LOCALITY AWARENESS
FOR TASK PARALLEL COMPUTATION

Stephen Lecler Olivier

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2012

Approved by

Jan F. Prins, Advisor

James H. Anderson

Bronis R. de Supinski

Robert J. Fowler

Allan K. Porterfield

©2012
Stephen Lecler Olivier
ALL RIGHTS RESERVED

ABSTRACT

STEPHEN LECLER OLIVIER: Locality Awareness for Task Parallel Computation
(Under the direction of Jan F. Prins)

The task parallel programming model allows programmers to express concurrency at a high level of abstraction and places the burden of scheduling parallel execution on the run time system. Efficient scheduling of tasks on multi-socket multicore shared memory systems requires careful consideration of an increasingly complex memory hierarchy, including shared caches and non-uniform memory access (NUMA) characteristics. In this dissertation, we study the performance impact of these issues and other performance factors that limit parallel speedup in task parallel program executions and propose new scheduling strategies to improve performance. Our performance model characterizes lost efficiency in terms of overhead time, idle time, and *work time inflation* due to increased data access costs.

We introduce a hierarchical run time scheduler that combines the benefits of work stealing and parallel depth-first schedulers. Matching the scheduler design to the memory hierarchy of multicore NUMA systems limits costly remote data accesses while maintaining load balance and exploiting constructive data sharing among threads that share a cache. We also propose a locality-based scheduling framework based on *locality domains* and comprising an API for programmers to specify application locality and a scheduler that honors those specifications. Implementations of the hierarchical and locality-based schedulers in our OpenMP run time system exhibit performance improvements on several task parallel benchmark applications over existing scheduling strategies and production OpenMP run time systems.

Deo Optimo Maximo

In memory of Edward Lecler and Gladys Bowen Lecler.

ACKNOWLEDGEMENTS

Throughout my graduate study I have had the support of exceptional colleagues, steadfast friends, and loving family during the completion of this dissertation. These acknowledgments include only a small subset of all the people who have helped to make this dissertation possible.

My advisor Jan Prins taught me how to be a researcher and put in countless hours to that end. Allan Porterfield showed me how to work in the bowels of multithreading libraries, funded me through his MAESTRO grant, and has been a tireless mentor and advocate. Bronis de Supinski connected me with the OpenMP community and the broader HPC community. He also directed my work during my semester at Lawrence Livermore National Laboratory. Jim Anderson taught me how to reason about concurrency, and Rob Fowler introduced me to counter-based performance measurement.

I thank the High Performance Computer Modernization Office for their sponsorship of my National Defense Science and Engineering Graduate Fellowship, which funded me for three years and gave me the freedom to pursue my research interests. Thanks to the RENCI and UNC Research Computing for the use of their computing facilities.

The computer science department at Carolina is an exceptional place to work. Thanks especially to Tim Quigg for making students a priority. Many of my fellow students have become cherished friends, including Keith Lee, Jamie Snape, Bjoern Brandenburg, Sasa Junusovic, Aaron Block, Russ Gayle, Srinivas Krishnan, Jason Sewall, Robbie Cochran, Catie Welsh, Cory Quammen, Tabitha Peck, Todd Gamblin, and Jeff Terrell. I am grateful to the generous alumni whose support of the Alumni Fellowship has allowed me to focus on my dissertation during my last year.

Coral Kelso and Ron Dupuis first introduced me to computer science back in the Nineties. Thanks to them, and all my former teachers and professors, for setting me on the path of knowledge.

My best friends Lap Trinh and Denis Gjoni have been a constant source of support, along with my many other friends back in Texas. I am also greatly indebted to my friends from the UNC Newman Center, including Cathy McCurry, Eric Burns, Danny Kumar, Kim Burke, Mary Landry,

Bryan Davis, and Erin Buller. Thanks especially to Susan Metallo, with whom I've shared the ups and downs of the final stages of graduate school.

My most heartfelt thanks go to my parents, Cindy and Errol Olivier, and my sister Leslie. No words could express how grateful I am for your constant love and support. Finally, it is to my Lord Jesus Christ that I owe all that I have and my very being. Doing God's will is the ultimate goal of this and all my life's work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
BOTS	Barcelona OpenMP Tasks Suite
CnC	Concurrent Collections
CQ	Centralized Queueuing
GCC	GNU Compiler Collection
DAG	Directed Acyclic Graph
FEB	Full/Empty Bit
ICC	Intel C Compiler
MTS	Multithreaded Shepherds
NUMA	Non-Uniform Memory Access
PDF	Parallel Depth-First
PGAS	Partitioned Global Address Space
QPI	QuickPath Interconnect
TBB	Threading Building Blocks
UTS	Unbalanced Tree Search
WS	Work Stealing

CHAPTER 1

INTRODUCTION

Emerging trends in computer hardware present new opportunities and challenges for high performance scientific computing. In this introductory chapter, we explain how these trends motivate our work on locality awareness for task parallel computation, assert our thesis, and describe the contributions and organization of this dissertation.

1.1 Multicore Architectures

In the context of this dissertation, parallel computation is the use of concurrent execution of operations to achieve performance, i.e., faster execution times, not possible with sequential (non-concurrent) computation. While parallel computers date back to the earliest electronic computer systems (Goldstine and Goldstine, 1946), only recently has parallel computation become the primary route to increased performance. Throughout the second half of the 20th century, computational performance of sequential programs grew exponentially and, from the point of view of the programmer, transparently. A program written in one year could execute several times faster in succeeding years with minimal programmer effort due to increases in clock speed frequency and hardware-managed instruction level parallelism (ILP), such as deep pipelining and multiple arithmetic units.

In the mid-2000's, power limitations halted the growth in clock speed and ILP. However, demand for application performance did not diminish. Vendors adopted a radically different approach. They abandoned complex high frequency microprocessor designs and instead replicated lower frequency central processing units (CPUs) with shorter pipelines on the same die, an architectural alternative demonstrated a decade earlier at Stanford University (Olukotun et al., 1996). Each CPU replica is called a *core*, and the chip as a whole is called a *multicore processor* or *chip multiprocessor*.

Operating at theoretical peak performance, a 1 Ghz four-core processor can perform the same calculations in the same time using less power than a 4 Ghz single-core processor using the same instruction set. However, concurrency is required to use all cores and maximize performance. The operating system can simultaneously launch different applications, or separate instances of a scientific simulation with different experimental parameters, on each core. In this paradigm, called *capacity computing*, the goal is to execute as many sequential application instances as possible in an interval of time. On the other hand, the operating system can instead launch a single application that uses all the cores of the chip in a coordinated effort to solve a single large problem, such as a high resolution simulation. In this paradigm, called *capability computing*, the goal is to execute the single application in as short a time as possible. Unlike capacity computing, capability computing requires the programmer to either write a parallel program or to convert an existing sequential program into a parallel program, and requires efficient hardware and software infrastructure to coordinate parallel execution. This dissertation extends the state-of-the-art in software infrastructure for capability computing on multicore architectures.

The number of cores per chip in production microprocessor designs has increased from one to sixteen in the period from 2006 to 2012, and Intel has announced a production processor with more than 50 cores. The complexity of memory subsystems has also increased to keep pace with concurrent memory requests. In addition to each core's private instruction and data caches, cores on the same chip often share a larger last-level cache. Multiple on-chip memory controllers distribute memory references to banks of memory through multiple memory channels. In many configurations, two to eight multicore chips are installed on the same motherboard and connected using a high speed interconnect. Each chip is installed in a *socket* with direct links to banks of *local memory* and additional links to other sockets to access banks of *remote memory*. Figure 1.1 shows an example four-socket multicore system with memory attached to each socket.

The enterprise server market drives the development of the hardware technology used in the smaller scientific computing market. Multi-socket systems serve well as centralized servers for capacity computing in enterprise applications, but they present a challenge for capability computing in HPC applications. The distribution of memory banks and controllers throughout a multi-socket system increases memory concurrency and available bandwidth but introduces differences in data access times. Despite the presentation of all system memory as a unified global address space,

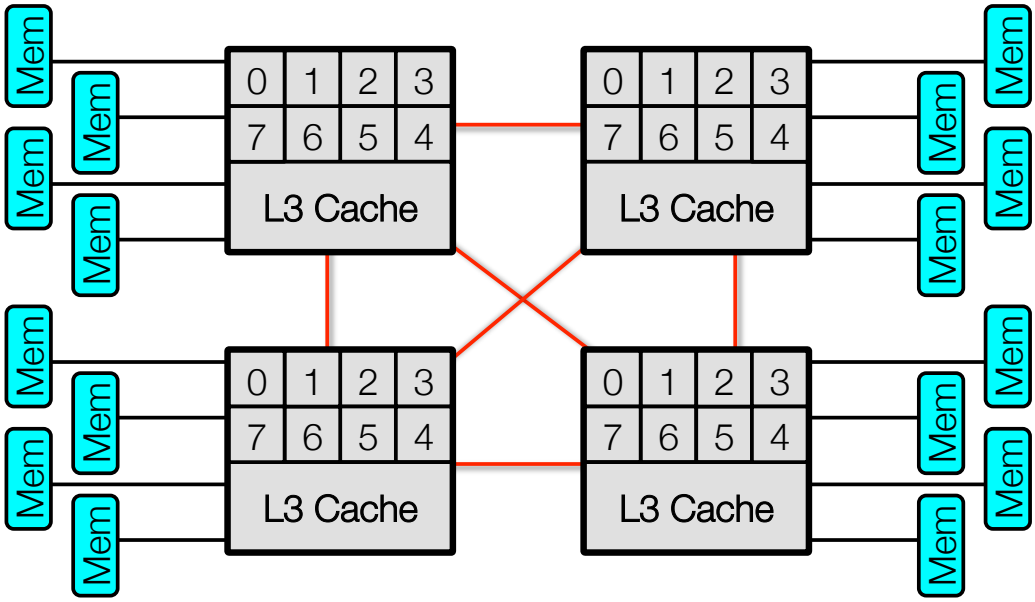


Figure 1.1: A typical four-socket X86 multicore system.

differences in access times between remote memory, local memory, and the various caches impact the efficiency of parallel computations executed on these systems. This dissertation characterizes these effects and proposes strategies to mitigate them.

1.2 Shared Memory Parallel Programming

Modern operating systems provide support for parallel execution on the multiple cores of a shared memory computer through *multithreading*. An application process can comprise multiple threads of control that share the same code and address space in memory. Each thread has its own program counter, copy of the registers, and stack. The operating system creates an initial thread for each application process. Any additional threads are created and managed by calls to the operating system from the process code, for example using the POSIX threads (Pthreads) application programming interface (API) (IEEE, 1995). While the Pthreads API gives a high level of control through explicit thread management and low-level synchronization, e.g., mutual exclusion to guard access to shared variables, effectively coordinating computational work among the threads often requires considerable programmer effort and code changes.

```
#pragma omp parallel for
  for (i = 0; i < nrows; i++) {
    for(j = 0; j < ncols; j++) {
      for (k = 0; k < nrows; k++) {
        c[i][j] = a[i][k] * b[k][j];
      }
    }
  }
```

Figure 1.2: Thread parallel matrix multiplication in OpenMP.

The OpenMP API provides a portable higher-level programming environment for shared memory programming (OpenMP Architecture Review Board, 2008) through the support of the compiler and a run time system for thread management. The programmer specifies *parallel regions*, i.e., code to be executed by multiple threads. A team of threads is implicitly created to execute the parallel regions. The OpenMP run time also maintains mutual exclusion for critical sections and coordinates progress of the team of threads. OpenMP was originally designed for and is particularly well suited to the expression of *loop-level parallelism*, in which different iterations of a loop are executed concurrently. Within parallel regions, the programmer can mark a `for` loop for parallel execution. The loop iterations are scheduled among the threads in the team using either a static schedule determined by the compiler during compilation or a dynamic schedule determined by the run time system during execution. This programming model has been successfully applied to a large number of computational problems, e.g., matrix multiplication as shown in Figure 1.2. The iterations of the outer loop are spread among the threads that execute the program, and elements of the output matrix `c` are computed in parallel. [1]

1.3 Task Parallel Programming Model

Some forms of application parallelism are not expressed easily using parallel loops. For example, recursive divide-and-conquer algorithms often create subproblems that can be solved in parallel. In the mergesort algorithm, the input array is subdivided into smaller arrays that are recursively sorted and then merged. The control flow of the program is clearly not iterative. The quicksort algorithm presents an additional challenge in that the division of work between subproblems is frequently

[1] While this code closely reflects the basic algorithm for matrix multiplication, it is poorly tuned to the memory hierarchy. A better implementation would use loop interchange and tiling for cache performance.

```
void sort(int a[ ], int length) {
  if (length > 1) {
    half = length / 2;
    #pragma omp task
      sort(a[0] , half);
    #pragma omp task
      sort(a[half], length - half);
    #pragma omp taskwait
    merge(a[0], half, a[half], length - half);
  }
}
```

Figure 1.3: Task parallel mergesort in OpenMP.

unequal due to imperfect pivot selection. A still more challenging example is a time-dependent computation using an oct-tree decomposition of three-dimensional space that changes over time to reflect the changing locations of objects in space. Not only do different branches of the oct-tree create subproblems with different amounts of work, but the distribution of work changes over time. The thread-centric focus of loop-level parallelism does not express the parallelism of these applications well.

The *task parallel* programming model is an alternative, complementary model that takes a problem-centric approach to the expression of parallelism. Rather than considering the division of code regions and loop nests among the available threads, the programmer only needs to identify the units of computational work in the application and the dependencies between them. These computational units, called *tasks*, express available parallelism and consist of a segment of code and its data context. Figure 1.3 shows example task parallel code for mergesort. The subproblems that sort subarrays are indicated as tasks that may execute concurrently.

Problem-centric problem decomposition enables scalability by exposing additional available parallelism at larger input sizes. When additional cores are available, that parallelism is exploited without programmer intervention as the run time system distributes the tasks among more threads for greater concurrency. In the task parallel model, it is the responsibility of the run time system to determine on which thread and in what order to schedule each task while respecting the dependencies between tasks. As a result, scheduling decisions have wide-ranging impact on application performance and scaling. The significant body of prior work surveyed in the next chapter attests to the difficulty of achieving high performance on even idealized machine models. The multicore architectures now ubiquitous in

scientific and technical computing deployments have much to gain from the productivity potential of task parallel programming, but their increasing complexity makes performance even more elusive. This dissertation describes systematic performance analysis techniques to diagnose the sources of performance loss and new scheduling strategies for efficient execution on these architectures.

1.4 Thesis

Existing performance analysis methods and run time systems that implement the task parallel programming model are agnostic to characteristics of the target hardware, limiting their efficacy. These characteristics include increased memory latency to access remote data, cache effects, and inter-chip synchronization costs. During execution, scheduling decisions that neglect hardware characteristics result in stalls on load instructions and increased overhead costs. This dissertation measures the impact of these issues and demonstrates the benefits of techniques that incorporate knowledge of the machine topology into performance analysis, run time systems, and program code. Our thesis is as follows.

> *A run time scheduler informed by knowledge of the hardware architecture and locality specifications in the application code can improve the performance of task parallel programs in comparison to performance using uninformed schedulers.*

1.5 Contributions

Performance of task parallel programs is limited by overhead costs, load imbalance, and data access costs due to non-uniform memory access (NUMA) and other related issues in the memory subsystem. This dissertation contributes to better understanding and mitigation of these issues in the following ways.

We introduce a component model for performance analysis of task parallel programs that attributes loss of parallel efficiency to overhead costs, idleness, and data access costs among threads in the execution. Application of our model to executions of a diverse set of benchmark applications reveals substantial inefficiencies that motivate the need for improved task scheduling.

We demonstrate the trade-off between maintaining load balance and limiting overhead costs. We define the Unbalanced Tree Search (UTS) benchmark to evaluate load balancing capabilities of parallel systems. We identify inadequate load balancing strategies, subsequent thread idleness, and overhead costs as primary factors contributing to poor speedup of task parallel executions of UTS, and show that better performance is possible using aggregation in load balancing operations.

We define the concept of *work time inflation* in task parallel executions as the total additional time spent by all threads on the work of the computation beyond the time required to perform the same work in a sequential execution. We characterize the performance loss in the execution of task parallel computations due to work time inflation, which arises from cache misses, memory bandwidth saturation, and non-local memory accesses on NUMA architectures.

We propose a hierarchical task scheduling strategy that targets modern multi-socket multicore shared memory systems with NUMA architectures that are not well supported by either work-stealing schedulers with one queue per core or by centralized schedulers. Our approach combines work stealing and shared queues for low-overhead load balancing and exploitation of shared caches.

We describe a working prototype OpenMP run time system implementing our hierarchical scheduler, as well as other task scheduling strategies. The run time extends the work of collaborators at the Renaissance Computing Institute (Porterfield et al., 2011) and Sandia National Laboratories (Wheeler et al., 2008) that enables the compilation and execution of a wide range of task parallel programs expressed in OpenMP.

We evaluate our hierarchical scheduler for OpenMP tasks on a 32-core Intel system. We compare against other schedulers implemented in our OpenMP run time and those from GNU and Intel, the first such performance study on a multi-socket multicore machine. Both improved speedup and several secondary metrics confirm the benefits of hierarchical scheduling. Evaluations on AMD and SGI machines show that our methods extend to different processors and interconnects, and we demonstrate the first scaling of OpenMP tasks on over 100 cores.

We propose a framework that enables locality-based task scheduling to minimize non-local memory accesses on NUMA architectures. The framework comprises a concise mechanism for the programmer to specify the placement of tasks on locality domains, and a run time scheduler to support that mechanism.

We evaluate the performance of our locality-based scheduling framework as implemented in extensions to the hierarchical scheduler of our OpenMP run time system on a multi-socket multicore NUMA architecture. Both speedup results and performance metrics from hardware performance counters confirm the effectiveness of the framework.

In addition to the original contributions listed above, our survey of prior work provides an overview of the landscape of task parallel computation, particularly task scheduling algorithms.

1.6 Organization

The remainder of the dissertation is organized as follows. Chapter 2 reviews prior work on the task parallel model. Chapter 3 presents an analysis of idleness and overhead costs in task parallel computation through the UTS load balancing benchmark. Chapter 4 defines our component model of performance analysis, including work time inflation. Chapter 5 describes the implementation of our run time system for OpenMP. Chapter 6 describes our hierarchical scheduler and provides a detailed performance evaluation. Chapter 7 defines our framework for explicit locality-based scheduling. Chapter 8 offers our final conclusions and some directions for future work.

CHAPTER 2

BACKGROUND AND PRIOR WORK

The body of knowledge concerning the task parallel programming model is both broad and deep. We examine prior work regarding several key aspects: the expression of task parallelism through language syntax and semantics, theoretical analysis of task scheduling algorithms, run time systems to support task parallelism, and software tools and methods for performance analysis. A complete software development environment combines elements of each, and we point out some of the interrelationships between them.

2.1 Expressing Task Parallelism

Many of the concepts used in task parallel programming languages originated in early attempts at multithreaded programming using functional languages, e.g., MultiLisp (Halstead, 1985). Cilk borrowed from and expanded upon these concepts to design and to implement a seminal task parallel extension to the declarative C programming language (Blumofe et al., 1996; Frigo et al., 1998). Its authors made several key choices in favor of language simplicity, compiler support, and run time efficiency that have been widely imitated in more recent task parallel languages such as OpenMP 3.0 and X10, although other implementation choices have not.

A Cilk program is a C program with three additional keywords: `cilk`, `spawn`, and `sync`. The `cilk` keyword indicates the declaration of a Cilk procedure, i.e., a function that may be executed in parallel. Parallel invocations to Cilk procedures are made using the `spawn` statement. The `sync` statement directs the run time to wait for the completion of any outstanding Cilk procedures spawned by the current procedure up to that point.

Like Cilk, OpenMP's task parallel model is expressed through a compiler-supported language extension. Version 3.0 of the OpenMP specification for Fortran and C/C++ shared memory par-

```
cilk int fib(int n) {
  if (n < 2)
    return n;
  else {
    int x, y;

      x = spawn fib(n-1);

      y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

```
int fib(int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    #pragma omp task
      x = fib(n-1);
    #pragma omp task
      y = fib(n-2);
    #pragma omp taskwait
    return (x+y);
  }
}
```

Figure 2.1: Calculating Fibonacci numbers in Cilk and OpenMP.

allel programming added explicit task parallelism to complement its existing data parallel constructs (OpenMP Architecture Review Board, 2008). The `task` and `taskwait` directives resemble Cilk `spawn` and `sync` statements respectively, as shown in Figure 2.1. However, the OpenMP `task` directive generates a task from a statement or structured block, not a procedure. In addition to the `taskwait` synchronization, the OpenMP `barrier` directive also provides task synchronization. Threads encountering a barrier must complete all outstanding tasks generated by threads in that team before they may pass the barrier. Data clauses define whether tasks get shared, private, or initialized private copies of variables from surrounding scopes.

The Cilk language and prior proprietary extensions of OpenMP for task parallelism, e.g., the Intel work-queueing model (Su et al., 2002), impose strict scheduling restrictions. However, the OpenMP 3.0 task model allows considerable flexibility in scheduler design (Ayguadé et al., 2009). This flexibility, along with the wide-spread support of OpenMP among hardware vendors and parallel programmers, prompted our choice of OpenMP 3.0 as the language to target in our scheduler implementations.

An unexecuted task may be scheduled onto any thread. OpenMP defines two classes of tasks based on restrictions of task suspension and rescheduling: *tied* and *untied* tasks. They differ in two important ways. First, a tied task may only be suspended at specific *task scheduling points*, e.g., generation of new tasks, `taskwait` synchronizations, and barriers. An untied task may be suspended at any point during execution. Second, a tied task can be scheduled initially onto any thread but is not allowed to migrate between different threads during execution. An untied task

```
...
int nodecount;
#pragma omp threadprivate(nodeCount)

void traverse(Node* node)
{
  nodeCount++;
  for (i = 0; i < node->numChildren; i++) {
     #pragma omp task firstprivate(i)
       traverse(node->childNode[i]);
  }
}

int main(int argv, char *argc[])
{
   int total;
   ...
   #pragma omp parallel
   {
     #pragma omp single
       #pragma omp task
         traverse(root);

