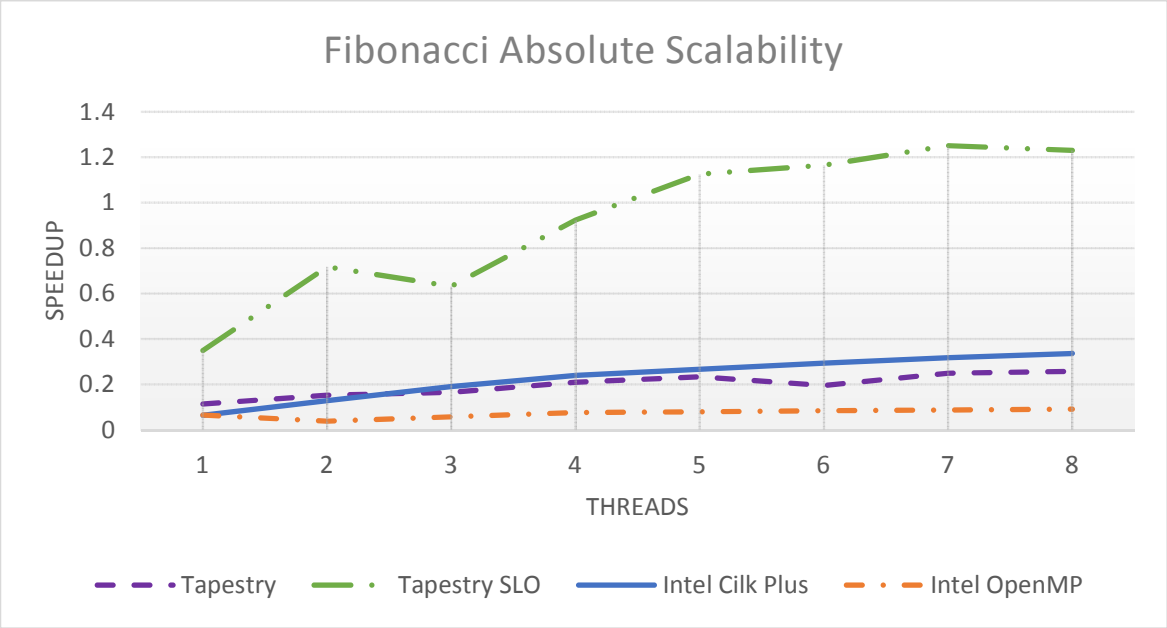


Figure A.20: Fibonacci Scalability: A Fibonacci number of 40 was calculated.

A.2.1.1 Fibonacci Scalability

Here we see that Tapestry's performance is not much different from Cilk Plus until we add in the Super Locality Optimization (Figure A.20). We are 5 to 6 times faster than Cilk Plus with this locality aware optimizations. This indicates very regular fine-grain tasks benefit much from these optimizations.

A.2.1.2 N-Queens Scalability

This application is more coarse-grain and less regular. As seen in Figure A.21, for this application we see that Tapestry still does well. In particular, it scales well with or without optimizations and is on par with Cilk Plus. The speedup relative to the recursive version is very good with both Tapestry and Cilk Plus almost meeting the sequential performance. However with optimizations turned on, Tapestry out performs the sequential at 2 cores and achieves a 2.5x speed up over the sequential when all cores are utilized.

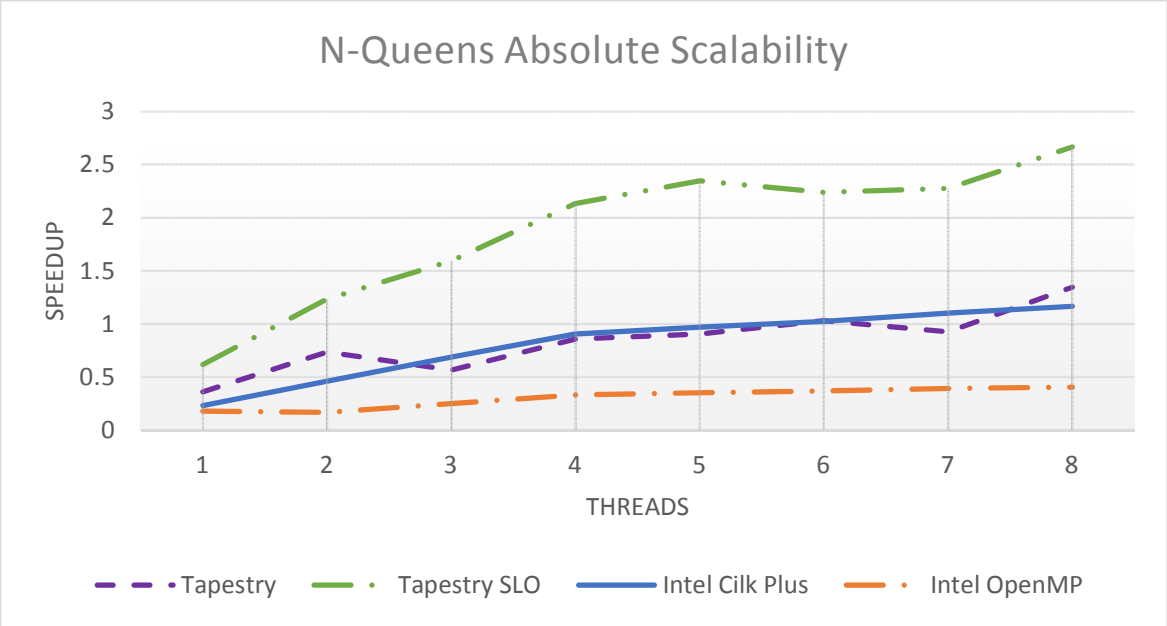


Figure A.21: N-Queens Scalability: A board size of 16 was used.

A.2.1.3 Quicksort Scalability

Here the relative scalability is lower for SLO which can be referenced in seen in Figure A.22 based on the initial performance on one node. Tapestry hits a speedup of 4.03x on this 4 core machine. I note the work begins coarse and in low quantity and becomes much more available over time but more fine. The coarse work makes up the bulk of the work performed which affects performance by limiting the overall speedup gain within the bounds of the first couple of iterations. In addition, the overhead as tasks become finer also attributes to slowdown.

A.2.1.4 N-Puzzle Scalability

The benchmark as seen in Figure A.23 shows the performance is not very regular. However, Tapestry SLO still outperforms in all cases.