     #pragma omp barrier

     #pragma omp atomic
       total += nodeCount;
   }
   printf("Total=%d\n", total);
   ...
}
```

Figure 2.2: Counting tree nodes in OpenMP.

may migrate between different threads during execution. Untied tasks allow for greater flexibility in scheduling, but care must be taken if a *threadprivate* (thread-local) variable is used in the task. A threadprivate variable is a global variable of which each thread has a copy. If such a variable is updated without an `atomic` directive, a race condition between untied tasks could potentially be created. The task may be swapped out in favor of another task that also modifies it.

Threadprivate variables enable efficient computation of a global summation. For example, Figure 2.2 shows an example program that uses tied tasks and a threadprivate variable to count nodes in a tree. In this program use of the threadprivate variable to count nodes locally on each thread limits the number of atomic operations to $\Theta(p)$, where p is the number of threads and requires no `taskwait` synchronizations. An alternative implementation without threadprivate variables would

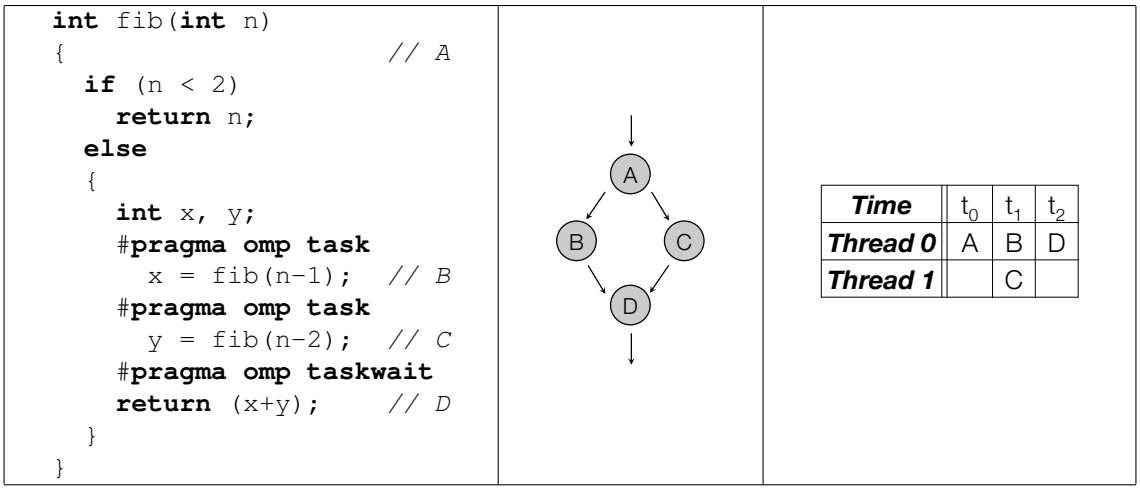


Figure 2.3: Code, task graph, and schedule of *fib* on two threads.

pass the counts as return values of the function calls, but that would require $O(\log n)$ `taskwait` synchronizations to wait for the intermediate values to be returned. The generation of the first task in `main` occurs within the context of OpenMP `parallel` and `single` constructs. The `parallel` construct opens a parallel region, creating a team of threads to execute the tasks. The `single` construct specifies that only one of those threads should create the first task, leaving the others to execute descendants of that task.

2.2 Scheduling Theory

A task scheduler assigns each task in the computation to a worker thread for execution, according to the constraints expressed as dependencies among tasks and the scheduling policy. Because this dissertation proposes new scheduling strategies for task parallelism, we now examine several existing task scheduling policies, along with their strengths and weaknesses. [1]

2.2.1 Task Parallel Computations as DAGs

The execution of a task parallel computation is viewed as a dynamically unfolding task graph. A task graph is a directed acyclic graph (DAG) in which each node represents a task instance and each edge represents a dependency between tasks: a new task creation (spawn edge) or synchronization

[1] Proofs and proof sketches in this section are condensed forms of the detailed proofs that appear in the papers cited for each theorem.

(join edge). An execution schedule is a mapping of the task graph onto p worker threads for execution in which each node appears exactly once in the schedule, no node may be scheduled before its ancestors, and at most p nodes are scheduled concurrently. Figure 2.3 shows the code for a function *fib*, a corresponding DAG, and a schedule for an execution of *fib* on two threads.

The computations that we consider in this dissertation are a subset of the class of all computations that can be represented as DAGs. In the general case, there can be an edge in a DAG from any vertex v_1 to any vertex v_2, as long as there is no path from v_2 back to v_1. For a computation, this would represent the possibility that any task may have a dependency on any other task. Instead, we consider only a set of computations with a restricted set of possible task dependencies. In these computations, called *strict* computations, each dependency must be between a task and one of its ancestors. In a *fully strict* computation, each dependency must be between a task and its parent (Blumofe and Leiserson, 1999). A *terminally strict* computation is a strict task in which each synchronization dependency is fulfilled upon the completion of the descendent task, not by the completion of some earlier instruction of the descendent task (Agarwal et al., 2007). The example in Figure 2.3 is both fully strict, since the two dependencies are both on child tasks, and terminally strict, since those child task must complete to satisfy the `taskwait`. This dissertation concerns only computations that are at least terminally strict.

Perfect nesting required by the shallow scope of Cilk `spawn` and `sync` statements ensures that Cilk computations are both fully strict and terminally strict. However, the fully strict model may impose constraints on the task graph that are not inherent in the problem. Given a problem in which only synchronization between the root task and last-level (leaf) tasks is required, the more general strict model allows parent tasks to complete earlier or later than their child tasks, potentially enabling earlier deallocation of space than the fully strict model. OpenMP task parallel computations are fully strict and terminally strict if perfectly nested `taskwait` synchronizations are used, but only terminally strict if task synchronization is done using a barrier. The task scheduling policies described in the following sections are applicable in either case, but the space bounds are conservative, i.e., they assume parent tasks are deallocated after their child tasks have completed.

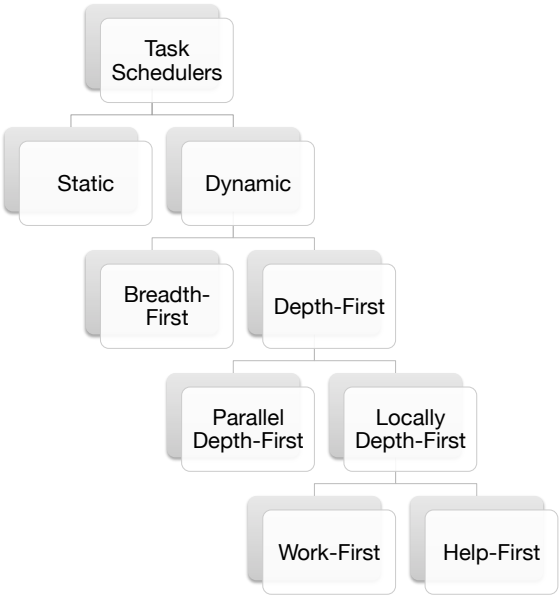
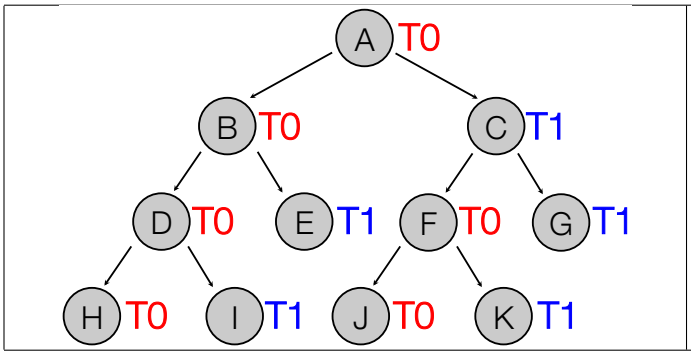


Figure 2.4: A taxonomy of task scheduling policies.

2.2.2 Overview of Task Scheduling Policies

Prior to examining particular schedulers in detail, consider the broad taxonomy of scheduling policies summarized in Figure 2.4. Static scheduling is not applicable for computations in which tasks are generated dynamically and not known a priori (Johnson et al., 1996). Even for applications in which the task graph is known a priori, static schedulers fail to provide dynamic load balancing required for efficient execution of many task parallel programs (Blumofe and Papadopoulos, 1998). Thus, we do not consider static scheduling in this dissertation.

For dynamic schedulers, the task graph concept allows us the expression of scheduling policies in terms of the order in which a DAG is traversed. A *breadth-first* scheduler executes all tasks at a level n of the task graph before executing tasks at level n+1, if we consider the level of the root task as the 0-th level. A *depth-first scheduler* executes all tasks in a single downward path before backtracking to execute the most recently generated but unexecuted tasks. In the context of parallel execution, there are two notions of depth-first: In a *parallel depth-first* (PDF) schedule, worker threads coordinate to follow a schedule close to the sequential depth-first schedule of the task graph. In a *locally depth-first scheduler*, each worker thread proceeds depth-first through a different subgraph of the overall task graph. Locally depth-first schedulers are further distinguished by the action taken upon task generation. A *work-first* scheduler immediately schedules each newly



Thread	Steps					
T0	A	B	D	F	H	J
T1	-	C	E	G	I	K

Figure 2.5: Task graph and breadth-first schedule on two threads.

generated task, while a *help-first* scheduler enqueues each new task and continues execution of the parent task.

2.2.3 Breadth-first and Greedy Scheduling

In a breadth-first schedule, all tasks at depth i relative to the root are executed before tasks at depth $i + 1$. Figure 2.5 gives an example task graph and a breadth-first schedule for its execution. In the initial step, thread T0 executes task A. Task A generates tasks B and C. In the next step, thread T0 executes task B, T1 executes task C, and tasks D, E, F, and G are generated. In the following step, thread T0 executes task D, thread T1 executes task E, and tasks H and I are generated. Maintaining breadth-first order, tasks F and G are executed next while tasks H and I remain in memory. Tasks J and K are generated by task F. In the penultimate step, tasks H and I are executed. Finally, tasks J and K are executed.

Richard Brent derived a time bound on multithreaded computations that evaluate arithmetic expressions using a breadth-first schedule (Brent, 1974):

Theorem 2.1 (Brent's Theorem)**.** *For a computation that completes in time T_1 on one processor and in time T_∞ with an infinite number of processors available, the time T_P required to complete the computation on P processors is*

$$T_P \leq T_\infty + \frac{T_1 - T_\infty}{P}. \tag{2.1}$$

Proof. Let s_i be the number of operations performed in step i for $i = 0, 1, 2, ..., T_\infty$. Then step i requires time $\lceil s_i/P \rceil$ using P threads.

$$T_P \leq \sum_{i=1}^{T_\infty} \left\lceil \frac{s_i}{P} \right\rceil \tag{2.2}$$

$$\begin{aligned} T_P &\leq \sum_{i=1}^{T_\infty} \frac{s_i + P - 1}{P} && (2.3) \\ &= \sum_{i=1}^{T_\infty} \frac{P}{P} + \sum_{i=1}^{T_\infty} \frac{s_i - 1}{P} && (2.4) \\ &= T_\infty + \frac{1}{P}\left(\sum_{i=1}^{T_\infty} s_i - \sum_{i=1}^{T_\infty} 1 \right) && (2.5) \\ &= T_\infty + \frac{T_1 - T\infty}{P} && (2.6) \end{aligned}$$

□

To benefit from parallel execution, a multithreaded computation must have sufficient parallel slack, defined as $T_1 \gg T_\infty$ and $T_1 \gg P$. In that case, Brent's Theorem simplifies to

$$T_P \leq T_\infty + \frac{T_1}{P} \tag{2.7}$$

and speedup approaches linear speedup, i.e.,

$$T_P \approx \frac{T_1}{P}. \tag{2.8}$$

In a *greedy* scheduler, when p or more tasks are ready to be scheduled in a time step, p tasks are scheduled. When less than p tasks are ready in a time step, all ready tasks are scheduled (Kleinrock, 1976). Brent's theorem holds not just for breadth-first schedulers, but for all greedy schedulers. The time of any greedy schedule for a computation is within a factor of 2 of the time of an optimal schedule for that computation (Eager et al., 1989).

Proof. Let $T_p^\star$ be the execution time required to complete a computation using an optimal scheduler on P threads. Since T_∞ time is required even with an infinite number of processors available and assuming all work in the serial computation must be done, $T_P^\star = max\{T_\infty, T_1/P\}$.

$$
\begin{aligned}
T_P &\leq T_\infty + \frac{T_1 - T\infty}{P} && (2.9)\\
&\leq T_\infty + \frac{T_1}{P} && (2.10)\\
&\leq 2 \cdot max\{T_\infty, \frac{T_1}{P}\} && (2.11)\\
&= 2 \cdot T_P^\star && (2.12)
\end{aligned}
$$

□

Apart from Brent's work, implementations of the Pthreads multithreading library (IEEE, 1995) use breadth-first scheduling by default. As observed by Narlikar and Blelloch, attempts to use the pthread scheduler to schedule fine-grained tasks typically result in out-of-memory errors even for small programs (Narlikar and Blelloch, 1998). Also, the time bounds in Brent's Theorem do not include overhead times. Breadth-first scheduling is most easily implemented using a FIFO centralized queue, which is prone to contention and fails to exploit cache locality. The suboptimal performance of breadth-first schedulers in practice due to overhead costs and inefficient use of memory and cache motivated the development of depth-first schedulers.

2.2.4 Work Stealing

Work stealing is a locally depth-first, globally breadth-first scheduling strategy that matches the theoretical optimal time bound for greedy schedulers while also mitigating real-world overhead time and using bounded space. Each worker thread has a local double-ended queue. As long as a worker thread has tasks in its local queue, it both enqueues tasks onto and dequeues tasks from the tail of the local queue. This LIFO queueing discipline results in depth-first execution of a subgraph of the overall task graph. When a worker thread's queue becomes empty, the thread becomes a *thief* and attempts to steal a task from the queue of a *victim* thread chosen at random. In a successful steal, the

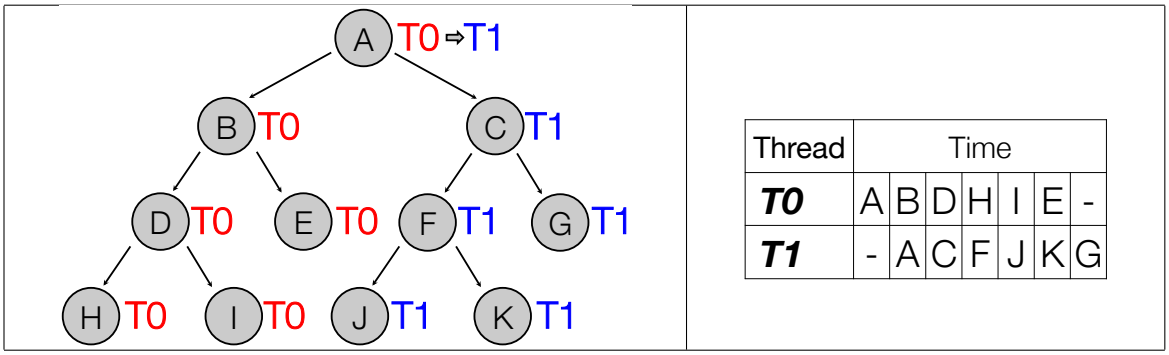


Figure 2.6: Task graph and work-first schedule on two threads.

thief dequeues and executes a task from the head of the victim's queue. In a failed steal attempt, no task is available in the victim's queue. In that case, the thief selects a new victim and again attempts to steal.

In the *work-first* class of work stealing schedulers, a worker thread that generates a new task enqueues the parent task on the local queue and immediately executes the new task. Figure 2.6 gives an example task graph and a work-first schedule for its execution. In the example, the root task A is stolen by thread $T1$ after TO has generated and started execution of task B, so task C and its descendants are generated on thread $T1$. A computation using a work-first scheduler completes in time $T_P = T_1/P + O(T_\infty)$, and requires space $S_P \leq S_1 P$, i.e., linear in the number of threads used (Blumofe and Leiserson, 1999). Proof sketches follow.

Theorem 2.2. *The time required to execute a computation on P processors using a work-first scheduler is*

$$T_P = T_1/P + O(T_\infty). \tag{2.13}$$

Proof Sketch. At each time step, a worker thread is either working on a task or stealing. The total time all threads spend working on tasks is equivalent to T_1, the amount of time required to execute the work on a single thread, assuming overhead costs for task creation do not depend on P.

The topmost task in each thread's queue is either the root or was stolen from some other thread, and a thread only steals when its queue is empty. Thus, the topmost task is the ancestor of all tasks below it in that queue. Then each active path in the computation is rooted at a task at the head of one of the queues. One of these is the *critical path*, a path with the longest chain of dependent tasks, with total path length T_∞. Thus each steal operation results in the stealing of the critical path with

probability $1/P$. Since a stolen task is executed immediately, each steal operation reduces the critical path by one with probability $1/P$. With P expected steal attempts decreasing the critical path by at least 1, and a total of T_∞ steps on the critical path, the total expected stealing time is $O(PT_\infty)$.

Since there are P processors, the execution time is $1/P$ of the total time spent working and stealing by all threads:

$$T_P = \frac{1}{P}\left(T_1 + O(PT_\infty)\right) \tag{2.14}$$

$$= T_1/P + O(T_\infty) \tag{2.15}$$

□

In practice, online scheduling incurs overhead costs. Work stealing mitigates overheads by following the *work-first principle*: "Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path" (Frigo et al., 1998). Analysis of the relative impact of overhead costs to the work and critical path supports the work-first principle. Let c_∞ be the smallest constant such that $T_P = T_1/P + c_\infty T_\infty$. Let T_S be the time required to execute a sequential program that is the serial equivalent of the task parallel program. We define the work overhead as the factor $c_1 = T_1/T_S$. Now we can express the time bound incorporating overhead costs:

$$T_P \leq c_1 \frac{T_S}{P} + c_\infty T_\infty. \tag{2.16}$$

Assume the computation has sufficient parallel slack, i.e., $T_1 \gg T_\infty$. With high probability, only $O(PT_\infty)$ steals occur. If the time cost of each steal is much less than T_1 then

$$T_P \approx c_1 \frac{T_S}{P} \tag{2.17}$$

Stealing costs are borne on the critical path. Thus, the remaining constant c_1 reflects overhead cost unrelated to stealing, such as the cost of task creation and destruction. These costs are implementation-dependent. The authors of Cilk report values of c_1 between 1 and 1.25 for most applications (Frigo et al., 1998). In comparison to a centralized breadth-first scheduler, these costs are generally

much lower in a distributed work-first scheduler because local enqueue and dequeue operations are lower latency than remote queue operations and the queues in a work stealing scheduler are rarely contended.

Theorem 2.3. *The space required to execute a computation on P processors using a work-first scheduler is*

$$S_P \leq S_1 P. \tag{2.18}$$

Proof Sketch. Define a *leaf task* as a task with no non-completed children. Define a *primary leaf task* as a non-completed leaf task generated earlier than any of its non-completed siblings. A schedule maintains the *busy-leaves property* if every primary leaf task has a thread working on it. We can reason by induction that a work-first schedule maintains the busy-leaves property.

Base case: At the start of the computation, the root task is a primary leaf task because it has no siblings and initially has no children. A thread works on the root task until it generates a child.

Inductive step: Assume the busy-leaves property: each primary leaf task has a thread working on it. Consider possible schedule actions:

1. A primary leaf task *t* generates a child task. Then *t* is no longer a primary leaf task, and it's child is now a primary leaf task. The thread working on *t* immediately suspends *t* and schedules its child for execution. The busy-leaves property is maintained.

2. A primary leaf task *t* completes and *t* has non-completed siblings. The sibling which was generated earliest is now a primary leaf task. The thread that was working on *t* schedules that sibling for execution. The busy-leaves property is maintained.

3. A primary leaf task *t* completes and *t* has no non-completed siblings and its parent is non-completed and not already executing on some other thread. The parent of *t* is now a primary leaf task. The thread that was working on *t* schedules its parent for execution. The busy-leaves property is maintained.

4. A primary leaf task *t* completes and *t* has no non-completed siblings and its parent is either completed or executing on some other thread. The thread that was executing *t* is idle and begins stealing. The busy-leaves property is maintained.

5. A task *t* that is not a primary leaf task becomes a primary leaf task following a steal. The thread executing *t* was idle prior to the steal, so no busy task was left unscheduled on its queue immediately before the steal. The task *t* must be scheduled for execution, because it must be the newest task in the leftmost path on the thread and first in the work-first schedule for that thread. The busy-leaves property is maintained.

We can associate each non-completed task with a primary leaf task according to the following rules: If the task is a primary leaf task, it is associated with itself. It the task is non-leaf task, then it is associated with the same primary leaf task as its earliest non-completed child. If the task is a leaf task that has earlier non-completed siblings, then it is associated with the same primary leaf task as its earliest non-completed sibling. The rules apply recursively. The tasks associated with a primary leaf are a subset of the non-completed tasks in a sequential execution of the computation. Thus, the total space of these tasks is at most S_1, the space of a sequential execution. By the busy-leaves property, each primary leaf task has a thread working on it. Since there are P threads, there are at most P primary leaf tasks. Thus, the total space requirement is at most $S_1 P$. □

Help-first scheduling is a variant of work stealing in which a worker thread that generates a new task enqueues the new task on the local queue and continues execution of the parent task. The space bounds for work-first scheduling are not guaranteed to hold for help-first scheduling. In general, more space is required for help-first scheduling than for work-first scheduling, since sibling tasks at each level of the graph can be on the stack at once. However, the earlier generation of sibling tasks exposes available parallelism earlier in the execution, which can be helpful for task graphs that are wide and shallow (Guo et al., 2009). Also, help-first scheduling is useful when the migration of partially completed tasks is disallowed, as with tied tasks in OpenMP. Under that condition, a partially completed parent task could never be stolen, so a work-first scheduler would serialize the execution. Newly generated and unexecuted child tasks generated by a help-first scheduler can be stolen, enabling parallel execution.

2.2.5 Parallel Depth-first Scheduling

Work-first schedules are locally depth-first because each thread dequeues tasks in LIFO order. The space bound $S_1 P$ represents the simultaneous presence in memory of allocated tasks on P

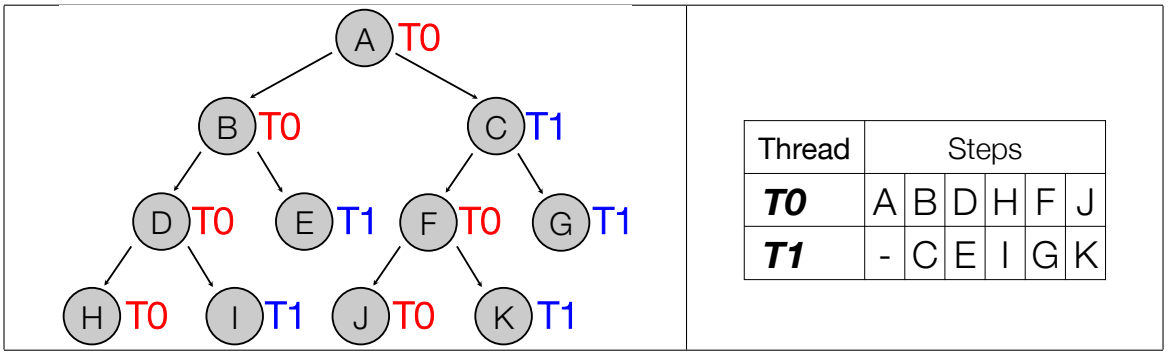


Figure 2.7: Task graph and PDF schedule on two threads.

paths of the computation graph. The parallel depth-first (PDF) schedule is an approach designed to coordinate the threads to execute as close as possible to a single path in the computation, resulting in a lower memory requirement. The particular single path to be followed is the sequential order. Because sequential execution is depth-first, concurrent execution according to sequential order can be approximated using a LIFO queue shared among all threads. A departure from sequential order is required when there is not enough available parallelism in the current branch of execution to keep all threads busy. In that case, an idle thread must then schedule a *premature task*, a task with predecessors in the serial order that have not all been executed. Figure 2.6 gives an example task graph and a work-first schedule for its execution. In the example, two tasks, C and G, are executed prematurely by thread $T1$. The space bound for a PDF schedule is $S_P = S_1 + O(PT_\infty)$, and the reasoning is based on bounding the number of premature tasks scheduled (Blelloch et al., 1999).

Theorem 2.4. *The space required to execute a computation on P processors using a PDF scheduler is*

$$S_P \leq S_1 + PT_\infty. \tag{2.19}$$

Proof Sketch. Consider a path in the task graph. Each task along the path and its siblings comprise a level. The current level is the level that holds the deepest non-completed tasks that occur earliest in the serial order. In a time step in which a premature task is scheduled, all tasks on the current level must have been scheduled on or before that time step. Otherwise, unscheduled tasks on the current level would have been scheduled instead. There must always be at least one task at the current level, because otherwise it would cease to be the current level and the next level of tasks in the serial order would be the current level. Thus, at most $P - 1$ tasks must be scheduled prematurely each time

step to keep all threads busy. The longest possible path in the computation is T_∞, so there are at most $(P-1)T_\infty$ premature tasks with space allocated at any time. A schedule exactly following the sequential order would have space S_1, so the total space bound adds to that the maximum space for premature tasks:

$$S_P \leq S_1 + (P-1)T_\infty \tag{2.20}$$

$$S_P \leq S_1 + PT_\infty \tag{2.21}$$

□

When P tasks are available on the shared queue, PDF schedules P tasks. When less than P tasks are available on the shared queue, it schedules all available tasks. Thus, a PDF schedule is a greedy schedule. By Brent's Theorem, it has the time bound $T_P = O(T_1/P + T_\infty)$ given sufficient parallel slack. In practice, a PDF scheduler implementation can incur overhead costs due to contention on the shared task queue.

2.2.6 DFDeques

Work stealing uses individual per-thread task queues rather than the contention-sensitive shared queue of a PDF scheduler. However, PDF has better space bounds. The DFDeques scheduler explicitly parameterizes a limit on the amount of space used, allowing sufficient parallelism to be exposed without exceeding memory capacity or driving up queue contention. As in work stealing, each thread schedules tasks from a local queue, but the number of queues can exceed the number of threads. When the memory consumed on a thread exceeds the limit K, the task currently executing on that thread is suspended, the entire queue is deferred, and a new queue is stolen for execution. The space bound on DFDeques is $S_P(K) = S_1 + O(KPT_\infty)$ and the time bound is $T_P = O(T_1/P + T_\infty)$ (Narlikar, 1999). Work stealing is a special case of DFDeques in which no queue is ever deferred due to exceeding memory constraints.

2.2.7 Bounds on Cache Misses

Data access time contributes to actual execution time for both sequential and multithreaded applications. Since data in a cache can be accessed with lower latency than data in main memory, the

number of cache misses during an execution should be minimized. There is a class of algorithms, known as *cache oblivious algorithms*, that attempt to exploit cache locality using a recursive divide-and-conquer strategy (Frigo et al., 1999). The subproblem size becomes smaller at each level of the recursive problem decomposition, and at some level the data for a level fits into cache without spilling. Also, function calls at adjacent levels of recursion typically use a subset of the same data that can be resident in cache due to temporal locality. For example, a matrix of size $n \times n$ can be decomposed into four matrices of size $n/2 \times n/2$.

Consider a parallelization of divide-and-conquer algorithms using the task parallel model. Recursive function calls become tasks. Since the sequential strategy has good locality, it is desirable to schedule the tasks so as to maintain that locality as much as possible. Analogous to the notion that function calls at adjacent levels of recursion use an overlapping set of data is the notion that a parent task and its child task use an overlapping set of data. For best maintenance of cache locality, the parent and child tasks should be scheduled successively and by either the same thread or a set of threads that share the same cache.

The LIFO queue discipline of work-first scheduling is designed to exploit private (per-core) caches. The most recently generated tasks are near the tail of the queue, and tasks are dequeued locally from the tail to be scheduled for execution. When an idle thread steals, the stolen task is taken from the head of the victim's queue. It is thus the least recently created task and least likely to have associated data resident in cache. If the depth of the queue is small, however, a cache miss that follows the steal is a miss that would not have occurred if the task had not been stolen. The intuition that the number of cache misses is a function of the number of steals is confirmed by theoretical analysis (Acar et al., 2000):

Theorem 2.5. *The number of cache misses M_P in a multithreaded computation scheduled by work stealing on P threads is*

$$M_P \leq M_1(C) + 2C \cdot \tau \tag{2.22}$$

where $M_1(C)$ is the number of cache misses in a sequential execution of the computation, C is the cache size, and τ is the number of steals.

Proof Sketch. Define a *drifted task* to be a task that in a multithreaded computation does not execute on the same thread as and immediately after the task that precedes it in a single processor execution

of the same computation. Each steal results in at most two drifted tasks. The stolen task is a drifted task. If its parent has a dependency on the stolen task then the join with the parent makes the parent a drifted task when it is rescheduled. The total number of cache misses is the sum of the cache misses in a sequential execution, $M_1(C)$, and the number of cache misses due to stealing, $2C \cdot \tau$. □

Recall that in the proof sketch for the time bound on work-first scheduling, we derived the expected number of steals as a function of critical path length and the number of threads, PT_∞. Acar et al. refine this expected value to include m, the time spent by an instruction that incurs a cache miss, and s, the time required to perform a steal (Acar et al., 2000). Their analysis gives

$$\tau = O\left(\left\lceil \frac{m}{s} \right\rceil \cdot P \cdot T_\infty\right). \tag{2.23}$$

Acar et al. also derive a time bound based on their analysis of the cache misses:

$$T_P = O\left(\frac{T_1(C)}{P} + m \cdot \left\lceil \frac{m}{s} \right\rceil \cdot C \cdot T_\infty + (m+s) \cdot T_\infty\right). \tag{2.24}$$

Note that this is a refinement of the bound of Blumofe et al., $T_P = T_1/P + T_\infty$.

While work stealing is well suited to private caches, PDF scheduling is well suited to shared caches. With the exception of data used by premature nodes, the cache usage of a PDF schedule is the same as that of a sequential execution. With a shared cache of size $C_P = C_1 + PT_\infty$, where C_1 is the size of the cache used in a single processor execution, a PDF schedule incurs at most M_1 misses, the same number of misses as the sequential execution (Blelloch and Gibbons, 2004). The proof resembles the proof of the space bound on PDF scheduling. Since a work-first schedule has P active paths in memory, the capacity required in a shared cache to limit the number of misses to the same number as a single processor execution is linear in the number of threads. With a shared cache of the same size as the cache used in a single processor execution, the number of cache misses is $M_P = M_1 + O(PT_\infty)$. Empirical studies demonstrate that the shared cache performance of PDF scheduling outperforms that of work stealing in practice (Chen et al., 2007). On the other hand, PDF scheduling can cause sharing or false sharing of data in cache lines in a system using private caches, which causes the cache lines to bounce back and forth between cores.

2.2.8 General DAG Scheduling

Computations that are not at least terminally strict, such as those generated in languages that support futures (Friedman and Wise, 1978; Halstead, 1985), are outside of the scope of this dissertation. Such computations with arbitrary dependencies can be scheduled efficiently in time (Arora et al., 2001). However, some are impossible to schedule efficiently in space (Blumofe and Leiserson, 1998), and the space bounds of schedulers for fully strict and terminally strict computations are not guaranteed to hold for the set of all possible multithreaded computations representable by DAGs.

2.3 Run Time System Implementations

In addition to the choice of scheduling policy, other design choices and implementation details impact the efficiency of run time systems. Moreover, each run time system implementation must conform to the semantics and constraints of the particular task parallel language it supports.

2.3.1 Cilk

The Cilk run time system uses work-first scheduling and attempts to limit overheads on the work term at the expense of the critical path (Frigo et al., 1998). In concrete terms, that means task creation and retirement should be fast while steal operations may be slower. To that end, the compiler generates two versions of each Cilk procedure: a fast clone and a slow clone. The fast clone resembles a regular C stack frame, while the slow clone has bookkeeping support for parallelism, e.g., for synchronization with child tasks. The run time instantiates each newly generated task as a fast clone. In the absence of stealing, the bookkeeping facilities of the fast clone are not needed, because upon completion of its child it will simply be returned to execution. The `sync` operation is free, and the overhead on the work term is only a few cycles. In the event of a steal, the stolen procedure is converted to its slow clone to support synchronization with child tasks that may be on different threads. Since the number of procedures converted to slow clones is equivalent to the number of steals, the cost is borne on the critical path. The reliance of the fast clone / slow clone scheme on strict adherence to a work-first schedule has limited its use outside of Cilk. Help-first

work stealing schedulers cannot use it, and must employ other methods to combat task generation and synchronization overheads.

Work stealing requires concurrent access to each thread's task queue. Again, Cilk follows the work-first principle. To maintain low overheads when enqueuing tasks, the local `dequeue` operation requires no locking. The local `enqueue` operation requires locking only if the queue is nearly empty. The `steal` operation always requires locking, but that overhead cost is borne on the critical path – ideal if executions have parallel slack. The protocol is called T.H.E., after the three variables used to implement it (Blumofe et al., 1996). The ABP work stealing queue was later developed to eliminate locking (Arora et al., 2001), and that queue was demonstrated in the contemporary Hood work stealing library (Blumofe and Papadopoulos, 1998).

2.3.2 OpenMP

The first implementation of OpenMP 3.0 was the open-source Nanos run time developed at the Barcelona Supercomputing Center (Ayguadé et al., 2007). They evaluated different scheduler implementations in Nanos, including centralized breadth-first schedulers and work stealing (Duran et al., 2008b). They observed that tied tasks cannot be scheduled work-first without serialization. Recall that tied tasks cannot be migrated once they have been partially executed. Thus, a tied parent task once suspended cannot be migrated to another thread to continue execution. The Nanos groups also shows that the performance of some applications with very fine-grained tasks can be improved by serializing tasks beyond some *cut-off* or threshold depth in the task graph or when the number of active tasks exceeds the number of threads by some factor. They later developed a scheduler that estimates cut-offs at run time by measuring application characteristics early in execution and applying a set of heuristics (Duran et al., 2008a).

Closed-source commercial compilers from Intel, Sun, Cray, and IBM all support OpenMP tasks. The GNU Compiler Collection (GCC) is the only open-source production compiler with task support. It uses a breadth-first scheduler implemented as a centralized queue. A completely overhauled version is under development for future releases of GCC. OpenUH is a research compiler that uses a help-first work stealing scheduler with two queues per core: a private queue for suspended tied tasks and a shared queue for untied tasks and unexecuted tied tasks (LaGrone et al., 2011). The developers

```
cilk int traverse(Node* node) {
  long nodeCount = 1;
  int i;

  inlet void accumulate (int result) {
     count += result;
  }

  for (i = 0; i < node->numChildren; i++)
      accumulate(spawn traverse(node->child[i]));
  sync;
  return count;
}
```

```
cilk::hyperobject<cilk::reducer_opadd<long> > nodeCount;

void traverse(Node* node)
{
  nodeCount()++;
  for (int i = 0; i < node->numChildren; i++)
      cilk_spawn traverse(node->child[i]);
}
```

Figure 2.8: Counting tree nodes in Cilk and Cilk++.

proposed locality extensions for OpenMP based on thread sub-teams and demonstrated performance improvement on parallel loop-based benchmarks in OpenUH (Huang et al., 2010)

ForestGOMP is an OpenMP library that supports thread parallelism but not tasks. It schedules threads generated from deep loop nests by oversubscription, i.e., generating more threads than available cores. Groups of threads generated together at the same level form a *bubble*, and bubbles are co-scheduled when possible (Thibault et al., 2007). The run time schedules threads using work stealing at two levels: on-chip and off-chip (Broquedis et al., 2010a).

2.4 Other Task Parallel Languages and Libraries

In addition to Cilk and OpenMP 3.0, a plethora of task parallel languages and libraries are available to programmers. They differ in the additional features provided, level of compiler support required, and scheduling restrictions imposed.

Cilk++ and its later incarnation, Intel Cilk Plus, build on the techniques of the Cilk and use C++ as the base language (Intel Corp., 2010). With the addition of the `cilk_for` construct for iteration and vector operations, Cilk++ supports both task, loop, and data parallelism. Cilk allows

embedded function called *inlets* that can be used to combine return values from Cilk procedure spawns atomically, e.g., for reduction operations. Rather than inlets, Cilk++ provides a family of templated classes called *hyperobjects* to share and to update concurrent data objects safely. Hyperobjects are global objects with member functions and overloaded operators to present a well-defined interface to the implicitly synchronized data (Frigo et al., 2009). Figure 2.8 shows example code to count nodes in a tree: The top code segment shows the use of a Cilk inlet and the bottom code segment shows the use of a Cilk++ reducer hyperobject.

Like Cilk++, Intel Thread Building Blocks (TBB) offers data and task parallel programming constructs based on C++ (Kukanov and Voss, 2007). Unlike Cilk Plus, TBB is a user-level library that may be used with any C++ compiler. A TBB task is an object of a derived class that extends the `task` base class. A constructor of that derived class may be used to initialize data members. The `execute` member function contains the user code for the task and returns a pointer to a task. Tasks are allocated using `new` and generated using the `spawn` static member function. The programmer must set a reference count with a value that is one more than the total number of tasks to be generated and synchronized. As tasks complete execution, the reference count is decremented. The `wait_for_all` synchronization function waits for the reference count to decrease to one then resets it to zero. A host of other classes are provided for concurrent data structures, parallel loops, locks, and mutexes (Reinders, 2007).

Microsoft Task Parallel Library (TPL) operates in the Microsoft .NET programming framework as part of the software ecosystem for Microsoft Windows systems (Leijen et al., 2009). At the most basic level, four basic constructs are supported: tasks, futures, replicable tasks, and replicable futures. A task takes a *delegate* (anonymous function) at creation and is executed by a single processor as assigned by the scheduler, and may be waited upon elsewhere in the code by the `Wait` method. A future returns a value with a generic type. A replicable task or future may itself be executed on multiple processors. These constructs are used to create the higher level abstractions of TPL for iteration. Microsoft has implemented several versions of parallel for loops, including a reduction-like `Aggregate` construct.

Other libraries for shared memory include Pfunc, wool, and Java fork-join. The Pfunc library allows the programmer to specify task priorities and other predicates on tasks to influence task

scheduling (Kambadur et al., 2009). Wool is a library-based solution for task parallelism in C (Faxén, 2009). The Java fork-join framework allows the expression of task parallelism in Java (Lea, 2000).

Several projects have applied concepts from Cilk to task parallel computation across distributed memory, including CilkNOW (Blumofe and Lisiecki, 1997), Satin (van Nieuwpoort et al., 2000), and kaapi (Gautier et al., 2007) for clusters and ATLAS (Baldeschwieler et al., 1996) for grids. The Scioto task parallel framework builds upon a software global address space layer for distributed memory and has been shown to scale to 8k cores on some codes (Dinan et al., 2008, 2009). In Scioto, tasks are generated and added to a *task collection*. The explicit `tc_process()` call directs the run time to execute all initial tasks in a collection and any others spawned by those tasks. While only one collection may be active at a time, tasks may be added to other collections at any time to support phased computations.

IBM X10 is a Java-based parallel language designed to meet the dual goals of performance and productivity specified by the DARPA High Productivity Computing Systems (HPCS) program (Charles et al., 2005). X10 targets a distributed cluster of multiprocessor nodes with a Partitioned Global Address Space (PGAS) memory. An `async` statement creates an *activity*. Execution of each activity is fixed to a particular processor or group of processors, called a *place*. The `finish` statement provides deep synchronization: the entire hierarchy of activities created in the scope of the enclosed code block must be completed before execution moves beyond it. In contrast, Cilk can only synchronize a hierarchy of spawned procedures by including `sync` statements at every level. Deep synchronizations on activities created within the scope of an X10 `final` statement result in terminally strict computations that need not be fully strict, but Guo et al. show that work stealing is still feasible for these computations (Guo et al., 2009).

Habanero Java is a derivative of X10 that allows activities to access data in remote places directly (Cave et al., 2011), and a particular extension of Habanero Java called Hierarchical Place Trees allows a hierarchy of places to match the parallel memory subsystem by creating tasks at each level, e.g., L1 cache, L2 cache, L3, memory (Yan et al., 2010). The Scalable Locality-aware Adaptive Work-stealing (SLAW) scheduler for Habanero Java dynamically switches between work-first and help-first schedulers based on available parallelism and space constraints, and it directs off-node accesses in cluster environments to the correct places (Guo et al., 2010). Concurrent collections (CnC) is a higher-level programming model supported by Habanero by extending the task scheduler.

	Shared Memory	Distributed Memory
Language Integration	Cilk	X10, Chapel
Language Extension	OpenMP	
Library	TBB, MS TPL	Scioto, Satin

Figure 2.9: Categorization of selected task parallel languages and libraries.

Programs in CnC are combinations of input data streams, labels that select ranges of the input, and operations to be applied to those inputs, with the scheduling of operations left to the run time system (Budimlić et al., 2010). Intel has an implementation of CnC for TBB.

Cray's Chapel language is another HPCS language based on the task parallel model (Chamberlain et al., 2007). While our work focuses on OpenMP tasks, the Chapel developers at Cray also use the run time system described in this dissertation to implement support for their language.

Figure 2.9 shows a categorization of some of the task parallel languages and libraries mentioned above based on the architecture they target and the level of language integration. Explicit task parallelism is integrated into the shared memory language Cilk and the PGAS languages X10 and Chapel, which can run on distributed memory. Compiler directives in OpenMP extend C/C++ and Fortran to support task parallelism on shared memory. TBB and MS TPL are libraries for shared memory, while Scioto and Satin are libraries for distributed memory.

Supporting task parallelism has also been investigated outside the programming language community. Scheduler activations, implemented in some versions of NetBSD, support an $m : n$ mapping of user threads to kernel threads in the operating system (Anderson et al., 1991). At the hardware level, the Carbon project at Intel proposed the implementation of local and global task queues in the microprocessor chip itself (Kumar et al., 2007). Another approach, asynchronous direct messaging (ADM), combines software support for tasks with lower complexity hardware acceleration on the chip (Sanchez et al., 2010).

2.5 Performance Analysis Tools

Improvement in task parallel application performance motivates the need for a software ecosystem of tools and analysis methods for the attribution of less than ideal speedup to particular performance issues. In the execution of task parallel applications, what are the actual impacts of overhead costs? To synchronization overheads in particular? Or to poor locality manifested in excessive memory access times? Effective performance analysis and tools enable the interpretation of data to identify performance issues in both applications and the run time system itself. They should assemble and present both thread-level and task-level data in a meaningful and actionable way. Lacking suitable tools and methodologies, users often turn to ad hoc performance analysis techniques.

While toolkits for serial and data parallel programming have matured considerably, limited solutions for task parallel performance analysis exist. A Cilk tool called the Nondeterminator detects race conditions (Feng and Leiserson, 1997), and Cilk Plus calls its race detector Cilkscreen. Cilkview is a tool to determine available parallelism given the work and span of a Cilk Plus program (He et al., 2010). Threadscope allows the display of task graphs, as well as graphs that show memory usage, enabling the identification of bottlenecks and memory leaks (Wheeler and Thain, 2010). An experimental extension to the ompP profiling tool allows timing and performance counter measurement for OpenMP tasks. It runs at user level, allowing portability among run time systems. However, it is unable to access information at the run time level, limiting its utility when tasks are allowed to migrate (Fürlinger and Skinner, 2009). Sun's OpenMP suite allows call stack profiling and task-related event monitoring with some limited visualizations, but those features are only fully supported in the Solaris/SPARC platform (Lin and Mazurov, 2009). Intel VTune and Thread Profiler provide interesting analysis of thread level but not task level timings (Intel Corp., 2011).

Prototype tool support for Cilk is implemented in an experimental extension to the HPCToolkit performance measurement and analysis suite (Tallent and Mellor-Crummey, 2009). It measures thread idle times and overhead costs. It also unwinds the stack to collect so-called logical call path profiles, which generalize call path profiles to enable correct correlation with user code in multithreaded programs. Consider the example task graph shown in Figure 2.10, which represents a snapshot during an execution in Cilk after Thread 1 has stolen task A and Thread 2 has stolen task B. The physical call path on Thread 2 consists solely of tasks executed on Thread 2: $(B \rightarrow E \rightarrow ...)$.

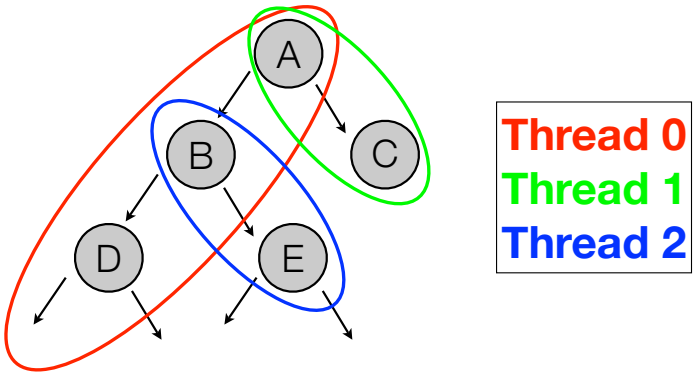


Figure 2.10: Task graph executing on three threads.

Although task A does not execute on Thread 2, it is considered part of the logical path profile for Thread 2 ($A \rightarrow B \rightarrow E \rightarrow ...$) as the parent of task B. Capturing such relationships allows the collection and presentation of performance data despite the migration of tasks by the Cilk work stealing scheduler.

Based on the logical path profiles, HPCToolkit measures and displays a breakdown of total execution time spent by all threads into three categories: work time, idle time, and overhead time. Work time is spent actually performing the computation within the tasks, idleness results from load imbalance, and overhead costs are incurred on task creation, scheduling, and synchronization. They show that overhead time typically can be decreased by coarsening the granularity of tasks and, conversely, idle time can be decreased by using finer-grained tasks. A further extension to HPCToolkit allows analysis of lock contention (Tallent et al., 2010).

CHAPTER 3

UTS: STRESS TEST FOR LOAD BALANCING*

Unbalanced Tree Search (UTS) is a synthetic benchmark designed to measure the ability of diverse parallel architectures and programming environments to support continuous dynamic load balancing. Since task parallel programming delegates the responsibility of scheduling computation to the run time system, UTS evaluates the ability of the run time system to accomplish load balancing efficiently.

3.1 The UTS Benchmark

The UTS problem is to count the number of nodes in an implicitly constructed tree that is parameterized in shape, depth, size, and imbalance (Prins et al., 2003; Olivier et al., 2007). Implicit construction means that each node contains all information necessary to construct its children. Thus, starting from the root, the tree can be traversed in parallel in any order as long as each parent is visited before its children. The imbalance of a tree is the variation in the size of its subtrees. Highly unbalanced trees pose significant challenges for parallel traversal because the work required for different subtrees may vary greatly. Consequently an effective and efficient continuous dynamic load balancing strategy is required to achieve good performance. This requirement is characteristic of the class of applications that UTS models, search and optimization problems that must enumerate a large state space of unknown or unpredictable structure.

The trees are generated using a Galton-Watson process (Harris, 1963), in which the number of children of a node is a random variable with a given distribution. To create deterministic results, each node is described by a 20-byte descriptor. The child node descriptor is obtained by application

*Contents of this chapter previously appeared in preliminary form in *Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing* (Olivier et al., 2007), *Proceedings of the Fifth Workshop on OpenMP* (Olivier and Prins, 2009) and *International Journal of Parallel Programming* (Olivier and Prins, 2010).

of the SHA-1 cryptographic hash (Eastlake and Jones, 2001) on the pair (parent descriptor, child index). The node descriptor also is the random variable used to determine the number of children of the node. Consequently the work in generating a tree with n nodes is n SHA-1 evaluations.

To count the total number of nodes in a tree requires all nodes to be generated; a shortcut is unlikely as it requires the ability to predict a digest's value from an input without executing the SHA-1 algorithm. Success on this task would call into question the cryptographic utility of SHA-1. Carefully validated implementations of SHA-1 exist that ensure that identical trees are generated from the same parameters on different architectures. The value r of the root node is specified as a parameter. Multiple instances of a tree type can be generated by varying this parameter to provide a check on the validity of an implementation.

A *binomial tree* is defined as a tree in which each node below the root has m children with probability q and has no children with probability $1 - q$, where m and q are parameters. When $qm < 1$, this process generates a finite tree with expected size $\frac{1}{1-qm}$. Since all nodes follow the same distribution, the trees generated are self-similar and the distribution of tree sizes and depths follow a power law (Leskovec et al., 2005). The variation of subtree sizes increases dramatically as qm approaches 1, and this variation is the source of the tree's imbalance. A binomial tree is an optimal adversary for load balancing strategies, since there is no advantage to be gained by choosing to move one node over another for load balance: the expected work at all nodes is identical.

A small example binomial tree is shown in Figure 3.1. Most nodes in the tree are in a few particular subtrees, while some nodes very high in the tree have no children at all. This tree has a total of 7079 nodes and a maximum depth of 142 nodes. In our experiments we use a much larger input: a tree of 4.1 million nodes with a depth of more than 1572 nodes.

3.2 UTS Implementations

Consider the example sequential code for the UTS traversal function shown in Figure 3.2. The required inputs, a pointer to the parent of the tree node to be explored and its child number (unique among its siblings), are used to generate the description of the current node using the SHA-1 hash. Then the number of children of the current node is determined by sampling a binomial distribution with probability q at the point defined by the node description. If the result of the sampling is 1 then

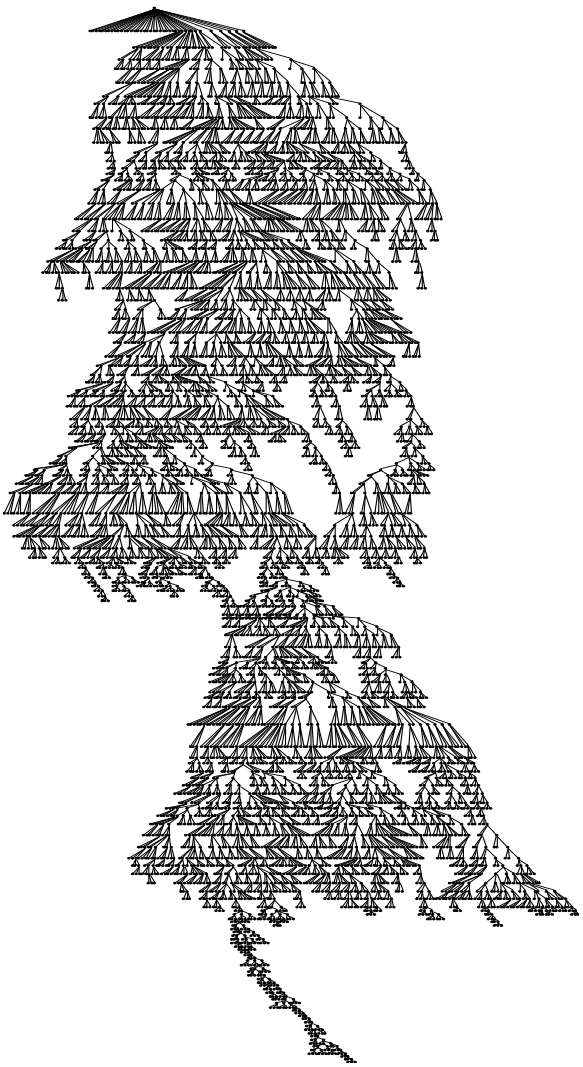


Figure 3.1: Example binomial tree.

the node has m children, but if the result is 0 then it has no children. For each child, the traversal function is called recursively. As each recursive call returns, its result is added to a running total count of nodes. When all children have been explored, the count is returned.

A parallel solution to UTS using OpenMP tasks is shown in Figure 3.3. The code resembles the serial code in structure, with recursive function calls marked as tasks. Since the exploration of each node is a task, the underlying run time system is responsible for performing load balancing as needed. Each task consists of a function that returns the count of nodes in the subtree rooted at its node, recursively creating tasks to count the subtrees rooted at each of its children. In order to accumulate the results correctly, the `partialCount` array is maintained in the function to hold