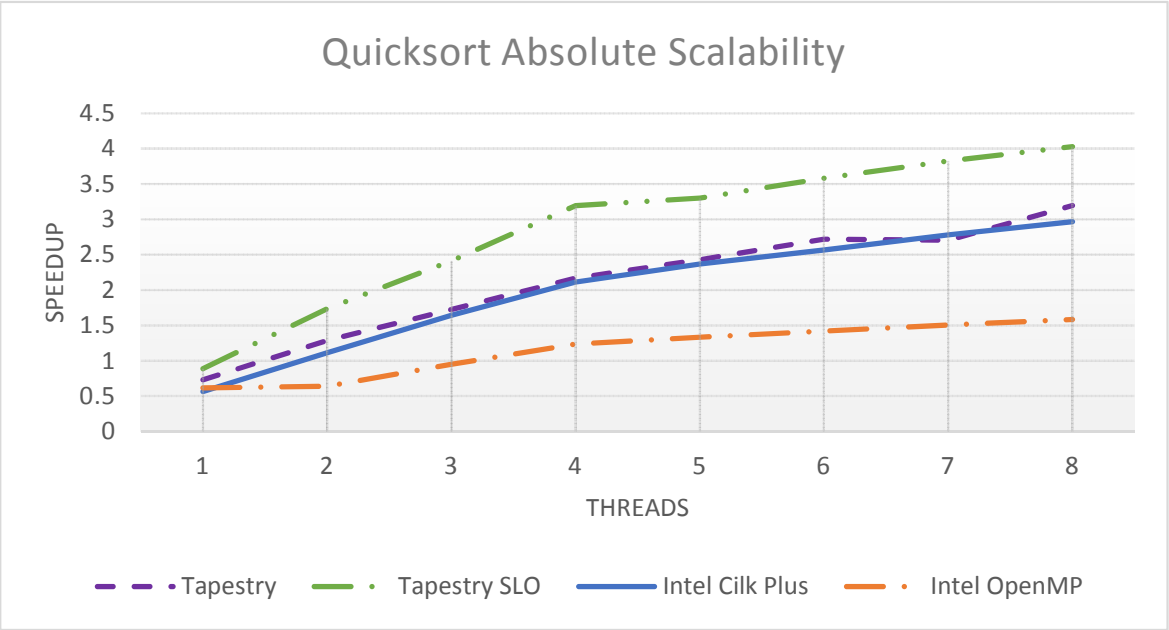


Figure A.22: Quicksort Scalability: 55 million integer numbers were sorted.

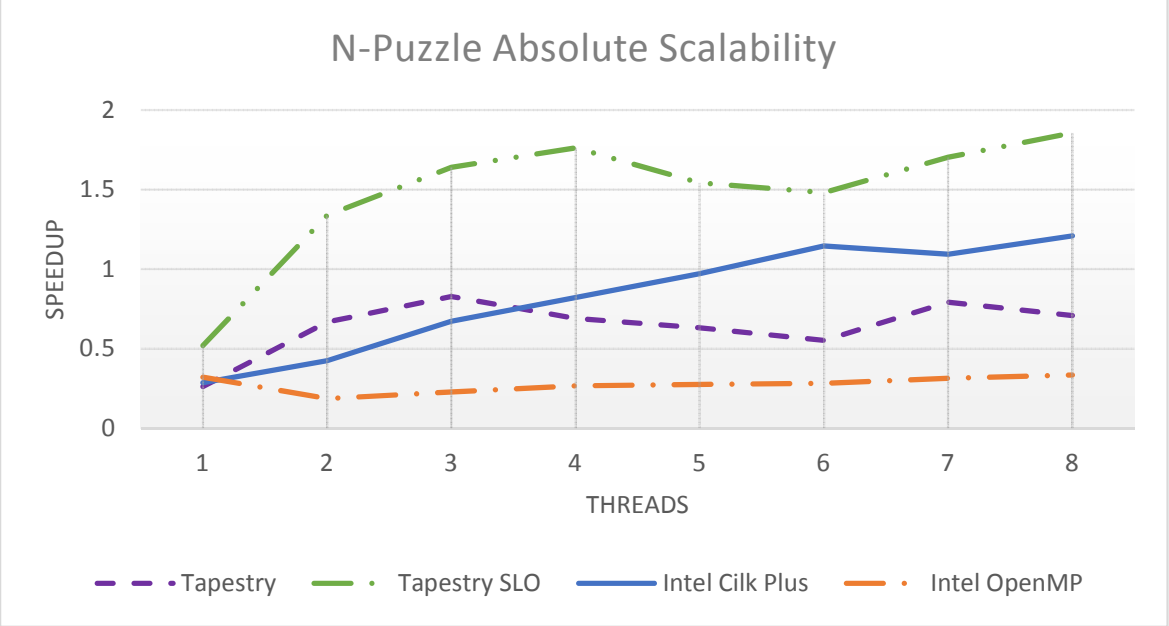


Figure A.23: N-Puzzle Scalability: Solution was at a depth of 18 for a 3x3 puzzle.

Appendix B

BENCHMARK CODE

B.1 Fork/Join Benchmarks

B.1.1 Thread Spawn

```
1  int threadSpawnB( int v )
2  {
3    return v+1;
4  }
5
6  int threadSpawnA( int v )
7  {
8    int result = _spawn_ threadSpawnB( v );
9
10   _sync_;
11
12   return result;
13 }
14
15 void threadSpawnTest( void )
16 {
17   for( int i = 0; i < COUNT; i++ )
18   {
19     _spawn_ threadSpawnA( 1 );
20   }
21
22   _sync_;
23 }
```

B.1.2 Fibonacci

```
int
fib( int in )
{
  if( fib < 2 )
    return fib;
  else
  {
    int x = _spawn_ fib( in - 2 )
    int y = fib ( in - 1 );

    _sync_;

    return  x + y;

  }
}
```

B.1.3 N-Queens

```
int nqueens(unsigned int RowsToBeFilled, unsigned int ColsToBeFilled,
      unsigned int LeftDiag, unsigned int RightDiag, unsigned int row)
{
  unsigned int r, c ;
  int found = 0;
  int i = 0;

  int results [BOARD-1];

  unsigned int copyRowsToBeFilled;
  unsigned int copyColsToBeFilled1;
  unsigned int copyLeftDiag1;
  unsigned int copyRightDiag1;
  unsigned int copyRowsToBeFilled1;

  copyRowsToBeFilled = RowsToBeFilled;
  if ( copyRowsToBeFilled!= 0 )
```

```
18    {
19      r = (-((signed)RowsToBeFilled) & RowsToBeFilled);
20      RowsToBeFilled &= ~( r );
21      if( row < MAX-1 )
22      {
23        copyColsToBeFilled1 = ColsToBeFilled | r;
24        copyLeftDiag1 = (LeftDiag | r) >> 1;
25        copyRightDiag1 = (RightDiag | r) << 1;
26        copyRowsToBeFilled1 = MASK & ~( copyColsToBeFilled1 | copyLeftDiag1
27                                        | copyRightDiag1 );
28      }
29    }
30
31    while( RowsToBeFilled != 0 )
32    {
33      r = (-((signed)RowsToBeFilled) & RowsToBeFilled);
34      RowsToBeFilled &= ~( r );
35
36      if ( row < MAX-1)
37      {
38
39        unsigned int copyColsToBeFilled = ColsToBeFilled | r;
40        unsigned int copyLeftDiag = (LeftDiag | r) >> 1;
41        unsigned int copyRightDiag = (RightDiag | r) << 1;
42        unsigned int copyRowsToBeFilled = MASK & ~( copyColsToBeFilled |
43                                          copyLeftDiag | copyRightDiag );
44
45        results[i] = _spawn_ nqueens( copyRowsToBeFilled,
46                    copyColsToBeFilled, copyLeftDiag, copyRightDiag, row+1);
47
48        i++;
49      }
50      else
51      {
52        found += 1;
```

```
53      }
54    }
55    int size = i;
56    if( copyRowsToBeFilled != 0)
57    {
58      if ( row < MAX-1)
59      {
60          found += nqueens( copyRowsToBeFilled1, copyColsToBeFilled1,
61                            copyLeftDiag1, copyRightDiag1, row+1);
62      }
63      else
64      {
65          found += 1;
66      }
67    }
68
69    _sync_
70
71    for(i=size-1; i >= 0; i--)
72    {
73      found += results[i];
74    }
75
76    return found;
77  }
78
79  void nqueensTop( )
80  {
81    unsigned int RowsToBeFilled, ColsToBeFilled, LeftDiag, RightDiag, rows;
82
83    unsigned int solutions, solutionsOdd = 0;
84
85    rows = ColsToBeFilled = LeftDiag = RightDiag = 0;
86
87    int half = BOARD>>1;
```

```
88    RowsToBeFilled = (1 << half) - 1;
89
90    timer.start();
91
92    solutions = _spawn_ 2*nqueens (RowsToBeFilled, ColsToBeFilled,
93                                   LeftDiag, RightDiag, rows) ;
94
95    if( MAX & 1) //half of middle column for odd
96    {
97          RowsToBeFilled = 1 << (MAX >> 1);
98          rows = 1;
99
100         ColsToBeFilled = RowsToBeFilled;
101         LeftDiag = (RowsToBeFilled >> 1);
102         RightDiag = (RowsToBeFilled << 1);
103         RowsToBeFilled = (RowsToBeFilled - 1) >> 1;
104
105         solutionsOdd = 2*nqueens_cilk (RowsToBeFilled, ColsToBeFilled,
106                                        LeftDiag, RightDiag, rows) ;
107   }
108
109   _sync_;
110
111   solutions+=solutionsOdd;
112 }
```

B.1.4 Quicksort

```
1 void qSort(int * begin, int * end)
2 {
3     if (begin != end) {
4         --end;  // Exclude last element (pivot) from partition
5         int * middle = std::partition(begin, end,
6                             std::bind2nd(std::less<int>(), *end));
7         using std::swap;
8         swap(*end, *middle);     // move pivot to middle
```

```
9           _spawn_ qSort(begin, middle);
10          qSort(++middle, ++end); // Exclude pivot and restore end
11          _sync_;
12      }
13  }
```

B.1.5 Monte-Carlo

```
1  static const int nopt=30;
2  static const  int maturities[] = { 4, 4, 4, 8, 8, 8, 20, 20, 20, 28, 28,
3                                     28, 40, 40, 40, 48, 48, 48, 60, 60, 60,
4                                     68, 68, 68, 80, 80, 80, 88, 88, 88 };
5  static const int nmat=5700;
6
7  static const int n=nmat+1;
8
9  static const double delta = 0.25; /* LIBOR interval  */
10 static const double swaprates[]  = { .045, .05, .055, .045, .05, .055,
11                                      .045, .05, .055, .045, .05, .055,
12                                      .045, .05, .055, .045, .05, .055,
13                                      .045, .05, .055, .045, .05, .045,
14                                      .05, .055, .045, .05, .055, .045,
15                                      .05};
16
17 void scalarKernel(double * L0, double * z, double * lambda, double * v)
18 {
19     double b, s, swapval;
20     double sqez, lam, con1, v_scal, vrat;
21     int i, j;
22     double B[nmat], S[nmat], L[n];
23
24     for(i=0;i<n;i++) {
25         L[i] = L0[i];
26     }
27
28     for(j=0; j<nmat; j++)
```

```
29      {
30          sqez = sqrt(delta)*z[j];
31          v_scal = 0.0;
32
33          for (i=j+1; i<n; i++) {
34              lam  = lambda[i-j-1];
35              con1 = delta*lam;
36              v_scal   += con1*L[i]/(1.0+delta*L[i]);
37              vrat = exp(con1*v_scal + lam*(sqez-0.5*con1));
38              L[i] = L[i]*vrat;
39          }
40      }
41
42      b = 1.0;
43      s = 0.0;
44
45      for (j=nmat; j<n; j++) {
46          b = b/(1.0+delta*L[j]);
47          s = s + delta*b;
48          B[j-nmat] = b;
49          S[j-nmat] = s;
50      }
51
52      v_scal = 0.0;
53
54      for (i=0; i<nopt; i++){
55          int k = maturities[i] -1;
56          swapval = B[k] + swaprates[i]*S[k] - 1.0;
57          if (swapval < 0.0)
58              v_scal += -100.0*swapval;
59      }
60
61      // apply discount
62
63      for (j=0; j<nmat; j++){
```

```
64          v_scal = v_scal/(0.0+delta*L[j]);
65      }
66
67      v[0]=v_scal;
68  }
69
70  void monteSpawn(double * z, double * v, double * L0, double * lambda,
71                  int npath)
72  {
73      int    path;
74
75      for (path=0; path<npath; path++)
76    {
77          _spawn_ scalarKernel(L0, &z[path*nmat], lambda, &v[path]);
78      }
79    _sync_;
80
81  }
```