```
long Generate_and_Traverse(Node* parentNode, int childNumber) {
  Node currentNode = generateID(parentNode, childNumber);
  int numChildren = m with prob q, 0 with prob 1-q
  long nodeCount = 1;

  for (i = 0; i < numChildren; i++)
      nodeCount += Generate_and_Traverse(currentNode, i);
  return nodeCount;
}
```

Figure 3.2: Sequential code for UTS traversal function.

```
long Generate_and_Traverse(Node* parentNode, int childNumber) {
  Node currentNode = generateID(parentNode, childNumber);
  int numChildren = m with prob q, 0 with prob 1-q
  long partialCount[numChildren], nodeCount = 1;

  for (i = 0; i < numChildren; i++) {
    #pragma omp task untied firstprivate(i)
      partialCount[i] = Generate_and_Traverse(currentNode, i);
  }
  #pragma omp taskwait

  for (i = 0; i < numChildren; i++)
    nodeCount += partialCount[i];
  return nodeCount;
}
```

Figure 3.3: UTS using OpenMP tasks.

the result of the subtasks. The task must then suspend and wait for all descendent tasks to complete using a `taskwait` statement before manually combining the results to arrive at the sum to return.

The implementation of UTS in Threading Building Blocks (TBB) requires the definition of a new class that extends the `task` class with the `execute()` function overloaded to perform the work of node generation and traversal, as shown in Figure 3.4. As in the OpenMP tasks implementation, an array of partial sums stores the count returned by each subtree. Other languages have constructs that abstract out this two-step collection of results, e.g., reducers in Cilk++ (Frigo et al., 2009). Figure 3.5 shows a reducer implementation. In this example, synchronization for safe concurrent access to the reducer object is managed by the Cilk++ run time.

```
class Generate_and_Traverse: public task {
public:
  Node *parentNode;
  int childNumber;
  long* const nodeCount;
  Generate_and_Traverse(Node *parentNode_,
    int childNumber_, long* nodeCount_) :
    parentNode(parentNode_), childNumber(childNumber_),
    nodeCount(nodeCount_) {}
task* execute() {
  long partialCount[numChildren];
  parTreeSearchTask* tArr[numChildren];
  Node currentNode = generateID(parentNode, childNumber);
  int numChildren = m with prob q, 0 with prob 1-q
  for (i = 0; i < numChildren; i++) {
     partialCount[i] = 1;
     tArr[i] = new(allocate_child()) Generate_and_Traverse
       (currentNode, childNumber, &partialCount[i]);
     spawn(*tArr[i]);
  }
  set_ref_count(numChildren+1);
  wait_for_all();
  for (i = 0; i < numChildren; i++)
     *nodeCount += partialCount[i];
 }
}
```

Figure 3.4: UTS using Threading Building Blocks.

```
cilk::hyperobject<cilk::reducer_opadd<long> > nodeCount;

long Generate_and_Traverse(Node* parentNode, int childNumber) {
  Node currentNode = generateID(parentNode, childNumber);
  int numChildren = m with prob q, 0 with prob 1-q

  nodeCount()++;
  for (i = 0; i < node->numChildren; i++)
      cilk_spawn traverse(node->child[i]);
}
```

Figure 3.5: UTS using Cilk++.

Compiler	ICC 11.1	Sun C 5.11	PGI 10.4	GCC 4.4.4
Execution Time (Sec.)	1.37	1.47	2.27	1.61
Rate (M. Nodes / Sec.)	3.00	2.79	1.81	2.56

Table 3.1: Sequential performance on UTS.

3.3 Performance Evaluation

Since UTS is designed as a stress test for load balancing, we hypothesize that a task parallel run time that fails to accomplish load balancing efficiently among threads in an execution will fail to achieve near-linear speedup on UTS. To test this hypothesis, we evaluate UTS performance on a 24-core Dell PowerEdge M905 shared memory system. The system consists of four six-core AMD Opteron 8425 HE processors running at 2.1 Ghz. 64KB 2-way associative L1 data cache, 64KB 2-way associative instruction cache, and 512MB 16-way associative L2 cache per core. The six cores on each chip share a 6MB 48-way associative L3 cache. The processors are connected by the AMD HyperTransport interconnect, and a total of 32GB of DRAM is distributed among the processors (8GB linked to each chip's memory controller).

We compare the ICC 11.1 compiler from Intel, the Sun Ceres C 5.11 compiler from Oracle SolarisStudio, the GCC 4.4.4 compiler from GNU, and the PGI 10.4 compiler. For comparison with TBB and Cilk++, we use the Cilk Arts distribution of Cilk++ 1.0 (based on GCC) and Intel Thread Building Blocks 2.2 (compiled and used with ICC 11.1). The `-O3` option is always used, and with the Intel compiler we also use `-ipo` to enable interprocedural optimization, which in the other compilers is implied by `-O3`. Unless otherwise noted, reported results represent the lowest time out of 10 trials.

In our evaluation, we consider a tree with input parameters of root branching factor $b_0 = 2000$, non-root branching factor $m = 8$, probability of a node generating children $q = 0.124875$, and random seed $r = 42$. Due to the determinism of the SHA-1 hash, these parameters generate the same tree each time a correct UTS implementation is executed. Since $q \cdot m = 0.999 < 1.0$, the tree is guarantied to be finite. Since qm is close to 1.0, the tree exhibits severe imbalance. A full traversal shows that the resulting tree has 4.1 million nodes total and a maximum depth of 1572 nodes, and that over 99% of the nodes are descended from only one particular child of the root node.

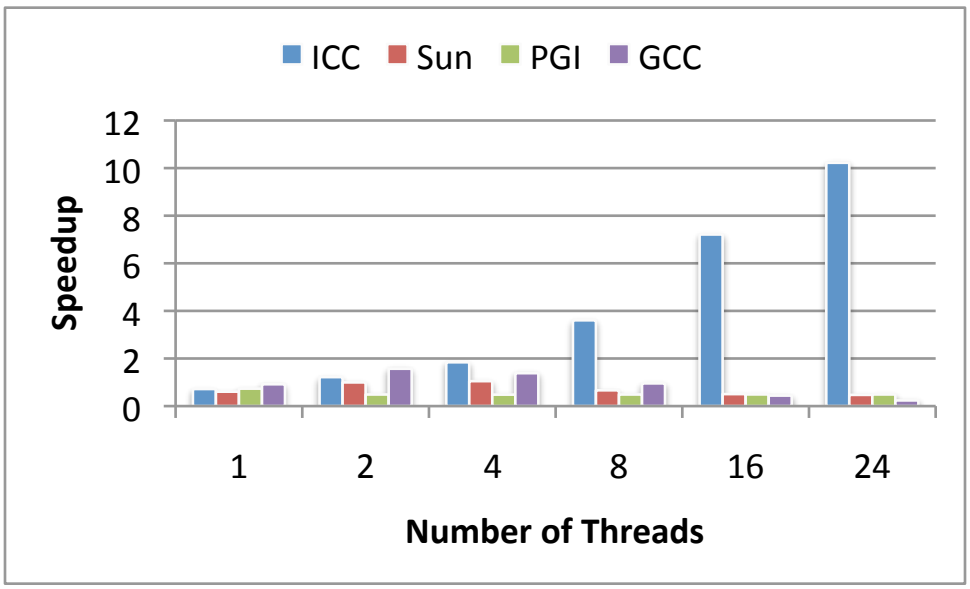


Figure 3.6: Parallel speedup of UTS using OpenMP Tasks.

Table 3.1 shows the sequential performance of the four compilers using the implementation shown in Figure 3.2. The first row gives the execution time in seconds, and the second row gives the execution rate expressed as millions of nodes explored per second. The executable produced by the Intel compiler achieves the fastest time, followed by those from Sun, GCC, and PGI

Figure 3.6 shows the speedup gained on the task parallel implementations using OpenMP 3.0, as measured against the sequential performance data given in Table 3.1. Only the Intel run time system delivers any significant speedup, and even it fails to achieve 45% efficiency. These results raise two questions: Is it even possible to achieve near-linear speedup on the UTS problem? What is the Intel run time system doing differently than the Sun, PGI, and GCC?

In answer to the first question, we compare the performance of the Intel OpenMP run time with TBB, Cilk++, and a custom work stealing implementation that we constructed on top of OpenMP threads and complied with ICC. The results are shown in Figure 3.7. TBB consistently outperforms the ICC OpenMP run time. Cilk++ fails to complete with less than 16 threads because the Cilk++ run time does not allow a large enough task queue per thread to support the number of active tasks needed for UTS. With 16 threads, TBB outperforms ICC OpenMP, but it falls short with 24 threads. The custom work stealing implementation on OpenMP threads performs the best, reaching 21X

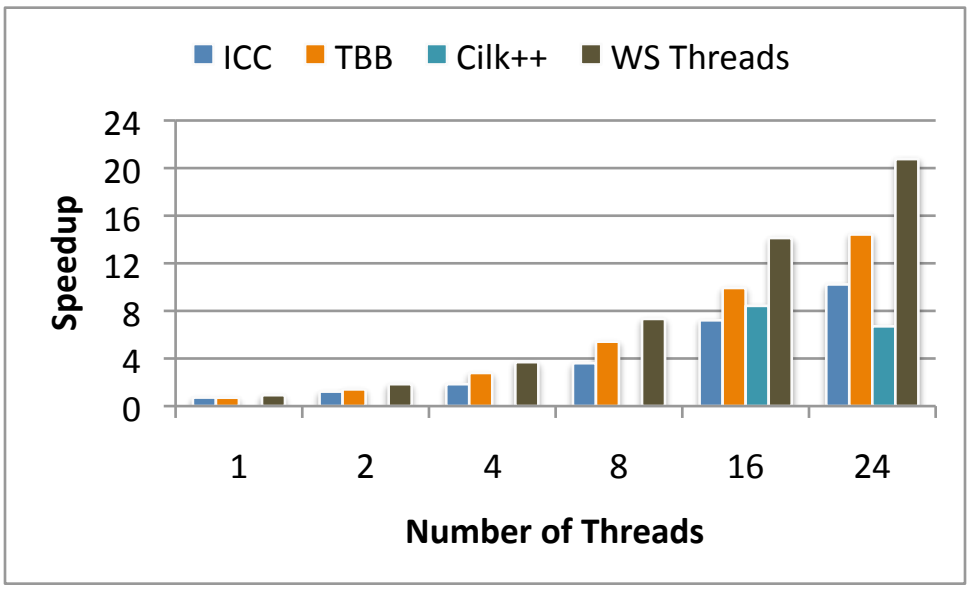


Figure 3.7: Speedup of UTS: OpenMP versus other task parallel languages and libraries.

speedup with 24 threads. Thus, high performance on UTS can be acheived. We will explain later how the custom implementation supports this level of performance.

Meanwhile, we return to our second question: Why is ICC the only run time to achieve even a modest parallel speedup? We hypothesized that UTS performance requires efficient load balancing. The work of each task in UTS is the same, one SHA-1 hash calculation. Thus, in an execution that successfully balances load, each thread should execute the same number of tasks, or equivalently, explore the same number of nodes in the tree. We instrumented UTS to measure this, and the results are shown in Figure 3.8. For each combination of compiler and number of threads used, we ran 10 trials. Each point displayed represents the number of tasks executed (and number of nodes visited) on a particular thread. Thus, in the ICC graph on the upper left-hand side, each of the 10 blue diamonds represents the one thread in each execution that executed all 4.1million tasks, visiting all tree nodes. Each of the 20 red squares represents one of the two threads in each execution and the roughly 2 million tasks executed by each thread. Likewise, there are 40 green triangles representing the threads in the 10 executions using four threads, and so on. In a perfectly load balanced execution, all threads would execute the same number of tasks, $tree\_size/number\_of\_threads$, and all points of the same color and shape would form a single horizontal line. The graph for ICC, which achieved the best speedup among the OpenMP run times, comes closest to this ideal load balance. The Sun run time

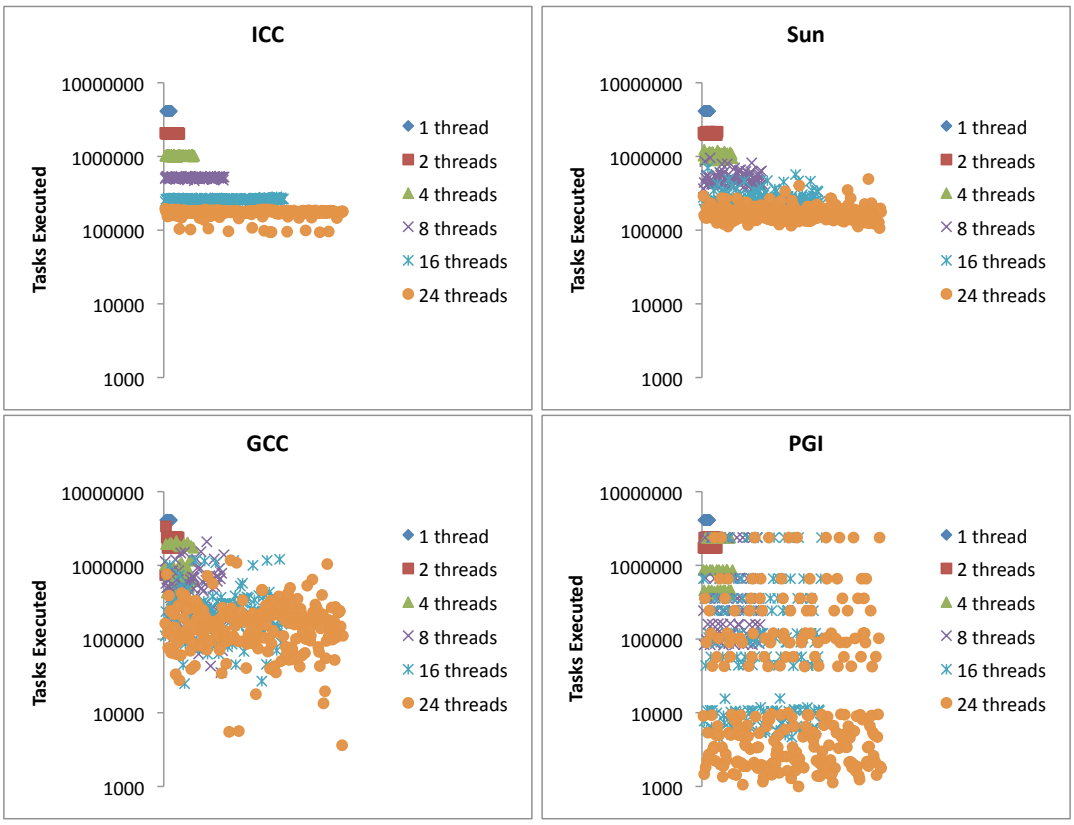


Figure 3.8: Tasks executed per thread during UTS execution.

shows more uneven distribution of tasks, and GCC shows even more. The PGI run time shows a fixed pattern of imbalance. Note that these results are plotted on a logarithmic scale.

When some threads perform millions of SHA-1 executions while others perform less than ten thousand, load balancing is either nonexistent, ineffective, or inefficient. We further instrumented UTS to record the instances in which a task executed on a different thread than its parent, or resumed execution on a different thread after a suspension. Although migration of suspended tasks is allowed for untied tasks in OpenMP, we did not observe any such migrations. We did observe many instances of tasks executing on different threads than their parents, indicating load balancing operations made by the run time scheduler. Figure 3.9 shows the average number of tasks moved per thread, again on a logarithmic scale. The order of magnitude of the moved tasks per thread for the ICC run time is relatively consistent for ICC, Sun, and GCC. Sun performs fewer load balancing operations than ICC. GCC performs an order of magnitude fewer load balancing operations than ICC. PGI performs

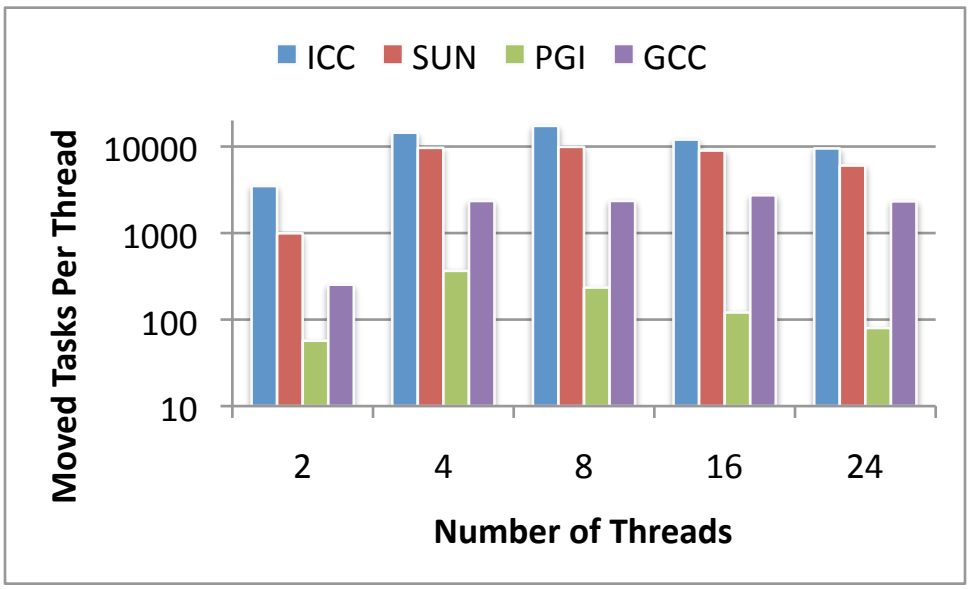


Figure 3.9: Average moved tasks per thread during UTS execution.

two orders of magnitude fewer load balancing operations than ICC. By inspection, these results correspond roughly with the imbalance graphs in Figure 3.8.

Another factor that contributes to suboptimal performance is time spent on overhead costs in the run time system that arise from task creation, synchronization, and load balancing operations. The same load balancing operations required to succeed on the UTS problem divert time from the work of the computation. These costs could be reduced partially by executing some tasks in place, but at the expense of load balance. The `if` clause on the OpenMP `task` directive forces a task to execute in place. We modified UTS to execute a fraction of the tasks in place using ICC. Figure 3.10 shows speedup results for executions using 24 threads as a function of percent tasks executed in-place. The results show a negative impact of in-place execution. Executing even a small fraction of tasks in-place limits the run time's capability to mitigate the extreme imbalance inherent in the UTS problem.

Recall from Figure 3.7 that much better performance was achieved using our custom work stealing implementation of UTS built atop OpenMP threads. The approach to controlling overhead costs in that implementation is to aggregate tasks during load balancing. Instead of stealing a single task at a time as in Cilk, an idle thread steals k tasks, where k is a tunable parameter. Figure 3.11 shows speedup using this implementation using 24 threads for various choices of k. When k is small, too many steal operations occur and overhead costs are high. When k is large, too few successful

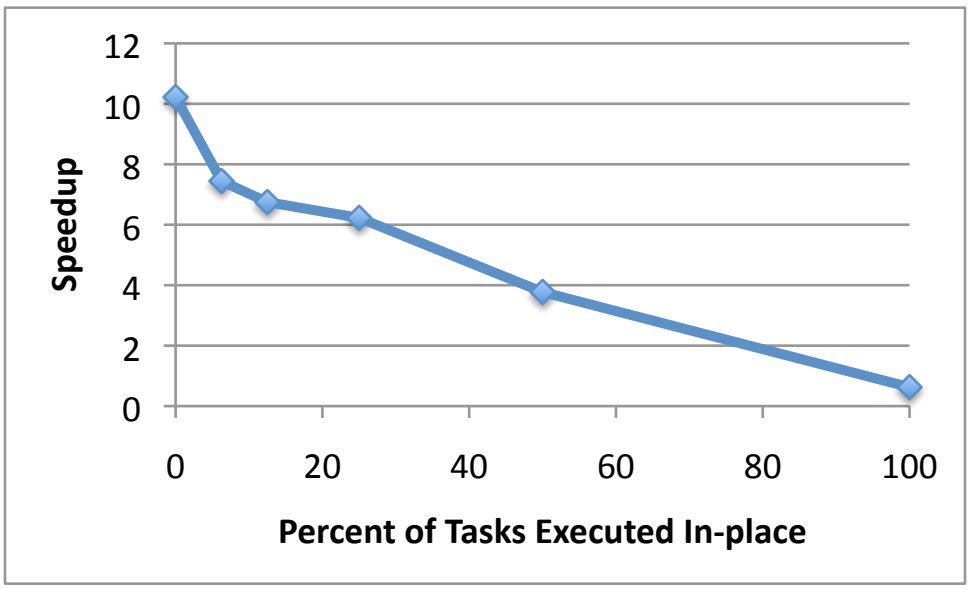


Figure 3.10: UTS speedup using OpenMP tasks as a function of tasks executed in-place.

Figure 3.11: UTS speedup (custom implementation) as a function of tasks stolen per steal.

steals occur to maintain load balance. Peak performance occurs in the middle of the range. The center and range of this "sweet spot" is a function of processor speed, memory access time, and the implementation choices of the stealing protocol used. A peak range of k near the vertical axis indicates low memory access time and/or low overhead costs for stealing, while a peak range of k

further away indicates processor speed that outpaces memory access time and/or an inefficient stealing protocol.

3.4 Summary

Explicit task parallelism provided in OpenMP 3.0 enables the simple expression of unbalanced applications as can be seen from the simplicity and clarity of the task parallel UTS implementations. However, the UTS benchmark exposes deficiencies in load balancing capabilities of run time systems for task parallelism. Among production OpenMP run time systems, only Intel's achieves any significant speedup, although less than half of ideal speedup. UTS exhibits a fundamental trade-off between maintaining load balance and limiting overhead costs, but we have shown that our custom work stealing implementation succeeds by aggregating tasks during steal operations. We shall examine these ideas further in the chapters that follow.

CHAPTER 4

CATEGORIZING EXECUTION TIME

Ideally, parallel execution of a task parallel application achieves linear speedup, i.e., the execution time with p threads is T_S/P where T_S is the sequential execution time of a serial equivalent of the application. UTS is designed to be a stress test, but more mundane applications also frequently fail to achieve linear or even near-linear speedup. According to *Amdahl's Law*, the speedup of a program using multiple processors is limited by the time needed for the sequential fraction of the program (Amdahl, 1967). Indeed, any period of the execution time in which any thread is not engaged in the computational work of the application, at an equal or faster rate as sequential execution of the same computational work, limits parallel speedup. A thread may contribute to performance loss for several reasons:

- *Overhead:* The thread is not be engaged in computational work because it is executing instructions in the run time system, such as task creation and bookkeeping for task synchronization.
- *Idleness*: The thread is idle due to inherent lack of available parallelism in the application or failure of the scheduler to supply ready tasks for all threads.
- *Work Time Inflation:* The thread is executing some sequence of instructions in the computational work at a slower rate than a sequential execution of the same instructions. The data needed to supply the operands to those instructions is a shared resource in multithreaded execution, and a common cause of work time inflation is increased latency to access that data.

Performance analysis methods that explain how these three factors contribute to the performance gap between linear speedup and observed speedup in task parallel programs can inform improvements in application development, scheduler design, run time system implementation, and deployment of hardware resources.

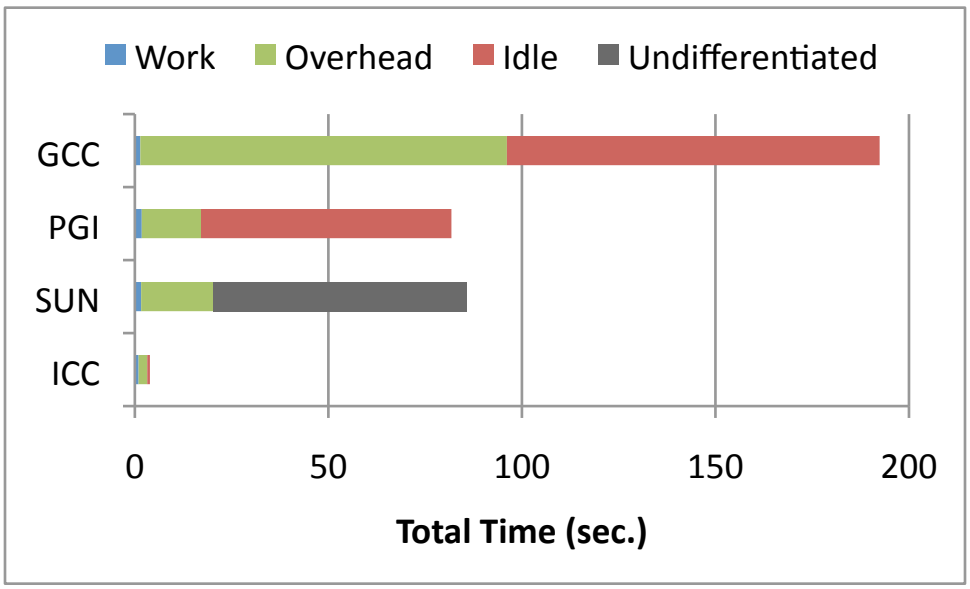


Figure 4.1: Total time over all threads on a 24-thread UTS execution.

Our *Component Model* of execution time for task parallel applications comprises *work time*, *overhead time*, and *idle time*. Together these times account for the total time spent by all threads. If the execution time is T_P using P threads, the total time is

$$P \cdot T_P = \sum_{i=0}^{P-1} work(i) + \sum_{i=0}^{P-1} ovh(i) + \sum_{i=0}^{P-1} idle(i) \tag{4.1}$$

where $work(i)$ is the time spent by thread i on the instructions of the application's computational work, $ovh(i)$ is the time spent by thread i on overhead costs for operations in the run time system, and $idle(i)$ is the time spent idle by thread i. Locking can be a source of overhead costs, idleness, or both, depending on the level of contention between tasks competing for lock access and the implementation of locking operations.

4.1 UTS Revisited

We can apply the component-based analysis to the UTS problem. Using HPCToolkit (Adhianto et al., 2010), we recorded the amount of time spent in the various functions and libraries during execution. Although the commercial run time systems are closed source, the function names in their OpenMP libraries typically denote their function clearly. Figure 4.1 shows a comparison of

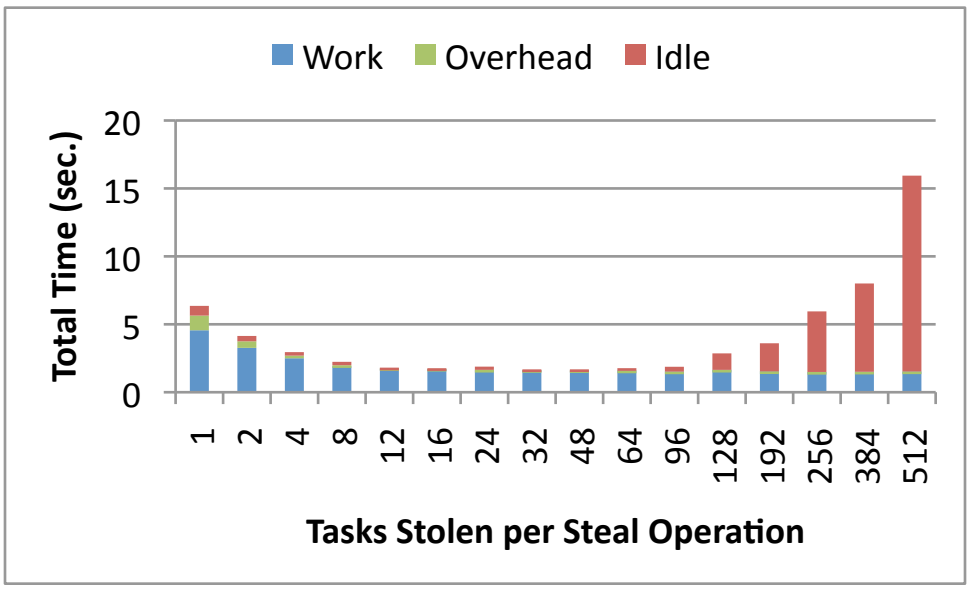


Figure 4.2: Total time over all threads on UTS (custom implementation) as a function of tasks stolen per steal.

the distribution of total time over all 24 threads of an execution categorized as work, overhead, and idle time. In an execution with near-linear speedup, the contribution of overhead and idle time must be negligible. Instead, overhead and idle time dwarf the time spent doing the actual work of the computation. GCC is the least efficient. Open source reveals its unscalable implementation: a single centralized task queue with global metadata updated on a per-task basis. Overhead costs in the PGI run time are lower, but idle time is nearly as bad. Time in the Sun run time system is dominated by opaque calls to pthreads-based functions, labelled on the chart as "undifferentiated." They may include *wait* and *yield* operations that constitute idle time, or synchronization, i.e., *mutex* operations that constitute overhead time. The run time calls that are transparent are categorizable as overhead. Recall that the Intel run time is the only run time that delivers any significant speedup and gives reasonable load balance, manifest here in the comparatively small measured idle time. However, overhead time dominates, accounting for the 14X lost speedup that we observed in Figure 3.6.

We can also break down the time using our custom work stealing implementation. As we saw in Figure 3.11, it achieves speedup of 21X on 24 threads by stealing multiple tasks per steal operation. Figure 4.2 shows the distribution of time over all threads for executions with various choices of the k parameter (tasks stolen per steal operation). As expected, overhead times are greatest when k is

small, but work time is also substantially greater that the observed work time of the executions with larger k. Each steal operation disturbs the locality of the computation, so that stealing one task at a time results in more cache misses. We shall study this issue in greater detail later. As k increases beyond 64, stealing is not fine-grained enough to maintain load balance. Total time is minimal in the ideal range ($k = 24, 32, 64$), as are the contributions of each individual component–work, overhead, and idle time.

4.2 Analysis of BOTS

We now apply our model to applications from the Barcelona OpenMP Tasks Suite (BOTS), version 1.1, available online (Duran and Teruel, 2011). The suite comprises a set of task parallel applications from various domains with varying computational characteristics (Duran et al., 2009). Our experiments used the following benchmark components and inputs:

- *Alignment*: Aligns sequences of proteins using dynamic programming (100 sequences)
- *Fib*: Computes the *n*th Fibonacci number using brute-force recursion ($n = 50$)
- *Health*: Simulates a national health care system over a series of timesteps (144 cities)
- *NQueens*: Finds solutions of the *n*-queens problem using backtrack search ($n = 14$)
- *Sort*: Sorts a vector using parallel mergesort transitioning to sequential quicksort and insertion sort (128M integers)
- *SparseLU*: Computes the LU factorization of a sparse matrix (10000×10000 matrix, 100×100 submatrix blocks)
- *Strassen*: Computes a dense matrix multiply using Strassen's method (8192 x 8192 matrix)

For the *Fib*, *Health*, and *NQueens* benchmarks, the default manual cut-off configurations provided in BOTS are enabled to prune the generation of tasks manually below a prescribed point in the task hierarchy. Because these applications are much more balanced than UTS, this technique is applicable. For *Sort*, cutoffs are set to transition at 32K integers from parallel mergesort to sequential quicksort and from parallel merge tasks to sequential merge calls. For *Strassen*, the cut-off giving the best performance for each implementation is used.