B.1.6 N-Puzzle

```
1  class nPuzzle
2  {
3    int empty;
4    char board[SIZE*SIZE];
5    unsigned int row;
6    unsigned int col;
7    bool up, down, left, right;
8
9  public:
10   nPuzzle( int newEmpty, char newBoard[SIZE*SIZE] ):
11   empty( newEmpty ),
12   row  ( newEmpty / SIZE ),
13   col  ( newEmpty - row * SIZE ),
14   up   ( true ),
15   down ( true ),
```

```
16    left ( true ),
17    right( true )
18    {
19      memcpy(board, newBoard, sizeof(char)*SIZE*SIZE);
20    }
21
22    nPuzzle(int newEmpty, int swap, char newBoard[SIZE*SIZE]):
23    empty( swap ),
24    row  ( swap / SIZE),
25    col  ( swap - row * SIZE),
26    up   ( true ),
27    down ( true ),
28    left ( true ),
29    right( true )
30    {
31      memcpy( board, newBoard, sizeof(char)*SIZE*SIZE );
32
33      std::swap( board[newEmpty], board[swap] );
34    }
35
36    bool solution()
37    {
38      return strncmp( board,goal, SIZE*SIZE ) == 0;
39    }
40
41    nPuzzle slideLeft()
42    {
43      return nPuzzle( empty, empty-1, board );
44    }
45
46    nPuzzle slideRight()
47    {
48      return nPuzzle( empty, empty+1, board );
49    }
50
```

```
51    nPuzzle slideUp()
52    {
53      return nPuzzle( empty, empty - SIZE, board) ;
54    }
55
56    nPuzzle slideDown()
57    {
58      return nPuzzle( empty, empty + SIZE, board );
59    }
60
61    bool canSlideUp()    { return row > 0 && up; }
62    bool canSlideDown()  { return row < ( SIZE-1 ) && down; }
63    bool canSlideRight() { return col < ( SIZE-1 ) && right; }
64    bool canSlideLeft()  { return col > 0 && left; }
65
66
67  };
68
69  bool search( nPuzzle puzzle, unsigned int depth )
70  {
71    if( cancel )
72      return false;
73    if( puzzle.solution() )
74    {
75      cancel = true;
76      return true;
77    }
78
79    bool up = false, down = false, left = false, right=  false;
80
81    if(depth != limit)
82    {
83      depth++;
84
85      if( puzzle.canSlideUp() )
```

```
86        up = _spawn_ search(puzzle.slideUp(),depth);
87      if( puzzle.canSlideDown() )
88        down = _spawn_ search(puzzle.slideDown(),depth);
89      if( puzzle.canSlideLeft() )
90      {
91
92        if( puzzle.canSlideRight() )
93        {
94          left =  _spawn_ search( puzzle.slideLeft(), depth );
95          right = _spawn_ search( puzzle.slideRight(), depth );
96        }
97        else
98          left = search( puzzle.slideLeft(), depth );
99      }
100     else
101       right = search( puzzle.slideRight(), depth );
102
103     _sync_;
104
105     return up || down || left || right;
106   }
107
108   return false;
109 }
110
111 void nPuzzleTop( )
112 {
113
114   int i = iters;
115   char testStart[] = "310628457";
116
117   nPuzzle testPuzzle( 2, testStart );
118
119   double result=0;
120
```

```
121     for(i; i>0; i--)
122     {
123       limit = 0;
124       cancel = false;
125       timer.start();
126       while( !search( testPuzzle, 0 ) )
127         limit++;
128     }
129
130 }
```