```
#pragma omp single
   for (si = 0; si < nseqs; si++)
      for (sj = i+1; sj < nseqs; sj++)
         #pragma omp task firstprivate(si, sj)
            compare(seq[si], seq[sj]);
```

```
#pragma omp for schedule(dynamic)
   for (si = 0; si < nseqs; si++)
      for (sj = si+1; sj < nseqs; sj++)
         #pragma omp task firstprivate(si, sj)
            compare(seq[si], seq[sj]);
```

Figure 4.3: Simplified code for the two versions of *Alignment*.

For both the *Alignment* and *SparseLU* benchmarks, BOTS provides two different parallel implementation. Simplified code given in Figure 4.3 illustrates the distinction between the two versions of *Alignment*. In the first (*Alignment*-`single`) the loop nest that generates the tasks is executed sequentially by a single thread. This version creates only task parallelism. In the second (*Alignment*-`for`) the outer loop is executed in parallel, creating both loop-level parallelism and task parallelism. Likewise, the two versions of *SparseLU* are one in which tasks are generated within single-threaded loop executions and another in which tasks are generated within parallel loop executions.

The hardware test system for the following experiments is a Dell PowerEdge M910 quad-socket blade with four Intel x7550 2.0GHz 8-core Nehalem-EX processors installed for a total of 32 cores. The processors are fully connected using Intel QuickPath Interconnect (QPI) links. Each processor has an 18MB shared L3 cache and each core has a private 256KB L2 cache as well as 32KB L1 data and instruction caches. The blade has 64 dual-rank 2GB DDR3 memory sticks (16 per processor chip) for a total of 132GB. It runs CentOS Linux with a 2.6.35 kernel. Although the x7550 processor supports HyperThreading (Intel's simultaneous multithreading technology), we pinned only one thread to each physical core for our experiments.

All executables using the Intel run time were compiled with ICC 11.1 and -O2 -xHost -ipo optimization. Executables using the GCC run time was compiled with GCC 4.4.4 with -g and -O2 optimization. Unless otherwise stated, reported results are the best of ten executions.

Table 4.1 shows sequential execution times for BOTS. For the most part, the sequential times are reasonably close between executables compiled with ICC and GCC. Results of parallel executions using 32 threads are shown in Figure 4.4. The ICC executables with the Intel run time achieves

Run Time	Alignment	Fib	Health	NQueens	Sort	SparseLU	Strassen
ICC	28.33	100.4	15.07	49.35	20.14	117.3	169.3
GCC	28.06	83.46	15.31	45.24	19.83	119.7	162.7

Table 4.1: Sequential execution times (in seconds) on BOTS.

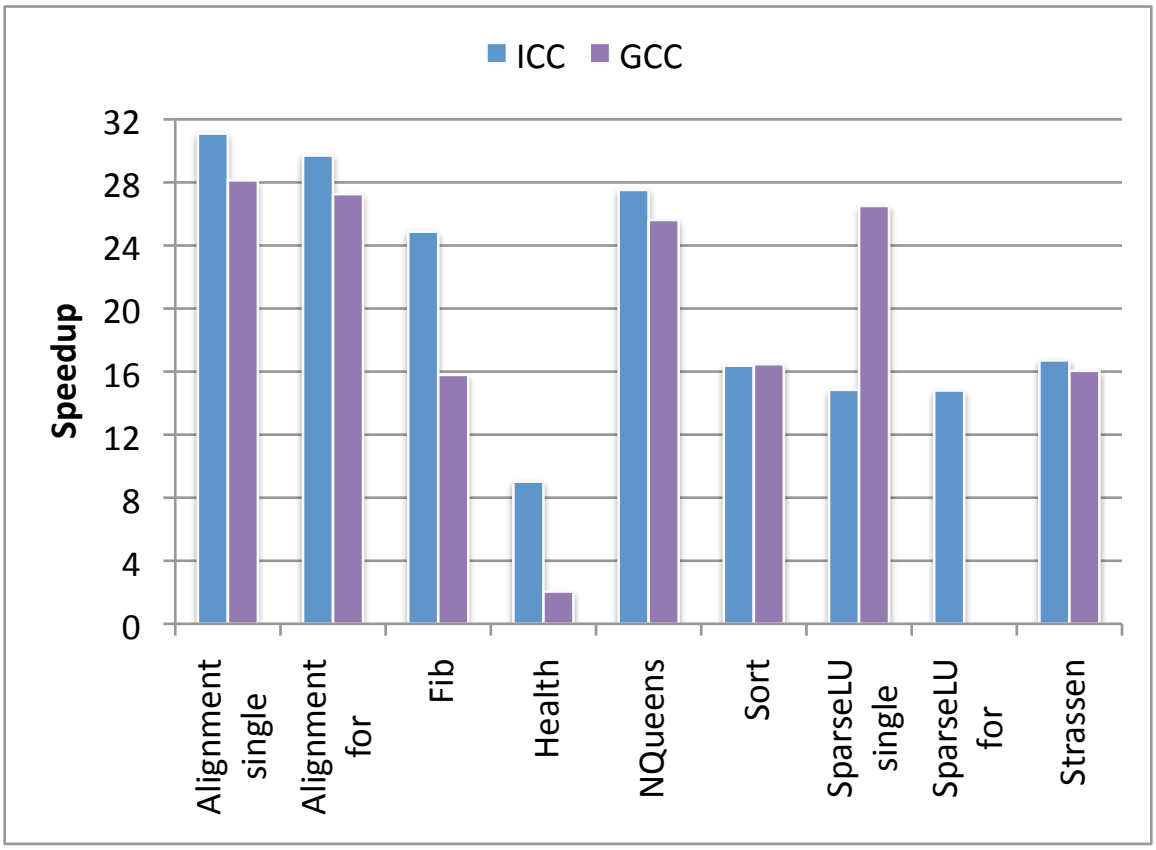


Figure 4.4: Speedup on BOTS using ICC and GCC with 32 threads.

24.8X-31.1X speedup on four benchmarks (both *Alignment* versions, *Fib*, and *NQueens*. On *Sort*, the two *SparseLU* versions, and Strassen, they reach just 14.8X-16.7X speedup. On the *Health* benchmark, speedup is only 9X. GCC substantially outperforms ICC only on the `single` version of *SparseLU*, where it achieves 26.5X speedup. On *SparseLU*-`for`, the GCC OpenMP runs were stopped after exceeding the sequential time; thus data is not reported. Speedup results are close between ICC and GCC on *Sort* and *Strassen*, but GCC speedup lags on the remaining benchmarks. Speedup is particularly poor on *Fib* (16.8X), and absolutely dismal on *Health* (2X).

We again consider the time spent by all threads on work, overhead, and idle. Figure 4.5 shows the results of executions using ICC with 32 threads. In most of the benchmarks, overhead and idle times

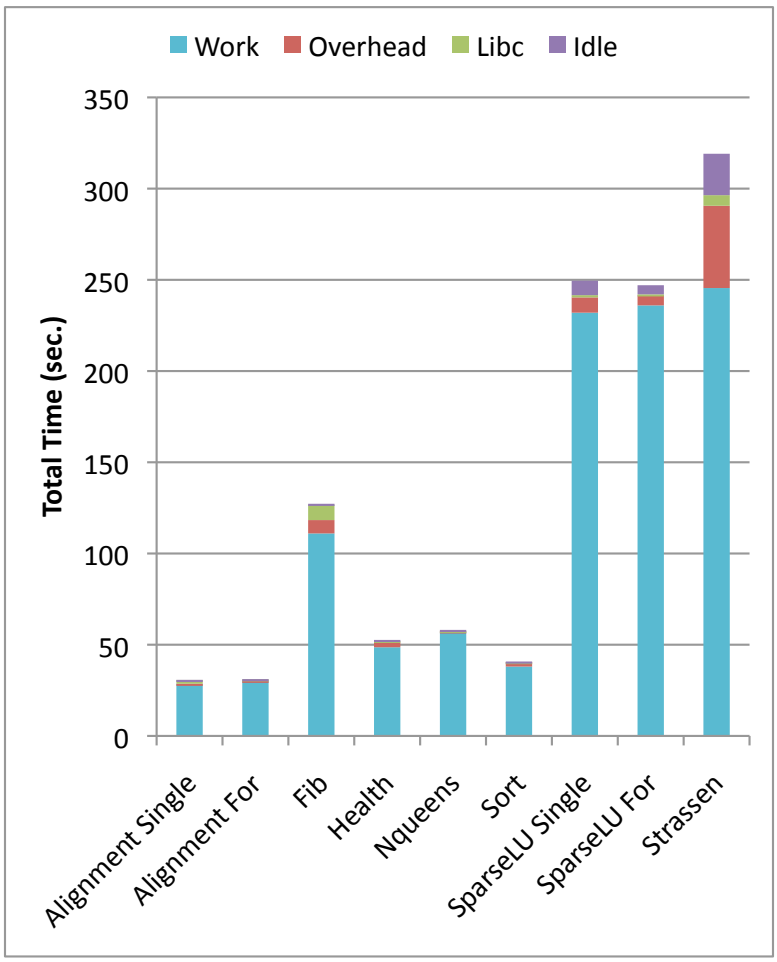


Figure 4.5: Total time over all threads on BOTS using ICC with 32 threads.

are negligible compared to work time. Time spent on calls to libc is shown separately. Non-work time is significant but not dominant for *Fib* and both versions of *SparseLU*. Only in the *Strassen* benchmark do overhead and idle times contribute a very large portion of the total time. Figure 4.5 gives the results of the same experiments using GCC. In addition to *Strassen*, the GCC run time also incurs large idle time on *Fib* and *Health*, and overhead time is also quite high on *Sort*.

A comparison of the speedup results from Figure 4.4 and the time breakdowns from Figures 4.5 and 4.6 partially validates the ability of the model to diagnose the source of poor speedup. Good speedup on the two versions of *Alignment* is consistent with low overhead and idle times with both ICC and GCC. Less impressive speedup on *Fib* is roughly consistent with non-negligible overhead and libc time in ICC and excessive idle times in GCC. Large overhead time and massive idle times help to explain the particularly poor speedup on *Health* with GCC, but disappointing speedup using

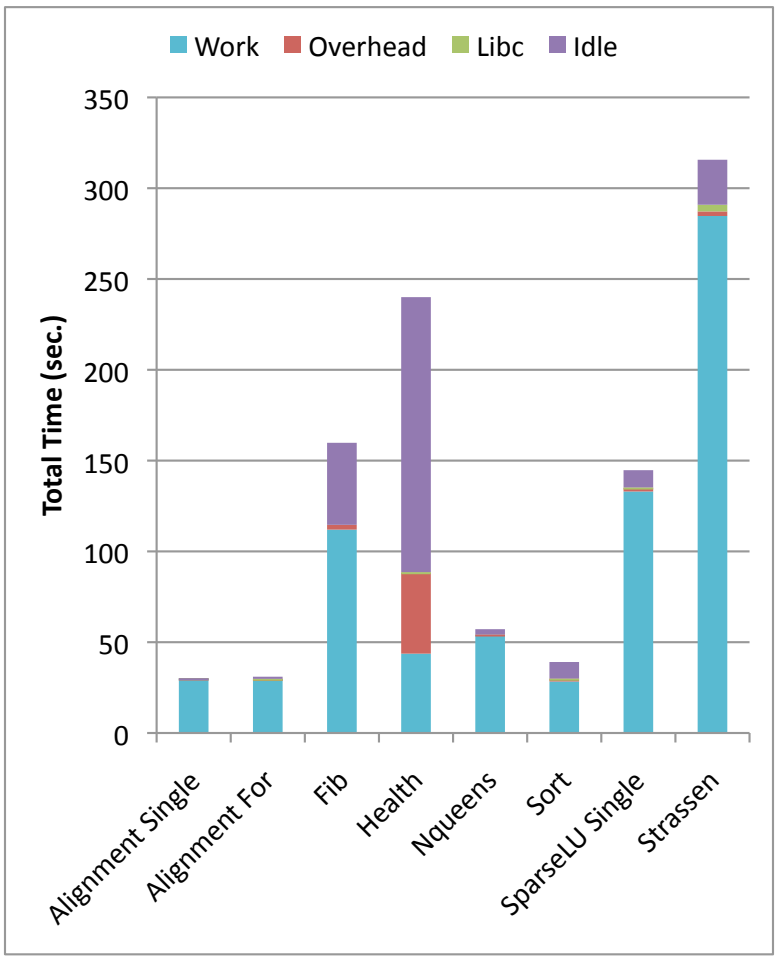


Figure 4.6: Total time over all threads on BOTS using GCC with 32 threads.

ICC is not evident in the time breakdown. The same is true of the remaining benchmarks, for both GCC and ICC. As it stands, the model clearly fails to pinpoint the factors responsible for the lost speedup.

4.3 Work Time Inflation

A quick glance at Table 4.1 reveals that the sequential execution times for the *Health* benchmark were 15.1 and 15.3 seconds using ICC and GCC respectively. In timing breakdowns of 32 thread executions shown in Figures 4.5 and 4.6, the work times on the same benchmark are over 40 seconds. However, the amount of computational work is the same. We call this phenomenon *work time inflation*.

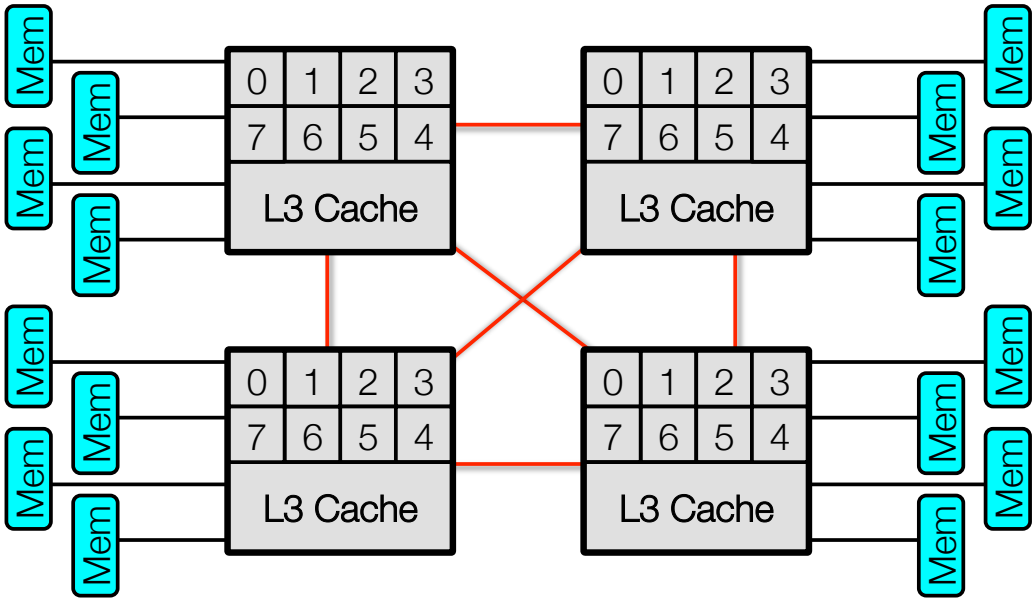


Figure 4.7: Topology of the four-socket Intel Nehalem-EX system.

Work-time inflation can result from the increasing demands placed on the memory system and caches by concurrent memory references. Non-uniform memory access times reflect the physical reality of memory organization; some memory devices are closer to some referencing processors than to others. For example, the four-socket Intel system we use for the BOTS experiments has a topology as shown in Figure 4.7. Data in memory linked directly to one of the four chips is more quickly accessible by that chip than data in memory linked to the other chips in the system. Besides fundamental latency constraints, memory device and processor to memory interconnection network bandwidth constrain the number of concurrent references that can be sustained (Mandal et al., 2010). In addition, substantial memory subsystem overheads and latency costs are incurred to maintain coherence of memory references in the presence of a parallel cache hierarchy.

We now refine our model to reflect the contribution of work time inflation by splitting work time into two components, $work_{seq}$ and $work_{inf}$. The amount of time to complete the work in a sequential execution of a serial equivalent of the program is $work_{seq}$. Additional time spent on the computational work during parallel executions is $work_{inf}$. Thus we have the refined equation for total time across all threads:

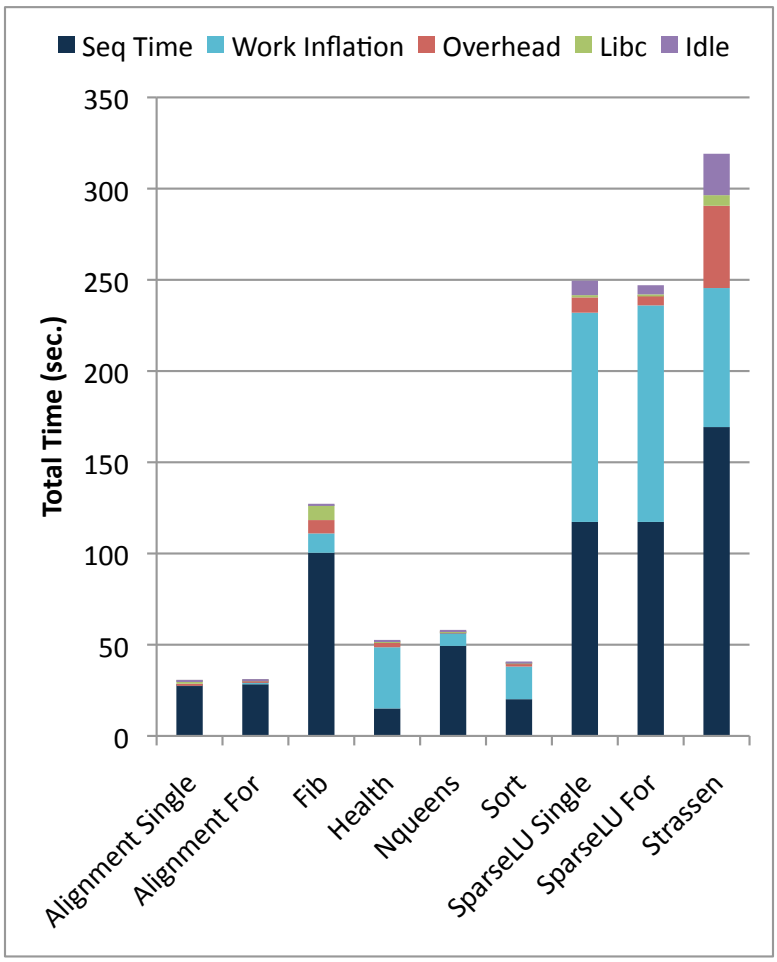


Figure 4.8: Total time on BOTS using ICC: Work time inflation.

$$P \cdot T_P = \sum_{i=0}^{P-1} work_{seq}(i) + \sum_{i=0}^{P-1} work_{inf}(i) + \sum_{i=0}^{P-1} ovh(i) + \sum_{i=0}^{P-1} idle(i) \tag{4.2}$$

Figures 4.8 and 4.9 give a clearer picture of lost performance using the refined model for ICC and GCC respectively. The results are from the same executions as Figures 4.5 and 4.6, but with the sequential work time and work time inflation shown separately. With both run times, only the two versions of *Alignment* have negligible work time inflation. With ICC, we see significant work time inflation in *Health*, *Sort*, both versions of *SparseLU*, and *Strassen*. With GCC, work time inflation is most extreme in *Health*, *Sort*, and *Strassen*.

When we consider overheads, idle time, and work time inflation, we account for the entirety of the lost time that is manifest in poor speedup and can work to improve it. Take for example the

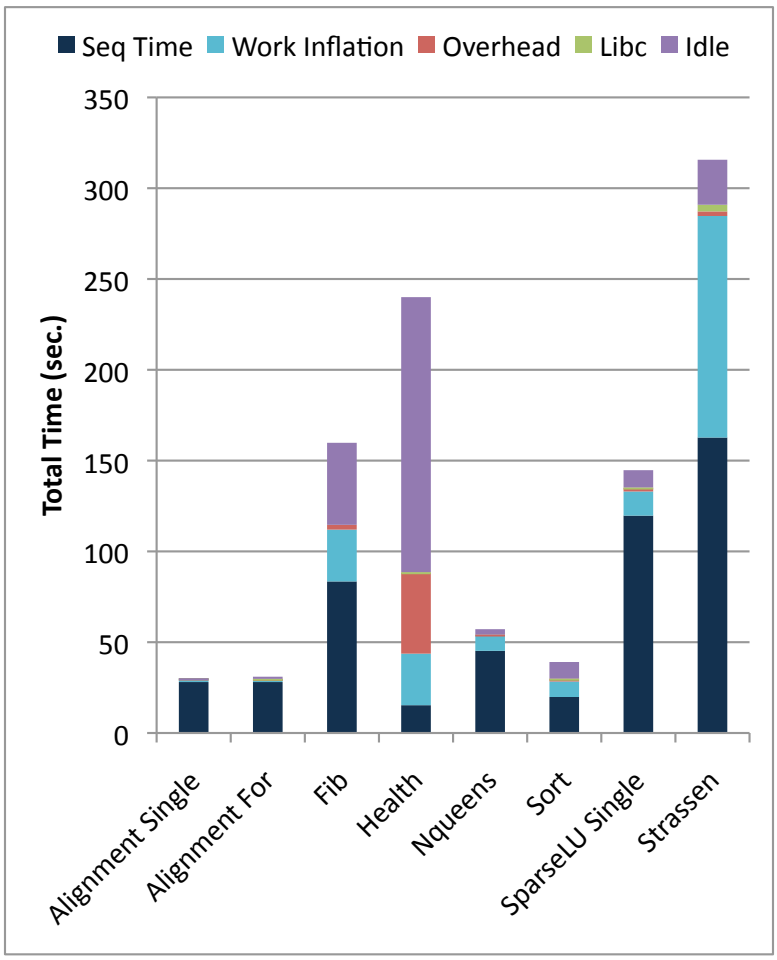


Figure 4.9: Total time on BOTS using GCC: Work time inflation.

Health benchmark, on which both run times perform most poorly. Threads in the GCC run time spend an enormous amount of time idle and on overheads during the execution of this benchmark. With both run times, the work time inflation in *Health* is even greater than the sequential work time, meaning that in 32 thread execution the computational work takes up over twice the amount of time. Clearly, improvement on *Health* requires the elimination of work time inflation.

We obtained these results on a quadsocket Intel platform with the Quick Path Interconnect (QPI). Many other Non-Uniform Memory Access (NUMA) systems exhibit similar pronounced differences in latencies to local and remote memory. IBM reports reports longer latencies to access remote data in POWER7*i* systems (Funk and Peterson, 2010). In addition to the latency inherent in such remote link traversals, contention on links and saturation of the buffers that serve them can also impact performance negatively. AMD reports completion times on AMD memory microbenchmarks with

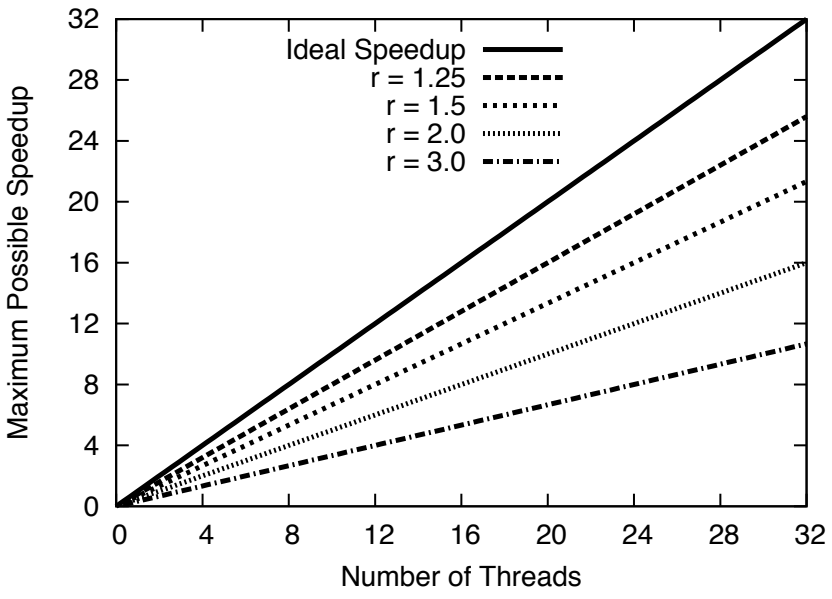


Figure 4.10: Speedup limitations due to work time inflation.

remote access cross-traffic that are frequently double the completion times when only local accesses occur (AMD Inc., 2006).

As we have observed in the benchmarks, work time inflation limits the maximum potential parallel speedup of an application. Let r be the ratio of the total work time of a p-processor parallel execution to the sequential work time. The lowest potential execution time is r/p and the highest potential parallel speedup is p/r. For executions using 32 threads, the health simulation yields $r > 3$. Figure 4.10 shows the maximum potential speedup for various levels of work inflation ($r = 1.25$ to 3.0). Even with a relatively small NUMA impact of r equal to 1.25, we observe a maximum speedup of under 26X with 32 threads. Actual speedups will be even lower given non-zero overheads and potential load imbalance.

4.4 Available Parallelism

The Cilk Plus analysis tool, Cilkview (He et al., 2010), estimates upper and lower bounds for speedup based on the overall available parallelism in the application and expected run time overhead costs. In the OpenMP executions of UTS and BOTS, we have observed that overhead costs vary not only by run time system but also depending on the application. Available parallelism changes

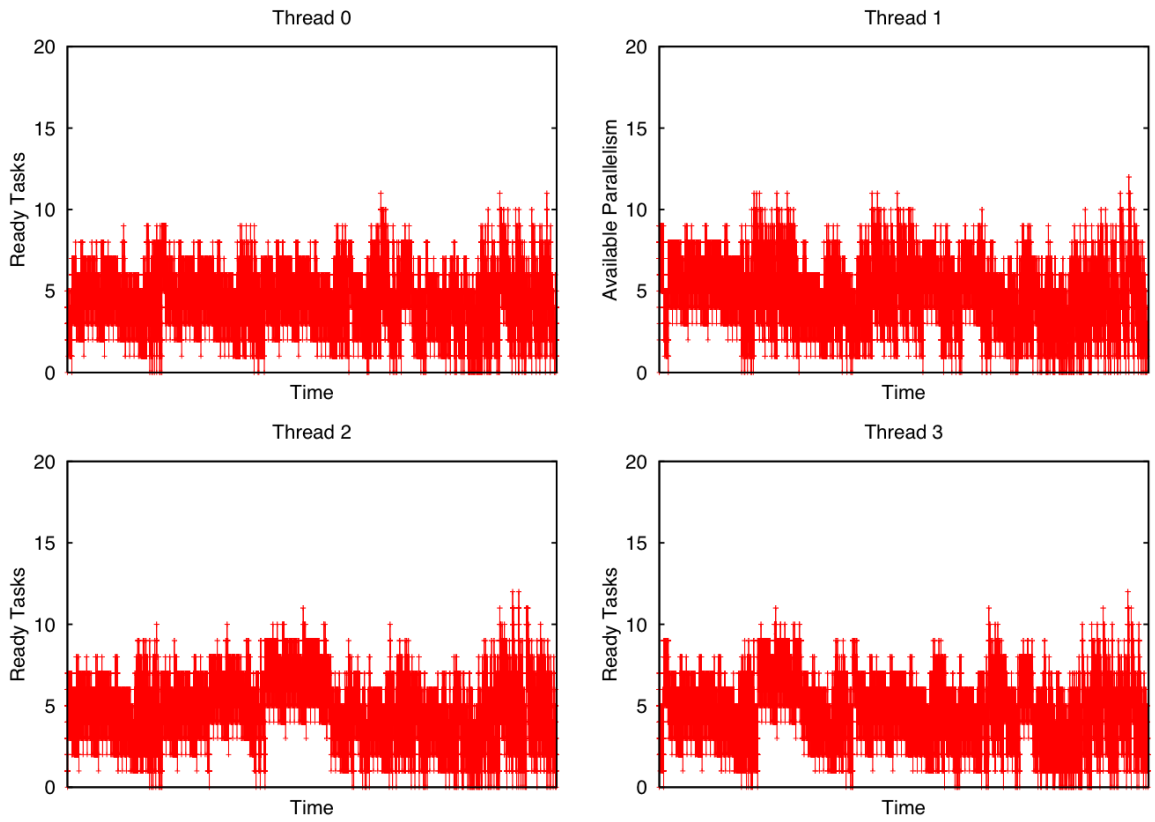


Figure 4.11: Queue length during execution of *Sort* using a work-first schedule.

throughout the execution of a task parallel program and also depends on the choice of scheduling policy. Figures 4.11 and 4.12 show queue length over time during executions of BOTS *Sort* using our custom work stealing implementation with four threads using work-first and help-first schedules, respectively. Higher queue lengths in the executions with help-first scheduling indicate greater available parallelism. The regular four-fold pattern shown in the graphs corresponds to the division of work into quarters in each recursive call to *Sort*, and the amount of parallelism varies as tasks are generated and retired.

4.5 Summary

The goal of ideal speedup for task parallel programs can be evasive, as we have seen with UTS and BOTS. Our performance model identifies the causes of poor speedup by categorizing the time spent by all threads in the execution into work, overhead, and idle times. We further divide work

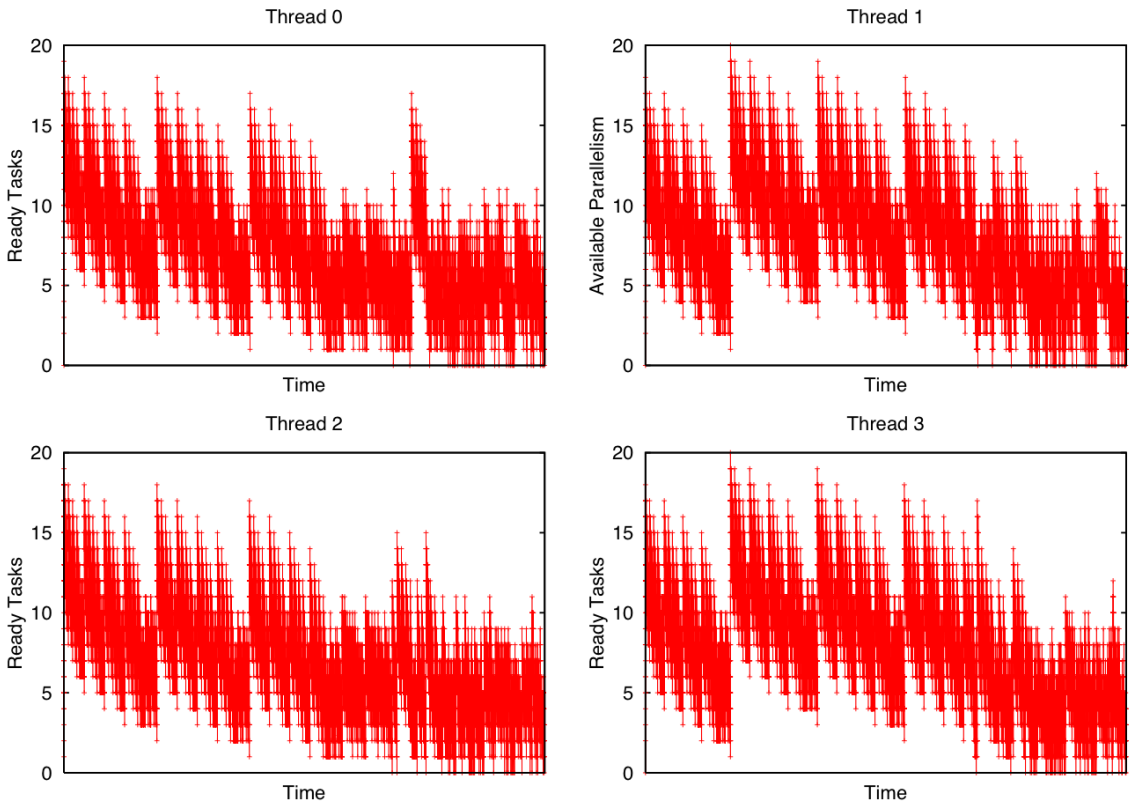


Figure 4.12: Queue length during execution of *Sort* using a help-first schedule.

time into two parts, equivalent sequential time and work time inflation, an important distinction not captured in earlier task parallel performance analysis research, for example in the Cilk-oriented extensions to HPCToolkit (Tallent and Mellor-Crummey, 2009) and the OpenMP profiling tool ompP (Fürlinger and Skinner, 2009). Unlike Cilkview (He et al., 2010), our model is not specific to a particular run time implementation and does not rely on the notion of overall available parallelism. After observing disappointing speedup results and identifying the sources of lost performance, the next logical question is how to improve performance. The following chapters of the dissertation describe the development of our own OpenMP run time, and techniques to mitigate overhead costs, idleness, and work time inflation.

CHAPTER 5

QTHREADS-BASED RUN TIME SYSTEM*

Our main thesis claims that scheduling and run time system implementation used to support task parallel execution should reflect the hardware topology. Evaluation of this claim on actual hardware necessitates a prototype run time system for evaluation. In this chapter we describe the design and implementation of our particular OpenMP run time system, extensions of which appear in Chapters 6 and 7.

5.1 Qthreads

Qthreads is a library designed to provide portable, high performance massive multithreading (Wheeler et al., 2008). It is modeled on the Tera MTA system (Alverson et al., 1992), which supports many simultaneous lightweight threads in hardware by providing a large number of register sets and interleaving instructions from the various active threads. The Qthreads library instead supports lightweight threads in software by providing compact stacks for each and fast context switching between them (Wheeler et al., 2008). The library is cross-platform, supporting IA32, IA64, X86-64, PowerPC, and SPARC architectures.

Figure 5.1 illustrates the software architecture of Qthreads. Each lightweight thread is called a *qthread*. Qthreads are scheduled onto a small set of heavyweight *worker* threads created using the POSIX threads (pthreads) library (IEEE, 1995). A qthread is the smallest schedulable unit of work, such as a set of loop iterations or an OpenMP task, and execution of an application generates many more qthreads than it has worker pthreads. Each worker pthread is pinned to a processor core and assigned to a group, called a *shepherd*. Whereas Qthreads previously allowed only one worker

*The prototype run time system described in this chapter extends the work of our collaborators at the Renaissance Computing Institute and Sandia National Laboratories. Kyle B. Wheeler is the primary author of Qthreads, and Allan K. Porterfield is the primary author of the XOMP interface to Qthreads.

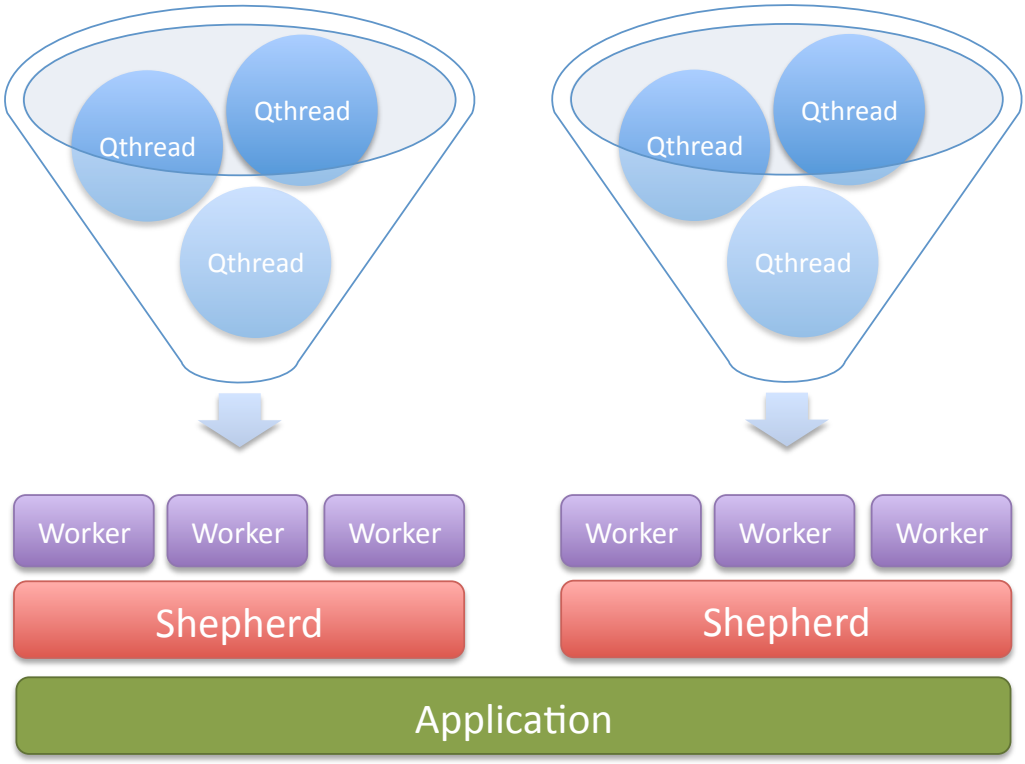


Figure 5.1: Software architecture of Qthreads.

pthread per shepherd, we added support for multiple worker pthreads per shepherd. This support enables us to map shepherds to different architectural components, e.g., one shepherd per core, one shepherd per shared L3 cache, or one shepherd per processor socket.

Like the MTA system, synchronization in Qthreads is supported by full/empty bit (FEB) operations, originally developed for the Heterogeneous Element Processor (HEP) machine (Smith, 1981). In the FEB scheme, a bit is associated with each word in memory. A thread can wait on the condition that a word's bit is full or empty either in a blocking or non-blocking operation. When qthreads block, e.g., when performing an FEB operation, a context switch is triggered. Because this context switch is done in user-space via function calls and requires neither signals nor saving a full set of registers, it is less expensive than an operating system or interrupt-based context switch. This technique allows qthreads to execute uninterrupted until blocked, and, once blocked, allows the scheduler to keep workers busy by switching to other qthreads.

The Qthreads library includes multithreaded loop execution, built upon the core threading components. The API provides three basic parallel loop behaviors: one to create a separate qthread

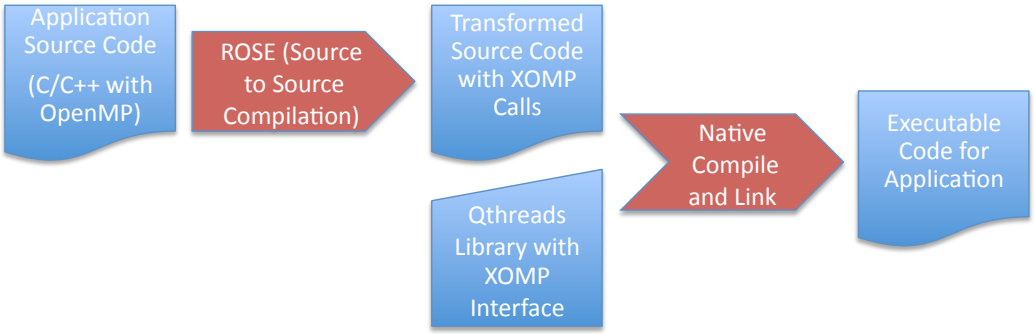


Figure 5.2: Compilation process using ROSE.

for each iteration, one that divides the iteration space evenly among all shepherds, and one that uses a queue-like structure to distribute subranges of the iteration space to enable self-scheduled loops. We use these loop configurations to support OpenMP loop parallelism. The self-scheduling configuration is also the starting point for our task scheduling implementations. The worker pthreads serve as OpenMP threads, and qthreads serve as OpenMP tasks.

5.2 Compilation

In order to execute OpenMP applications using Qthreads, the application source code must first be compiled. Compilation is a two-phase process using both the ROSE source-to-source compiler and the native C++ compiler, as summarized in Figure 5.2. ROSE performs syntactic and semantic analysis on the code and transforms the OpenMP directives into function calls in an API called XOMP (Liao et al., 2010). XOMP defines a common interface for OpenMP 3.0, abstracting out internal implementation details of the run time system. Using a run time library with ROSE is as simple as creating XOMP wrappers for the library, and the ability to hot-swap different implementations of our own run time system enables fast development and testing. In the second compilation stage, the transformed source code is compiled into executable code by the native C++ compiler and linked with the Qthreads library, which implements the XOMP functions. A benefit of this two-step compilation is ease of porting applications: Since the transformed code is standard C++, it is possible to compile the original source code using ROSE on a development system then compile the transformed code using the native compiler on another system with a different ISA.