B.1.7 Matrix Multiplication

```
1  //
2  //Title: Pthread Matrix Multiplication Block
3  //
4  //File: pthread_matmul_grid.c
5  //
6  //Description:
7  //This program performs matrix multiplication on a randomly generated
8  //matrix.The size of the matrix is passed in via command line. We
9  //create a grid based on BLOCK size and divide up each of the blocks
10 //among threads as evenly as possible so at most any thread will
11 //have to compute 1 extra block. Block size should be chosen to
12 //maxamize cache performance and works well at 256x256. We do a
13 //whole slew of optimizations to improve performance.We do register
14 //tile using loop unrolling at 2x4 block increments, which also should
15 //be vectorized by the compiler. We do a zig-zag access pattern to
16 //improve locality and reduce cache conflicts among threads. We do
17 //minimim address computations when multiplying. We set each thread to
18 //only be used on 1 processor to improve locality.
19 //
20 //Joshua Landwehr 4/09/2010 (snapcore@gmail.com)
21
22 //-------------------------------------------------------------------
23 //Group: Includes ---------------------------------------------------
```

```
24 //---------------------------------------------------------------------
25
26 #define _GNU_SOURCE
27
28 /************************
29   Include: "stdio.h"
30
31   Description:
32     Standard C I/O operations. We use the printf, fprintf, and
33     perror functions.
34 ************************/
35 #include <stdio.h>
36
37 /************************
38   Include: "stdlib.h"
39
40   Description:
41     Standard C library. We use the exit function and the
42     EXIT_FAILURE constant.
43 ************************/
44 #include <stdlib.h>
45
46 /************************
47   Include: "limits.h"
48
49   Description:
50     Characteristics of common variables. We use the LONG_MAX and
51     LONG_MIN constants.
52 ************************/
53 #include <limits.h>
54
55 /************************
56   Include: "errno.h"
57
58   Description:
```

```
59      Used for reporting errors. We use errno global variable and
60      RANGE constant.
61  **********************/
62  #include <errno.h>
63
64  /**********************
65    Include: "time.h"
66
67    Description:
68      Standard C time and data. We use the time, gettimeofday function.
69      We also use the timeval struct.
70  **********************/
71  #include <time.h>
72
73  /**********************
74    Include: "pthread.h"
75
76    Description:
77      Includes functions for thread creation. We use pthread_create
78      and pthread_join functions. We also use the pthread_t struct.
79  **********************/
80  #include <pthread.h>
81
82  #include <sched.h>
83
84  #include <math.h>
85
86  //Constants: PRINT_SIZE
87  //The biggest size of NxN matrix we will print
88  #define PRINT_SIZE 5
89
90  /*
91    Constants: Random Bounds
92
93   RANMAX - Defines the maximum bound of our random numbers.
```

```
94    RANMIN - Defines the minimium bound of our random numbers.
95  */
96  #define RANMAX 1.0
97  #define RANMIN -1.0
98
99  #define BLOCK 256
100
101 #define ERROR_TOL 0.00001
102
103 #define matmul matmul_block
104
105 double* matrix_a;
106 double* matrix_b;
107 double* matrix_a1;
108 double* matrix_b1;
109 double* matrix_c;
110 int requested_matrix_size;
111 int num_blocks_p;
112
113 //--------------------------------------------------------------------
114 //Group: Implementation ----------------------------------------------
115 //--------------------------------------------------------------------
116
117 /***********************
118   Function: init_matrix_block
119
120   Description:
121     Fills a matrix with random numbers between RANMAX and RANMIN.
122
123   Parameters:
124     requested_matrix_size - Total size of matrix
125     matrix - location of matrix to init.
126
127 ***********************/
128 init_matrix_block(int requested_matrix_size, double* matrix)
```

```
129 {
130   int i,j;
131   double range = RAN_MAX - RAN_MIN;
132   for(i = 0; i < requested_matrix_size; i++)
133   {
134     matrix[i] = rand() * range / RAND_MAX + RAN_MIN;
135   }
136 }
137
138 /**********************
139   Function: matmul_simple
140
141   Description:
142     Performs vanilla matrix multiplication on a contigous memory
143     location. Used for verification.
144
145   Parameters:
146     requested_matrix_size - Size of matrix in 1 dimension
147     matrix_a - location of matrix A.
148     matrix_b - location of matrix B.
149     matrix_c - location of matrix C.
150
151 **********************/
152 void matmul_simple(int requested_matrix_size, double* matrix_a,
153                    double* matrix_b, double* matrix_c)
154 {
155   int i,j,k;
156   for(i = 0; i < requested_matrix_size; i++)
157   {
158     register double *a_temp = matrix_a + i * requested_matrix_size;
159     for(j = 0; j < requested_matrix_size; j++)
160     {
161       register double *b_temp = matrix_b + j * requested_matrix_size;
162       register double c_ij = *(matrix_c + j + i * requested_matrix_size);
163       for(k = 0; k < requested_matrix_size; k++)
```

```
164         {
165           c_ij += *(a_temp + k) * *(b_temp+k);
166         }
167         *(matrix_c + j + i * requested_matrix_size) = c_ij;
168       }
169     }
170 }
171
172 /************************
173   Function: perform_block_matmul_fin
174
175   Description:
176     Performs matrix multiplication on remaining elements of blocks
177     that could not be fit in our register tile size.
178
179   Parameters:
180     requested_matrix_size - Size of matrix in 1 dimension
181     z - number of C elements across to calculate
182     x - number of C elements down to calculate
183     c - number of elements from A, B to use for C element.
184     matrix_a - location of our starting fringe in A.
185     matrix_b - location of our starting fringe in B.
186     matrix_c - location of our starting fringe in c.
187
188 ************************/
189 void perform_block_matmul_fin(int requested_matrix_size, int z, int x,
190                               int c, double* matrix_a, double* matrix_b,
191                               double* matrix_c)
192 {
193   int i, j, k;
194
195   for(i = 0; i<z; i++)
196   {
197     register double *a_temp = matrix_a + i * BLOCK;
198     for(j = 0; j<x; j++)
```

```
199     {
200       register double *b_temp = matrix_b + j * BLOCK;
201       register double c_ij = *(matrix_c + j + i * BLOCK);
202       for(k = 0; k<c; k++)
203       {
204           c_ij += *(a_temp + k) * *(b_temp+k);
205       }
206       *(matrix_c + j + i * BLOCK) = c_ij;
207     }
208
209   }
210 }
211
212 /**********************8
213   Function: perform_block_matmul_unroll_8
214
215   Description:
216     Performs matrix multiplication on blocks using a 2x4 register
217     tile.
218
219   Parameters:
220     requested_matrix_size - Size of matrix in 1 dimension
221     matrix_a - location of matrix A.
222     matrix_b - location of matrix B.
223     matrix_c - location of matrix C.
224     i - starting i element for this block
225     j - starting j element for this block
226     k - starting k element for this block
227     bl_i - starting i block for this bock
228     bl_j - starting j block for this bock
229     bl_k - starting k block for this bock
230     num_blocks_p - blocks per row
231
232 **********************/
233 void perform_block_matmul_unroll_8(int requested_matrix_size,
```

```
234                                     double* matrix_a, double* matrix_b,
235                                     double* matrix_c, int i, int j, int k,
236                                     int bl_i, int bl_j, int bl_k,
237                                     int num_blocks_p)
238 {
239   //z - number of C elements across to calculate
240   //x - number of C elements down to calculate
241   //c - number of elements from A, B to use for C element.
242   int z, x, c;
243
244   //If any of these are true our block is bigger than elements to use
245   //in that dimension: i,j,k
246   if(i+BLOCK > requested_matrix_size)
247     z = requested_matrix_size-i;
248   else
249     z = BLOCK;
250
251   if(j+BLOCK > requested_matrix_size)
252     x = requested_matrix_size-j;
253   else
254     x = BLOCK;
255
256   if(k+BLOCK > requested_matrix_size)
257     c = requested_matrix_size-k;
258   else
259     c = BLOCK;
260
261   //Offset a,b,c into the correct blocks
262   matrix_a += bl_k * BLOCK*BLOCK + bl_i * BLOCK*BLOCK*num_blocks_p;
263   matrix_b += bl_k * BLOCK*BLOCK + bl_j * BLOCK*BLOCK*num_blocks_p;
264   matrix_c += bl_j * BLOCK*BLOCK + bl_i * BLOCK*BLOCK*num_blocks_p;
265
266   //Loop movement
267   int u_i=i;
268   int u_j=j;
```

```
269
270   //Stride of our tiles
271   int j_stride = 2;
272   int i_stride = 4;
273
274   //Number of elements in i and j dimensions
275   int num_i = z/i_stride;
276   int num_j = x/j_stride;
277
278   //Move one C down
279   for(u_i = 0; u_i<num_i; u_i++)
280   {
281     i=u_i*i_stride;
282     register double *a_temp = matrix_a + i * BLOCK;
283     //Move one C across
284     for(u_j = 0; u_j<num_j; u_j++)
285     {
286       //Find real j
287       j=u_j*j_stride;
288       register double *b_temp = matrix_b + j * BLOCK;
289
290       //Compute 2 Across and 2 Down thus 2x2 = 4
291       register double c_ij1 = *(matrix_c + (j + 0) + (i + 0) * BLOCK);
292       register double c_ij2 = *(matrix_c + (j + 0) + (i + 1) * BLOCK);
293       register double c_ij3 = *(matrix_c + (j + 0) + (i + 2) * BLOCK);
294       register double c_ij4 = *(matrix_c + (j + 0) + (i + 3) * BLOCK);
295       register double c_ij5 = *(matrix_c + (j + 1) + (i + 0) * BLOCK);
296       register double c_ij6 = *(matrix_c + (j + 1) + (i + 1) * BLOCK);
297       register double c_ij7 = *(matrix_c + (j + 1) + (i + 2) * BLOCK);
298       register double c_ij8 = *(matrix_c + (j + 1) + (i + 3) * BLOCK);
299
300       //Accumulate tiles
301       for(k = 0; k<c; k++)
302       {
303         c_ij1 +=*(a_temp + k + (0) * BLOCK)* *(b_temp + k + (0) * BLOCK);
```

```
304           c_ij2 +=*(a_temp + k + (1) * BLOCK)* *(b_temp + k + (0) * BLOCK);
305           c_ij3 +=*(a_temp + k + (2) * BLOCK)* *(b_temp + k + (0) * BLOCK);
306           c_ij4 +=*(a_temp + k + (3) * BLOCK)* *(b_temp + k + (0) * BLOCK);
307           c_ij5 +=*(a_temp + k + (0) * BLOCK)* *(b_temp + k + (1) * BLOCK);
308           c_ij6 +=*(a_temp + k + (1) * BLOCK)* *(b_temp + k + (1) * BLOCK);
309           c_ij7 +=*(a_temp + k + (2) * BLOCK)* *(b_temp + k + (1) * BLOCK);
310           c_ij8 +=*(a_temp + k + (3) * BLOCK)* *(b_temp + k + (1) * BLOCK);
311       }
312
313       //Save results
314       *(matrix_c + (j + 0) + (i + 0) * BLOCK) = c_ij1;
315       *(matrix_c + (j + 0) + (i + 1) * BLOCK) = c_ij2;
316       *(matrix_c + (j + 0) + (i + 2) * BLOCK) = c_ij3;
317       *(matrix_c + (j + 0) + (i + 3) * BLOCK) = c_ij4;
318       *(matrix_c + (j + 1) + (i + 0) * BLOCK) = c_ij5;
319       *(matrix_c + (j + 1) + (i + 1) * BLOCK) = c_ij6;
320       *(matrix_c + (j + 1) + (i + 2) * BLOCK) = c_ij7;
321       *(matrix_c + (j + 1) + (i + 3) * BLOCK) = c_ij8;
322     }
323
324   }
325   //We could not compute a elements as they did not fit in our stride
326   if(z%i_stride != 0 || x%j_stride !=0 )
327   {
328       //Finish missing elements wide (j) downward fringes
329       perform_block_matmul_fin(requested_matrix_size, z ,
330                                x - num_j * j_stride, c,
331                                matrix_a,
332                                matrix_b+num_j*j_stride*BLOCK,
333                                matrix_c+num_j*j_stride);
334
335       //Finish missing elements down (i) rightward fringes
336       perform_block_matmul_fin(requested_matrix_size,
337                                z - num_i * i_stride,
338                                num_j*j_stride, c,
```

```
339                                 matrix_a+num_i*i_stride*BLOCK,
340                                 matrix_b,
341                                 matrix_c+num_i*i_stride*BLOCK);
342   }
343 }
344
345 /**********************
346   Function: matmul_block
347
348   Description:
349     Divides matrices into blocks and allocates work based on the
350     start_id. Then it iterates through the blocks calling a lower
351     function to perform the actual matrix multiplication per block.
352
353   Parameters:
354     requested_matrix_size - Size of matrix in 1 dimension
355     matrix_a - location of matrix A.
356     matrix_b - location of matrix B.
357     matrix_c - location of matrix C.
358     start_id - id of thread to perform this matrix multiplication.
359
360 **********************/
361 void matmul_block(int block_i)
362 {
363   //Local loop block versions are used to indicate i and j for that block
364   int i, j, k, block_j, block_k;
365
366   //bl_i is the col and bl_j is the row
367   int bl_i = block_i/(num_blocks_p);
368   int bl_j = block_i%(num_blocks_p);
369
370   //Element in real full matrix
371   i = bl_i*BLOCK;
372   j = bl_j*BLOCK;
373
```

```
374   for(block_k = 0; block_k < num_blocks_p; block_k++)
375   {
376     //Element in real full matrix
377     k = block_k*BLOCK;
378     perform_block_matmul_unroll_8(requested_matrix_size, matrix_a,
379                                   matrix_b, matrix_c, i, j, k,
380                                   bl_i, bl_j, block_k,num_blocks_p);
381   }
382 }
383
384 //Converts normal contigous matrix X passed into block version Y
385 //passed out
386 int map_standard_to_block(int row_size, int x)
387 {
388   int num_blocks = (row_size)/(BLOCK);
389   if((row_size)%(BLOCK))
390     num_blocks++;
391   int y;
392   int row = (x)/row_size;
393
394   int col = (x)%row_size;
395
396   int block_row = (row)%BLOCK;
397
398   int block_col = (col)%BLOCK;
399
400   y = row/BLOCK*num_blocks+col/BLOCK;
401
402   y = y*BLOCK*BLOCK+block_col+block_row*BLOCK;
403
404   return y;
405 }
406
407 //Just do malloc and calloc for us
408 void * malloc_blocks(int num_blocks)
```

```
409 {
410   return malloc(sizeof(double) * num_blocks);
411 }
412
413 //Just do malloc and calloc for us
414 void * calloc_blocks(int num_blocks)
415 {
416
417   return calloc(num_blocks,sizeof(double));
418 }
419
420
421 /**********************
422   Function: main
423
424   Description:
425     Performs parallel matrix multiplication where data is randomly
426     generated. Argv[1] should contain the NxN size of the matrix.
427
428   Parameters:
429     argc - number of arguments passed to program.
430     argv - an array of the arguments as char* (c strings).
431
432   Return:
433     0 - on success.
434     not 0 - on failure.
435 **********************/
436 int main(int argc, char* argv[])
437 {
438
439   struct timeval start_time;
440   struct timeval int_stop_time;
441   struct timeval com_stop_time;
442   float int_time;
443   float com_time;
```

```
444   float tot_time;
445   float gflops, ptime, rtime;
446   long long flops;
447   //Used for error checking in strtol.
448   char* input_end;
449   pthread_t *threads_desc;
450   //Row and column size
451
452
453   //AxB=C
454   double* matrix_d;
455   unsigned int matrix_size;
456
457   //Local loop vars
458   int i,j,k;
459
460   if(argc == 2)
461   {
462     int retval;
463
464     /* Initialize the library */
465     requested_matrix_size = strtol(argv[1], &input_end, 10);
466
467     //Error checking.
468     if ((errno == ERANGE && (requested_matrix_size == LONG_MAX ||
469                              requested_matrix_size == LONG_MIN)) ||
470               errno != 0 && requested_matrix_size == 0)
471     {
472       perror("strtol");
473       exit(EXIT_FAILURE);
474     }
475
476     if (input_end == argv[1])
477     {
478       fprintf(stderr, "No digits were found\n");
```

```
479         exit(EXIT_FAILURE);
480       }
481
482       if (*input_end != '\0')
483       {
484         fprintf
485         (
486           stderr,
487           "Further characters after number: %s\n",
488           input_end
489         );
490         exit(EXIT_FAILURE);
491       }
492
493       if(requested_matrix_size < 1)
494       {
495         fprintf(stderr, "Matrix Size < 1\n");
496         exit(EXIT_FAILURE);
497       }
498
499       matrix_size = sizeof(double) * requested_matrix_size *
500                                      requested_matrix_size;
501
502       int num_blocks = (requested_matrix_size)/(BLOCK);
503
504
505       num_blocks_p = (requested_matrix_size)/(BLOCK);
506
507       if(requested_matrix_size%BLOCK)
508       {
509         num_blocks++;
510         num_blocks_p++;
511       }
512       int total_blocks = num_blocks*num_blocks;
513
```

```
514     matrix_a = malloc_blocks(num_blocks*num_blocks*BLOCK*BLOCK);
515     matrix_b = malloc_blocks(num_blocks*num_blocks*BLOCK*BLOCK);
516     matrix_c = calloc_blocks(num_blocks*num_blocks*BLOCK*BLOCK);
517     threads_desc = malloc(T*sizeof(pthread_t));
518
519     srand ( time(NULL) );
520
521     //Measure Initialiaztion Time
522     gettimeofday(&start_time, NULL);
523
524     init_matrix_block(num_blocks*num_blocks*BLOCK*BLOCK, matrix_a);
525     init_matrix_block(num_blocks*num_blocks*BLOCK*BLOCK, matrix_b);
526
527     gettimeofday(&int_stop_time, NULL);
528
529
530     //Total Blocks
531
532     for(i = 0; i < total_blocks; i++)
533       _spawn_ matmul_block(i);
534
535     _sync_;
536
537     gettimeofday(&com_stop_time, NULL);
538
539     int_time = (int_stop_time.tv_sec +
540       (float)int_stop_time.tv_usec/1000000.0) - (start_time.tv_sec +
541                                 (float)start_time.tv_usec/1000000.0);
542
543     com_time = (com_stop_time.tv_sec +
544                               (float)com_stop_time.tv_usec/1000000.0) -
545      (int_stop_time.tv_sec + (float)int_stop_time.tv_usec/1000000.0);
546
547     tot_time = int_time + com_time;
548
```

```
549     gflops = ((2*pow(requested_matrix_size,3)-
550               pow(requested_matrix_size,2))/(com_time))/pow(10,9);
551     printf
552     (
553       "Intialization Time: %.3f\n" \
554       "Computation Time: %.3f\n" \
555       "Total Time: %.3f\n" \
556       "GFLOPs: %.3f\n",
557       int_time,
558       com_time,
559       tot_time,
560       gflops
561     );
562
563   return 0;
564 }
```