```
int fib(int n) {
  int x, y;
  if (n < 2) return n;
  #pragma omp task shared(x) untied
     x = fib(n - 1);
  #pragma omp task shared(y) untied
     y = fib(n - 2);
  #pragma omp taskwait
  return x + y;
}
```

```
struct 1_data {
  int n;
  void *x_p;
};

static void OUT__1(void *__out_argv) {
  int n = (int )(((struct 1_data *)__out_argv) -> 1_data::n);
  int *x = (int *)(((struct 1_data *)__out_argv) -> 1_data::x_p);
  int _p_n = n;
   *x = fib((_p_n - 1));
}

struct 2_data {
  int n;
  void *y_p;
};

static void OUT__2(void *__out_argv) {
  int n = (int )(((struct 2_data *)__out_argv) -> 2_data::n);
  int *y = (int *)(((struct 2_data *)__out_argv) -> 2_data::y_p);
  int _p_n = n;
   *y = fib((_p_n - 2));
}

int fib(int n) {
  int x, y;
  if (n < 2) return n;
  struct 1_data __out_argv1;
  __out_argv1.1_data::x_p = ((void *)(&x));
  __out_argv1.1_data::n = n;
  XOMP_task(OUT__1,&__out_argv1,0,sizeof(struct 1_data ),4,1,1);
  struct 2_data __out_argv2;
  __out_argv2.2_data::y_p = ((void *)(&y));
  __out_argv2.2_data::n = n;
  XOMP_task(OUT__2,&__out_argv2,0,sizeof(struct 2_data ),4,1,1);
  XOMP_taskwait();
  return x + y;
}
```

Figure 5.3: Code for *fib* before and after transformation by ROSE.

Figure 5.3 shows a code example for the recursive calculation of the n-th Fibonacci number, before and after transformation by ROSE.[1] The `task` and `taskwait` directives are transformed into calls to `XOMP_task()` and `XOMP_taskwait()`, respectively. Arguments to `XOMP_task()` include a function that wraps the work to be executed in the task, a structure that contains the data context, and relevant flags, e.g., to indicate if the task is *tied* or *untied*. For *shared* variables, e.g., `x` and `y`, a pointer to the variable is stored in the corresponding member of the structure. For *firstprivate* variables, e.g., `n`, only the value is stored. Similar XOMP functions are defined for loops, sections, barriers, etc.

5.3 Execution

The implementation of `XOMP_task()` consists principally of two steps: a new qthread is created and initialized with the function, data context, and flags provided in the arguments, and the new qthread is either enqueued or scheduled for execution, depending on the scheduling policy. Recall that under a help-first task scheduling policy, each new task is enqueued, while under a work-first policy, each new task is scheduled for execution immediately on the thread where it is generated. In the remainder of this discussion, we refer to *tasks* rather than *qthreads* both for simplicity and because the scheduling concepts that we describe are applicable to other task-parallel languages and libraries.

The behavior of the worker threads in the run time system can be modeled as a finite state machine, as shown in Figure 5.4. For some span of time, the thread is actively executing a task. The following events may occur:

- A new child task is generated. Either the child task or the parent task is scheduled, according to the task scheduling policy.
- The task suspends, e.g., waiting at a synchronization point or lock. Control returns to the scheduler to select a ready task to execute.

[1] The transformed code has been partially sanitized for simplicity.

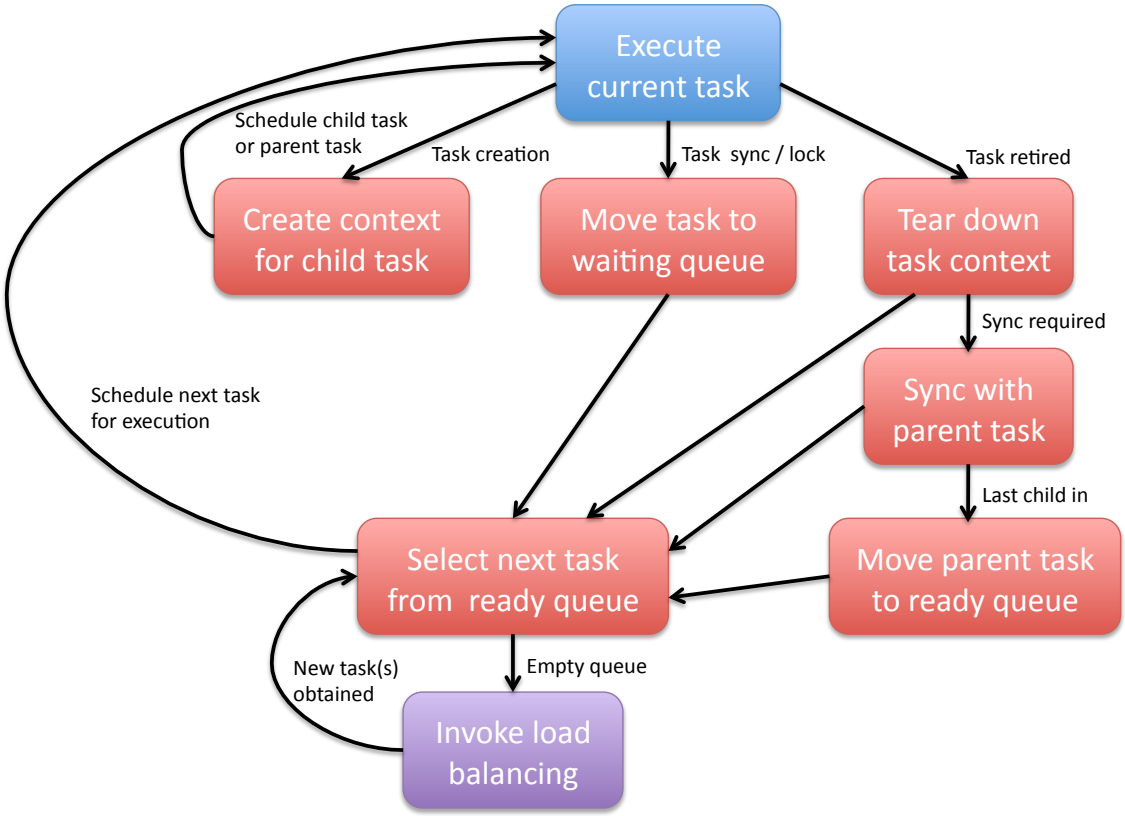


Figure 5.4: State diagram for a worker thread during execution.

- The task completes and is torn down. If required, i.e., the parent task includes a `taskwait`, the task synchronizes with its parent. If this task is the last outstanding child of the parent task, the parent task is enabled for rescheduling. Control returns to the scheduler.

Task scheduling continues according to the task scheduling policy until the task queue is empty. At that point, load balancing is invoked, if available. The implementation of load balancing depends on the scheduling policy.

Our OpenMP library also includes implementations of the other XOMP functions and OpenMP run time functions, e.g., `omp_num_threads()` and `omp_thread_num()`, either through existing Qthreads functionality or new extensions to Qthreads.

CHAPTER 6

HIERARCHICAL SCHEDULING*

In Chapter 2, we presented a survey of scheduling policies for task parallelism, including theoretical bounds on time, space, and cache misses. However, we observed in Chapters 3 and 4 that overhead costs, load imbalance, and work time inflation limit application speedup using existing run time systems in practice. We posit that these issues are exacerbated on multi-socket multicore HPC systems due to NUMA and cache effects, and that a hierarchical scheduler matched to the hardware topology mitigates them for improved performance.

6.1 Limitations of Work Stealing and PDF Schedulers

Figure 6.1 shows a snapshot of the task graph that executes a computation on three threads according to a work stealing schedule. Consider the implications of the cache configuration for this execution. If each thread has its own cache, then the data associated with the subgraph that is executed will reside in its cache. Since tasks running on each thread are from different subgraphs of the task graph, they also use data on different cache lines, minimizing coherence misses. However, if the three threads share a cache, then a fraction of that cache must be used to hold data from each of the three subgraphs of the task graph.

Consider instead an execution of the same task graph using the PDF schedule. The entire subgraph shown in light green would be executed before the other graphs. If each of the threads has its own cache, then coherence misses will occur as cache lines that contain data shared by tasks nearby in the task graph move between the individual caches. If the three threads share a cache, then tasks nearby in the task graph will execute concurrently as the data shared among them is resident

*Contents of this chapter previously appeared in preliminary form in *Proceedings of the First Workshop on Run Time and Operating Systems for Supercomputers* (Olivier et al., 2011).

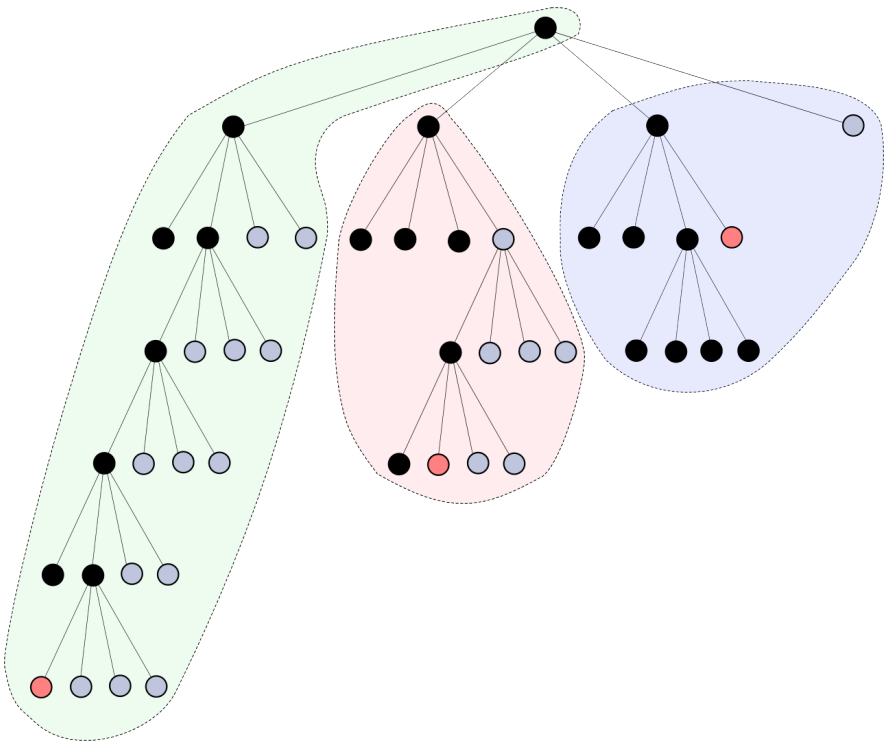


Figure 6.1: Task graph for a computation executing on three threads.

in the shared cache (the *constructive cache sharing* effect (Chen et al., 2007)). In terms of the task graph shown in Figure 6.1, to have all needed data resident in a shared cache (after the inevitable initial cold cache misses) using a PDF schedule would require capacity for only one of the colored subgraphs at once. Using a work stealing schedule, sufficient capacity in the shared cache for data from each of the colored subgraphs would be needed. Of course, insufficient cache capacity would result in capacity misses.

Consider the cache behavior of work stealing and PDF schedulers on a multi-socket multicore system such as the 32 core, four-socket Intel Nehalem system introduced in Section 4.2 and shown in Figure 6.2. Cores on each processor chip share an integrated memory controller and local memory with a shared cache, and access via the interconnect to the processor chips and memory on the other sockets. Each core also has private caches. A work stealing scheduler exploits the private cache well. However, the combined capacity of the private caches is smaller than the shared cache, and the penalty for a miss in the shared cache is much less than a penalty for a miss in the private cache that hits in the shared cache. Local off-chip access takes at least 100 cycles and remote accesses multiple

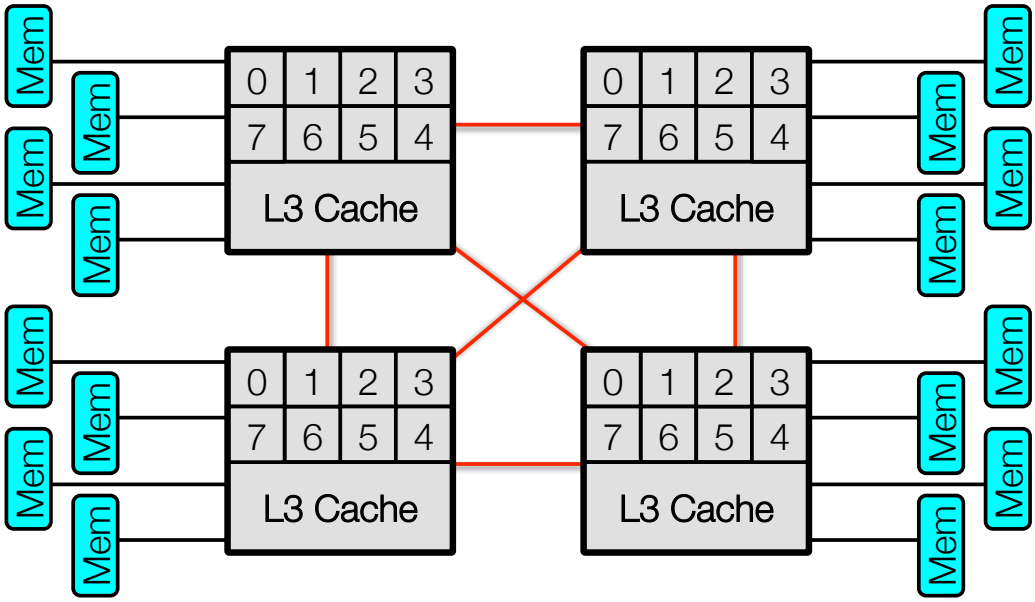


Figure 6.2: Topology of the four-socket Intel Nehalem-EX system.

hundreds of cycles compared to only tens of cycles for L3 hits (Levinthal, 2009). A PDF scheduler more efficiently uses the combined capacity of the shared caches on all the chips, but incurs frequent and expensive inter-chip coherence misses as cache lines are shared between different chips.

Work stealing and PDF schedulers are also challenged by the cost of load balancing operations. In a work stealing scheduler, steals between threads on the same chip are inexpensive, but steals between threads on different threads incur high latency. A PDF scheduler is implemented through a centralized shared queue, so many task enqueue and dequeue operations on multi-socket system are costly remote accesses. Latency costs are compounded by inter-chip locking overheads and resulting queue contention.

6.2 Hierarchical Scheduling with *MTS*

To overcome the limitations of work stealing and centralized shared queues on multi-socket multicore and NUMA systems, we propose a hierarchical approach: multithreaded shepherds, *MTS*. It is designed to combine the benefits of work stealing and PDF schedulers: low-overhead load balancing, isolation of subgraphs that execute on different chips, and constructive sharing of shared cache for tasks that execute on cores of the same chip.

In the MTS scheduler, one shepherd serves all cores on the same chip. These cores share a cache, typically L3, and all are proximal to a local memory attached to that socket. Within each shepherd, we map one pthread worker to each core. Among workers in each shepherd, a shared LIFO queue provides depth-first scheduling close to serial order to exploit the shared cache. Thus, load balancing happens naturally among the workers on a chip and concurrent tasks have possible overlapping localities that can be captured in the shared cache.

Between shepherds work stealing is used to maintain load balance. Each time the shepherd's task queue becomes empty, only the first worker to find the queue empty sets a flag and commences stealing. The other workers in the shepherd spin on cached copies of the flag until the steal is complete and the stealing thread resets the flag. The thief thread steals enough tasks from another shepherd's queue to supply the workers in its shepherd with work. The number of tasks stolen per steal is a tunable parameter, but stealing one per worker in the shepherd ensures that at least immediately following the steal all threads have a task to execute. In practice, we have observed this heuristic to be effective, and Section 6.4.1.3 shows how performance varies for different choices of this parameter. If fewer tasks are available then the thief steals all available tasks on the victim's queue. The stolen tasks are dequeued and collected into a small linked list then enqueued at the thief's queue. If the steal attempt fails because no tasks are available, then the thief thread selects a new victim and begins another steal attempt.

Centralized task queueing for workers within each shepherd reduces the need for remote stealing by providing local load balance. By allowing only one representative worker to steal at a time, in bulk for all workers in the shepherd, communication overheads are reduced. While a shared queue can be a performance bottleneck, the number of cores per chip is bounded, and intra-chip locking operations are fast.

6.3 Scheduler Implementation in Qthreads

The Qthreads scheduler, henceforth called *Q*, supports only one worker pthread per shepherd, and each shepherd maintains a non-blocking FIFO queue. Newly generated tasks are enqueued onto the queues according to a round-robin policy, a suitable strategy for the loop-level parallelism it was originally designed to support.

The first step toward support of *MTS*, is to replace the FIFO queue of the *Q* scheduler with a LIFO queue. A challenge posed by our hierarchical scheduling strategy is the need for a queue that supports concurrent access on both ends, since workers within a shepherd share a queue. Most existing lock-free queues for work stealing, such as the Arora, Blumofe, and Plaxton (ABP) queue (Arora et al., 1998) and resizable variants (Hendler et al., 2006; Chase and Lev, 2005), allow only one thread to execute `push()` and `pop()` operations. Completely lock-free doubly-ended queues (deques) generalize the ABP queue to allow for concurrent insertion and removal on both ends of the queue. Lock-free deques have been implemented with compare-and-swap atomic primitives (Michael, 2003; Sundell and Tsigas, 2005), but speed is limited by their use of linked lists. The ideal non-blocking queue implementation would be array-based and support lock-free concurrent access at both ends. Our present solution is to use a lock-based queue, which forgoes the non-blocking properties of lock-free implementations.

In a modified scheduler, *L*, the LIFO scheduling policy allows the improved locality for child tasks that operate on subsets of the data sets used by their parent tasks. Although the lock-protected enqueue and dequeue operations support concurrent access at both ends, *L* retains the round robin scheduling of new tasks among the queues so that effects of the improved locality can be observed independently in our evaluation.