B.2 Dependency Benchmarks

B.2.1 Thread Spawn

```
1  int threadSpawnB( int v )
2  {
3    return v+1;
4  }
5
6  int threadSpawnA( int v )
7  {
8
9    return v;
10 }
11
12 void threadSpawnTest( void )
13 {
14   for( int i = 0; i < COUNT; i++ )
15   {
```

```
16      int result = _spawn_ threadSpawnB( 1 );
17      _spawn_ threadSpawnA( result );
18    }
19
20    _sync_; //Wait for threads to finish
21  }
```

B.2.2 Fibonacci

```
1  int fibAdd ( int x, int y )
2  {
3      return x + y;
4  }
5
6  int fib ( int n )
7  {
8      if( n < 2 )
9          return n;
10     else
11     {
12          int x = _spawn_ fib( n - 1);
13          int y = _spawn_ fib( n - 2);
14
15         _continue_  fibAdd ( x, y);
16
17         return 0;
18     }
19 }
```

B.2.3 N-Queens

```
1  int NumberOfSetBits(int i)
2  {
3      i = i - ((i >> 1) & 0x55555555);
4      i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
5      return (((i + (i >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
```

```
6 }
7
8 unsigned int aggregate_nqueens(unsigned int value)
9 {
10   return value;
11 }
12
13 int nqueens(unsigned int RowsToBeFilled, unsigned int ColsToBeFilled,
14       unsigned int LeftDiag, unsigned int RightDiag, unsigned int row)
15 {
16   unsigned int r, c ;
17   int found = 0;
18   int i = 0;
19
20   int results [BOARD-1];
21
22   unsigned int copyRowsToBeFilled;
23
24   int nCodelets = NumberOfSetBits(RowsToBeFilled);
25   ThreadAggregate aggregate
26   if(nCodelets-1 > 0)
27   {
28     //Create a codelet that aggregates nCodelets values
29     aggregate = ThreadAggregate (&aggregate_nqueens, nCodelets);
30     _continue_ ( aggregate );
31     aggregate.dependsOn(this);
32   }
33
34   while( RowsToBeFilled != 0 )
35   {
36     r = (-((signed)RowsToBeFilled) & RowsToBeFilled);
37     RowsToBeFilled &= ~( r );
38
39     if ( row < MAX-1)
40     {
```

```
41
42        unsigned int copyColsToBeFilled = ColsToBeFilled | r;
43        unsigned int copyLeftDiag = (LeftDiag | r) >> 1;
44        unsigned int copyRightDiag = (RightDiag | r) << 1;
45        unsigned int copyRowsToBeFilled = MASK & ~( copyColsToBeFilled |
46                                                    copyLeftDiag | copyRightDiag );
47
48        results[i] = _spawn_ nqueens( copyRowsToBeFilled,
49                    copyColsToBeFilled, copyLeftDiag, copyRightDiag, row+1);
50
51        aggregate.dependsOn(results[i]);
52
53        i++;
54      }
55      else
56      {
57        found += 1;
58      }
59    }
60
61    return found;
62  }
63
64  void nqueensTop( )
65  {
66    unsigned int RowsToBeFilled, ColsToBeFilled, LeftDiag, RightDiag, rows;
67
68    unsigned int solutions, solutionsOdd = 0;
69
70    rows = ColsToBeFilled = LeftDiag = RightDiag = 0;
71
72    int half = BOARD>>1;
73    RowsToBeFilled = (1 << half) - 1;
74
75    ThreadAggregate threadEnd(&aggregate_nqueens, 2);
```

```
76
77    if(MAX & 1)
78      threadEnd.setSignalSize(2);
79    else
80      threadEnd.setSignalSize(1);
81
82    solutions = _spawn_ 2*nqueens (RowsToBeFilled, ColsToBeFilled,
83                                   LeftDiag, RightDiag, rows) ;
84
85    threadEnd.dependsOn( solutions );
86    if( MAX & 1) //half of middle column for odd
87    {
88          RowsToBeFilled = 1 << (MAX >> 1);
89          rows = 1;
90
91          ColsToBeFilled = RowsToBeFilled;
92          LeftDiag = (RowsToBeFilled >> 1);
93          RightDiag = (RowsToBeFilled << 1);
94          RowsToBeFilled = (RowsToBeFilled - 1) >> 1;
95
96          solutionsOdd = 2*nqueens (RowsToBeFilled, ColsToBeFilled,
97                                    LeftDiag, RightDiag, rows) ;
98
99      threadEnd.dependsOn( solutionsOdd );
100   }
101
102 }
```

B.2.4 Quicksort

```
1 void qSort(int * begin, int * end, Thread * threadEnd)
2 {
3     if (begin != end) {
4         --end;  // Exclude last element (pivot) from partition
5         int * middle = std::partition(begin, end,
6                         std::bind2nd(std::less<int>(), *end));
```

```
7           using std::swap;
8           swap(*end, *middle);     // move pivot to middle
9
10      threadEnd->dependsOnSignalOnly( _spawn_ qSort(begin, middle) );
11
12          qSort(++middle, ++end); // Exclude pivot and restore end
13      }
14  }
```

B.2.5 Monte-Carlo

```
1  static const int nopt=30;
2  static const  int maturities[] = { 4, 4, 4, 8, 8, 8, 20, 20, 20, 28, 28,
3                                     28, 40, 40, 40, 48, 48, 48, 60, 60, 60,
4                                     68, 68, 68, 80, 80, 80, 88, 88, 88 };
5  static const int nmat=5700;
6
7  static const int n=nmat+1;
8
9  static const double delta = 0.25; /* LIBOR interval  */
10 static const double swaprates[]  = { .045, .05, .055, .045, .05, .055,
11                                      .045, .05, .055, .045, .05, .055,
12                                      .045, .05, .055, .045, .05, .055,
13                                      .045, .05, .055, .045, .05, .045,
14                                      .05, .055, .045, .05, .055, .045,
15                                      .05};
16
17 void scalarKernel(double * L0, double * z, double * lambda, double * v)
18 {
19      double b, s, swapval;
20      double sqez, lam, con1, v_scal, vrat;
21      int i, j;
22      double B[nmat], S[nmat], L[n];
23
24      for(i=0;i<n;i++) {
25          L[i] = L0[i];
```

```
26      }
27
28      for(j=0; j<nmat; j++)
29      {
30          sqez = sqrt(delta)*z[j];
31          v_scal = 0.0;
32
33          for (i=j+1; i<n; i++) {
34              lam  = lambda[i-j-1];
35              con1 = delta*lam;
36              v_scal   += con1*L[i]/(1.0+delta*L[i]);
37              vrat = exp(con1*v_scal + lam*(sqez-0.5*con1));
38              L[i] = L[i]*vrat;
39          }
40      }
41
42      b = 1.0;
43      s = 0.0;
44
45      for (j=nmat; j<n; j++) {
46          b = b/(1.0+delta*L[j]);
47          s = s + delta*b;
48          B[j-nmat] = b;
49          S[j-nmat] = s;
50      }
51
52      v_scal = 0.0;
53
54      for (i=0; i<nopt; i++){
55          int k = maturities[i] -1;
56          swapval = B[k] + swaprates[i]*S[k] - 1.0;
57          if (swapval < 0.0)
58              v_scal += -100.0*swapval;
59      }
60
```

```
61      // apply discount
62
63      for (j=0; j<nmat; j++){
64          v_scal = v_scal/(0.0+delta*L[j]);
65      }
66
67      v[0]=v_scal;
68  }
69
70  void endFunc()
71  {
72    //print results
73  }
74
75  void monteSpawn(double * z, double * v, double * L0, double * lambda,
76                  int npath)
77  {
78      int path;
79    Thread end(&endFunc);
80
81    end.setSignalSize(path);
82
83
84      for (path=0; path<npath; path++)
85    {
86      end.dependsOnSignalOnly( _spawn_ scalarKernel(L0,
87                                  &z[path*nmat], lambda, &v[path]));
88      }
89    _sync_;
90
91  }
```

B.2.6 Fibonacci Dynamic

```
1  int fibAdd ( int x, int y )
2  {
```

```
3       return x + y;
4   }
5
6   int fib ( int n )
7   {
8       if( n < 2 )
9           return n;
10      else
11      {
12           int x = _spawn_ fib( n - 2);
13
14          _continue_  fibAdd ( x, this);
15
16      return fib( n - 1);
17      }
18  }
```

B.2.7 N-Queens Dynamic

```
1   int NumberOfSetBits(int i)
2   {
3       i = i - ((i >> 1) & 0x55555555);
4       i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
5       return (((i + (i >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
6   }
7
8   unsigned int aggregate_nqueens(unsigned int value)
9   {
10    return value;
11  }
12
13  int nqueens(unsigned int RowsToBeFilled, unsigned int ColsToBeFilled,
14          unsigned int LeftDiag, unsigned int RightDiag, unsigned int row)
15  {
16    unsigned int r, c ;
17    int found = 0;
```

```
18  int i = 0;
19
20  int results [BOARD-1];
21
22  unsigned int copyRowsToBeFilled;
23  unsigned int copyColsToBeFilled1;
24  unsigned int copyLeftDiag1;
25  unsigned int copyRightDiag1;
26  unsigned int copyRowsToBeFilled1;
27
28  int nCodelets = NumberOfSetBits(RowsToBeFilled);
29  ThreadAggregate aggregate
30  if(nCodelets-1 > 0)
31  {
32    //Create a codelet that aggregates nCodelets values
33    aggregate = ThreadAggregate (&aggregate_nqueens, nCodelets);
34    _continue_ ( aggregate );
35    aggregate.dependsOn(this);
36  }
37
38  copyRowsToBeFilled = RowsToBeFilled;
39  if ( copyRowsToBeFilled!= 0 )
40  {
41    r = (-((signed)RowsToBeFilled) & RowsToBeFilled);
42    RowsToBeFilled &= ~( r );
43    if( row < MAX-1 )
44    {
45      copyColsToBeFilled1 = ColsToBeFilled | r;
46      copyLeftDiag1 = (LeftDiag | r) >> 1;
47      copyRightDiag1 = (RightDiag | r) << 1;
48      copyRowsToBeFilled1 = MASK & ~( copyColsToBeFilled1 | copyLeftDiag1
49                                      | copyRightDiag1 );
50    }
51  }
52
```

```
53   while( RowsToBeFilled != 0 )
54   {
55     r = (-((signed)RowsToBeFilled) & RowsToBeFilled);
56     RowsToBeFilled &= ~( r );
57
58     if ( row < MAX-1)
59     {
60
61       unsigned int copyColsToBeFilled = ColsToBeFilled | r;
62       unsigned int copyLeftDiag = (LeftDiag | r) >> 1;
63       unsigned int copyRightDiag = (RightDiag | r) << 1;
64       unsigned int copyRowsToBeFilled = MASK & ~( copyColsToBeFilled |
65                                           copyLeftDiag | copyRightDiag );
66
67       results[i] = _spawn_ nqueens( copyRowsToBeFilled,
68                   copyColsToBeFilled, copyLeftDiag, copyRightDiag, row+1);
69
70       aggregate.dependsOn(results[i]);
71
72       i++;
73     }
74     else
75     {
76       found += 1;
77     }
78   }
79   int size = i;
80   if( copyRowsToBeFilled != 0)
81   {
82     if ( row < MAX-1)
83     {
84         found += nqueens( copyRowsToBeFilled1, copyColsToBeFilled1,
85                           copyLeftDiag1, copyRightDiag1, row+1);
86     }
87     else
```

```
88      {
89          found += 1;
90      }
91    }
92
93    return found;
94  }
95
96  void nqueensTop( )
97  {
98    unsigned int RowsToBeFilled, ColsToBeFilled, LeftDiag, RightDiag, rows;
99
100   unsigned int solutions, solutionsOdd = 0;
101
102   rows = ColsToBeFilled = LeftDiag = RightDiag = 0;
103
104   int half = BOARD>>1;
105   RowsToBeFilled = (1 << half) - 1;
106
107   ThreadAggregate threadEnd(&aggregate_nqueens, 2);
108
109   if(MAX & 1)
110     threadEnd.setSignalSize(2);
111   else
112     threadEnd.setSignalSize(1);
113
114   solutions = _spawn_ 2*nqueens (RowsToBeFilled, ColsToBeFilled,
115                                  LeftDiag, RightDiag, rows) ;
116
117   threadEnd.dependsOn( solutions );
118   if( MAX & 1) //half of middle column for odd
119   {
120         RowsToBeFilled = 1 << (MAX >> 1);
121         rows = 1;
122
```

```
123           ColsToBeFilled = RowsToBeFilled;
124           LeftDiag = (RowsToBeFilled >> 1);
125           RightDiag = (RowsToBeFilled << 1);
126           RowsToBeFilled = (RowsToBeFilled - 1) >> 1;
127
128           solutionsOdd = 2*nqueens (RowsToBeFilled, ColsToBeFilled,
129                                                 LeftDiag, RightDiag, rows) ;
130
131       threadEnd.dependsOn( solutionsOdd );
132     }
133
134     _sync_;
135
136     solutions+=solutionsOdd;
137 }
```