The final steps toward *MTS* support are to change the placement policy for new tasks so that they are enqueued on the queue of the local shepherd and to implement work stealing between the shepherds. These changes allow the flexibility to support several different scheduling strategies depending on the number of shepherds and the number of worker pthreads per shepherd. For PDF scheduling, we chose a configuration with only one shepherd and as many worker pthreads in that shepherd as there are cores on the system. We designate this centralized shared queue configuration *CQ*. For work stealing with one queue per core, *WS*, chose a configuration with as many shepherds as there are cores on the system and only one pthread worker per shepherd. For our hierarchical scheduler, *MTS*, chose a configuration with as many shepherds as there are chips and as many pthread workers per shepherd as there are cores on each chip. These configurations can be set and reset using environment variables, so that neither the application nor the run time library require recompilation to change them.

Qthreads Implementations, compiled Rose/GCC -O2 -g					
Version Name	Scheduler Implementation	Number of Shepherds	Task Placement	Internal Queue Access	External Queue Access
Q	Stock	one per core	round robin	FIFO (lock-free)	none
L	LIFO	one per core	round robin	LIFO (blocking)	none
CQ	Central Queue	one	N/A	LIFO (blocking)	N/A
WS	Work Stealing	one per core	local	LIFO (blocking)	FIFO stealing
MTS	Multithreaded Shepherds	one per chip	local	LIFO (blocking)	FIFO stealing
ICC	Intel 11.1 OpenMP, compiled -O2 -xHost -ipo -g				
GCC	GCC 4.4.4 OpenMP, compiled -O2 -g				

Table 6.1: Scheduler implementations evaluated: five Qthreads implementations, ICC, and GCC.

6.4 Evaluation

To evaluate the performance of our hierarchical scheduler and the other Qthreads schedulers, we present an evaluation using seven scheduler implementations: five versions of Qthreads[1], the GNU GCC OpenMP implementation, and the Intel ICC OpenMP implementation, as summarized in Table 6.1. The Qthreads implementations are as follows.

- *Q* is the original version of Qthreads and defines each core to be a separate locality domain or *shepherd*. It uses a non-blocking FIFO queue to schedule tasks within each shepherd (individual core). Each shepherd only obtains tasks from its local queue, although tasks are distributed across shepherds on a round robin basis for load balance.

- *L* incorporates a simple double ended locking LIFO queue to replace the original non-blocking FIFO queue. Concurrent access at both ends is required for work stealing, though *L* retains round robin task distribution for load balance rather than work stealing.

- *CQ* uses a single shepherd and centralized shared queue to distribute tasks among all cores in the system. This approach should provide load balance, but contention for the queue limits scalability for fine-grained tasks.

- *WS* provides a shepherd (and individual queue) for each core, and idle shepherds steal tasks from the shepherds running on the other cores. Initial task placement is not round robin

[1] all compiled with GCC 4.4.4 -O2

between queues, but onto the local queue of the shepherd where it is generated, exploiting locality among related tasks.

- *MTS* assigns one shepherd to every processor memory locality (shared L3 cache on chip and attached DIMMs). Each core on a chip hosts a worker thread that shares its shepherd's queue. Only one core is allowed to actively steal tasks on behalf of the queue at a time and tasks are stolen in chunks large enough to keep all cores busy.

The benchmarks for our evaluation are from BOTS (Duran and Teruel, 2011), the suite of benchmarks that we analyzed in Section 4.2:

- *Alignment*: Aligns sequences of proteins using dynamic programming (100 sequences)
- *Fib*: Computes the *n*th Fibonacci number using brute-force recursion ($n = 50$)
- *Health*: Simulates a national health care system over a series of timesteps (144 cities)
- *NQueens*: Finds solutions of the *n*-queens problem using backtrack search ($n = 14$)
- *Sort*: Sorts a vector using parallel mergesort transitioning to sequential quicksort and insertion sort (128M integers)
- *SparseLU*: Computes the LU factorization of a sparse matrix (10000×10000 matrix, 100×100 submatrix blocks)
- *Strassen*: Computes a dense matrix multiply using Strassen's method (8192 x 8192 matrix)

The configuration of the benchmark applications,.e.g., the `for` and single task generation variants of *Alignment* and *SparseLU* and the use of cut-offs, were described in more detail in Section 4.2.

6.4.1 Performance on Intel Nehalem

The first hardware test system for our experiments is the same system described in Section 4.2 and shown in Figure 6.2: a Dell PowerEdge M910 quad-socket blade with four Intel x7550 2.0GHz 8-core Nehalem-EX processors installed for a total of 32 cores. The processors are fully connected using Intel QuickPath Interconnect (QPI) links. Each processor has an 18MB shared L3 cache and each core has a private 256KB L2 cache as well as 32KB L1 data and instruction caches. The blade

Configuration	Alignment	Fib	Health	NQueens	Sort	SparseLU	Strassen
ICC -O2 -xHost -ipo Serial	28.33	100.4	15.07	49.35	20.14	117.3	169.3
GCC -O2 Serial	28.06	83.46	15.31	45.24	19.83	119.7	162.7
ICC 32 threads	0.9110	4.036	1.670	1.793	1.230	7.901	10.13
GCC 32 threads	0.9973	5.283	7.460	1.766	1.204	4.517	10.13
Qthreads MTS 32 workers	1.024	3.189	1.122	1.591	1.080	4.530	10.72

Table 6.2: Sequential and parallel execution times (in seconds) using ICC, GCC, and Qthreads MTS.

has 64 dual-rank 2GB DDR3 memory sticks (16 per processor chip) for a total of 132GB. It runs CentOS Linux with a 2.6.35 kernel. Although the x7550 processor supports HyperThreading (Intel's simultaneous multithreading technology), we pinned only one thread to each physical core for our experiments.

All executables using the Qthreads and GCC run times were compiled with GCC 4.4.4 with -g and -O2 optimization, for consistency. Executables using the Intel run time were compiled with ICC 11.1 and -O2 -xHost -ipo optimization. Reported results are from the best of ten runs unless otherwise specified.

Overall the GCC compiler and ICC compiler produce executables with similar serial performance on the various applications, as shown in Table 6.2. For *Alignment* and *SparseLU*, the best time between the two parallel variants (`single` and `for`) is shown. These sequential execution times provide a basis for us to compare the relative speedup of the various benchmarks. If the -ipo and -xHost flags are not used with ICC on *SparseLU*, the GCC serial executable runs 3x faster than the ICC executable compiled with -O2 alone. The significance of this difference will be clearer in the presentation of parallel performance on *SparseLU* in Section 6.4.1.1. Several other benchmarks also run slower with those ICC flags omitted, though not by such a large margin.

Qthreads *MTS* 32 core performance is faster or comparable to the performance of ICC and GCC. In absolute execution time, *MTS* runs faster than ICC for 5 of the 7 benchmarks by up to 74.4%. It is over 6.6x faster for one benchmark than GCC and up to 65.6% faster on 4 of the 6 others. On two benchmarks *MTS* runs slower: for *Alignment*, it is 12.4% slower than ICC and 2.7% slower than GCC and for *Strassen* it is 5.8% slower than both (although *WS* equaled GCC's performance [see discussion on *Strassen* in sec. 6.4.1.1]). Thus even as a research prototype, ROSE/Qthreads provides competitive OpenMP task execution.

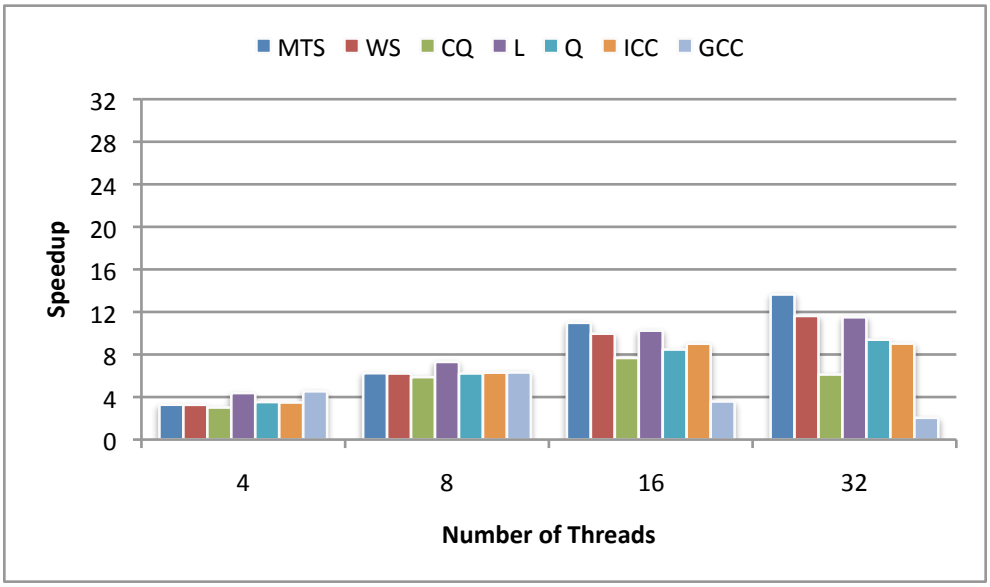


Figure 6.3: Health on 4-socket Intel Nehalem

6.4.1.1 Speedup Results

Differences in achieved speedup on the various benchmarks reveal the strengths and weaknesses of each scheduling approach based on application needs for locality and load balance.

The *Health* benchmark, Figure 6.3, shows significant diversity in performance and speedup. GNU performance is slightly superlinear for 4 cores (4.5x), but peaks with only 8 cores active (6.3x) and by 32 cores the speedup is only 2x. Intel also has scaling issues and performance flattens to 9x at 16 cores. Stock Qthreads *Q* scales slightly better (9.4x), but just switching to the LIFO queue *L* to improve locality between tasks allows speedup on 32 cores to reach 11.5x. Since the individual tasks are relatively small, *CQ* experiences contention on its task queue that limits speedup to 7.7x on 16 cores, with performance degrading to 6.1x at 32 cores. When work stealing, *WS*, is added to Qthreads the performance improves slightly and speedup reaches 11.6x. *MTS* further improves locality and load balance on each processor by sharing a queue across the cores on each chip, and speedup increases to 13.6x on 32 cores. This additional scalability allows Qthread *MTS* a 17.3% faster execution time on 32 cores than any other implementation, much faster than ICC (48.7%) and GCC(116.1%). *Health* provides an excellent example of how both work stealing and queue sharing

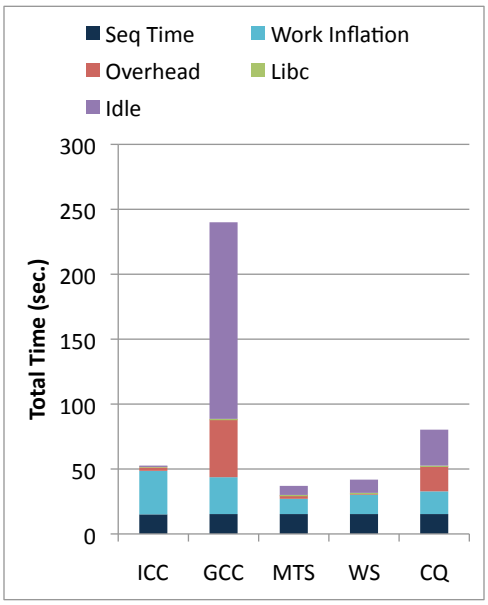


Figure 6.4: Total time over all threads on Health using 32 threads.

within a system can independently and together improve performance, though the failure of any run time to reach 50% efficiency on 32 cores shows that there is room for improvement.

In addition to measuring the parallel speedup, we also used the performance model from Chapter 4 to identify the contributions of idle time, overhead time, sequential equivalent time, and work time inflation to the total time spent by all threads. Figure 6.4 shows the results using 32 threads for ICC, GCC, and the *MTS*, *WS*, and *CQ* Qthreads configurations. The GCC run time and *CQ* suffer from excessive overheads and idleness. Work time inflation is the dominant source of inefficiency for *MTS*, *WS*, and the ICC run time, but *MTS* exhibits the least work time inflation among all schedulers. The work time inflation in Health arises from remote memory accesses and coherence misses that occur more often than necessary, a pathology that is magnified with increasing thread counts. Health is a time-dependent simulation that uses a divide-and-conquer approach to simulate disease-related events (infected population, patients, hospitals) in small villages and to propagate the effects across geographic areas over time. The geographic areas are organized hierarchically so the communication is often localized. In fact, only 2% of patients are transferred between hospitals in different regions. Thus the algorithmic design of *Health* can exploit locality, and we explore this direction further in the next chapter.

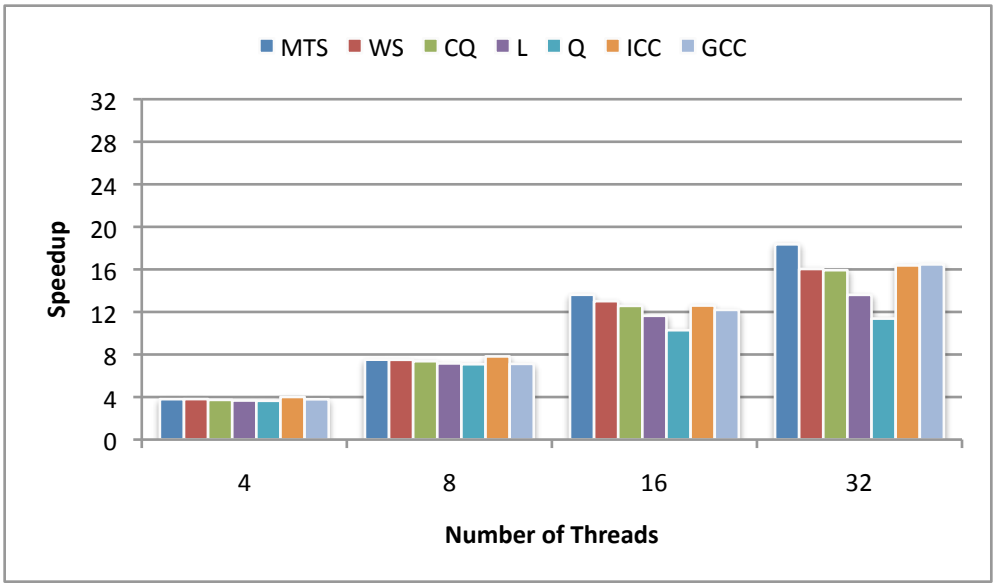


Figure 6.5: Sort on 4-socket Intel Nehalem

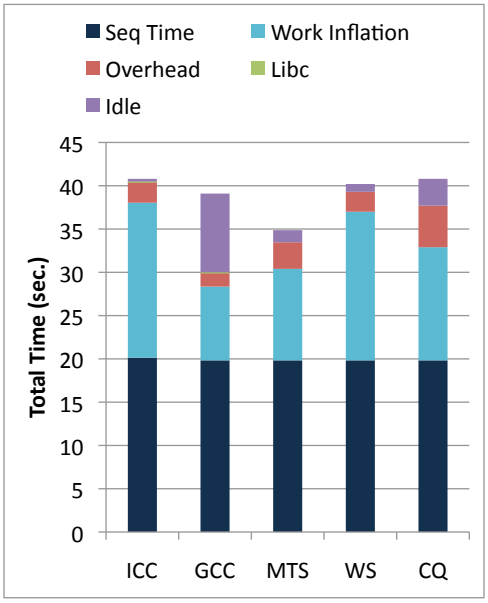


Figure 6.6: Total time over all threads on Sort using 32 threads.

The benefits of hierarchical scheduling can also be seen in Figure 6.5. *Sort*, for which we used a manual cutoff of 32K integers to switch between parallel and serial sorts, achieved speed up of about 16x for 32 cores on ICC and GCC, but just 11.4x for the base version of Qthreads, *Q*. The switch to a LIFO queue, *L*, improved speedup to 13.6x by facilitating data sharing between a parent and child. Independent changes to add work stealing, *WS*, and improve load balance, *CQ*, both improved

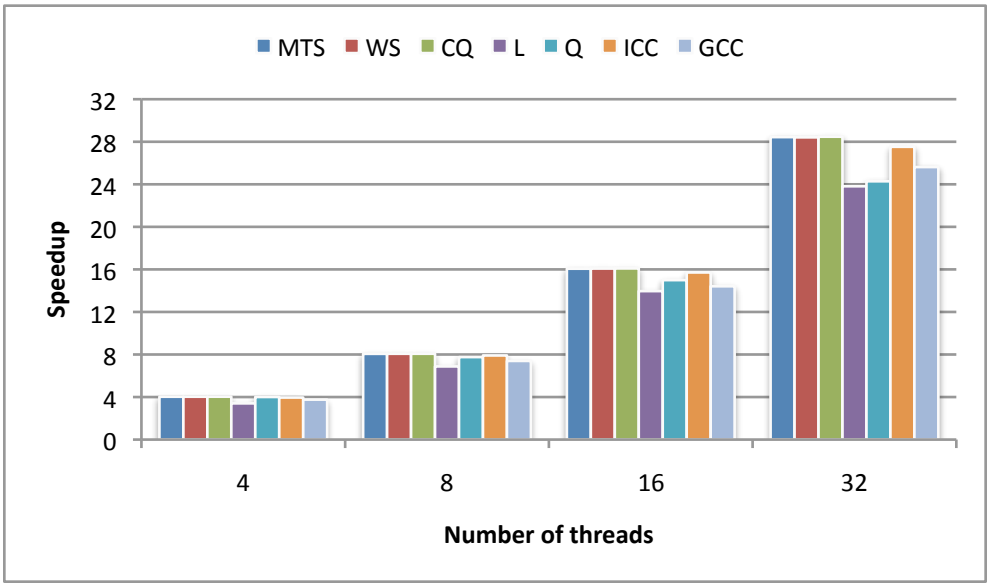


Figure 6.7: NQueens on 4-socket Intel Nehalem

speedup to 16x. By combining the best features of both work stealing and multiple threads sharing a queue, *MTS* increased speedup to 18.4x and achieved a 13.8% and 11.4% reduction in overall execution time compared to ICC and GCC OpenMP versions.

The break-down of total time across all threads is shown in Figure 6.6. GCC suffers from excessive idleness and all schedulers exhibit work time inflation, though *MTS* and GCC have the least. *Sort* permutes the elements of a vector, which inherently leads to remote data accesses. Tasks primarily compare values in two sublists of the vector, which lacks computational intensity. Efficient execution depends on the ability of the memory subsystem to supply the values, a difficult challenge as the thread count increases. On our test machine, the available memory bandwidth per thread decreases when more than 12 threads are used. Ensuring adequate memory concurrency for *Sort* and other bandwidth-limited applications is critical, especially as the number of cores per chip increases.

Locality effects allow *NQueens* to achieve slightly super-linear speedup for 4 and 8 cores using Qthreads. As seen in Figure 6.7, speedup is near-linear for 16 threads and only somewhat sub-linear for 32 threads on all OpenMP implementations. By adding load balancing mechanisms to Qthreads, its speedup improved significantly (24.3x to 28.4x). *CQ* and *WS* both improved load balance beyond what the LIFO queue (*L*) provides but little is gained by combining them together in *MTS*. The additional scaling of these three versions results in a execution time 12.6% faster than ICC and 10.9%

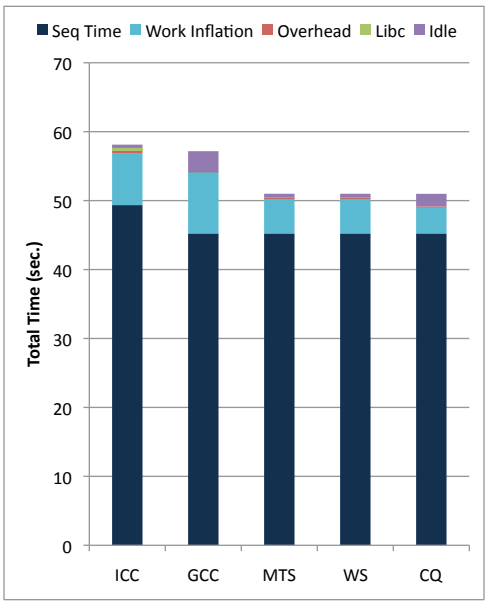


Figure 6.8: Total time over all threads on NQueens using 32 threads.

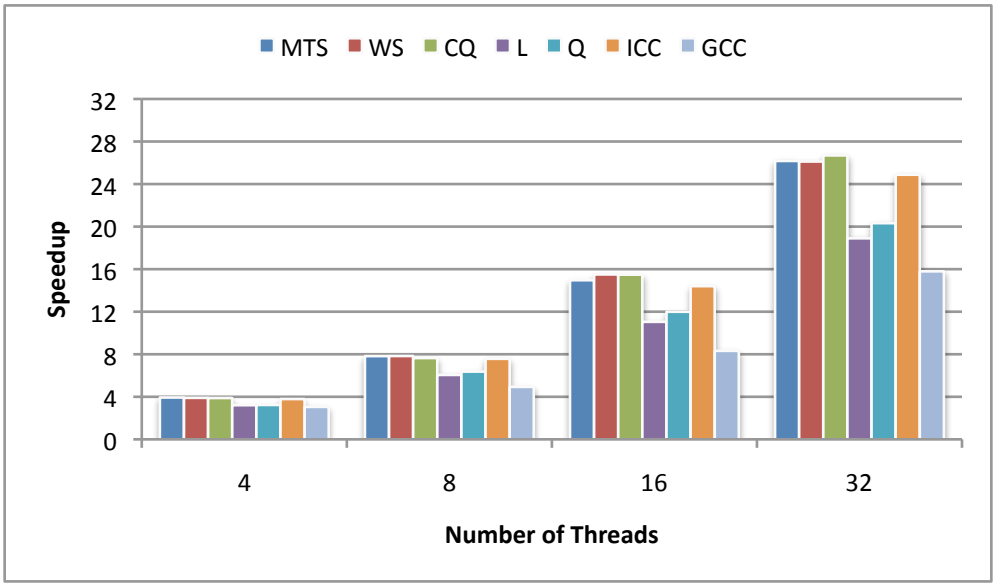


Figure 6.9: Fib on 4-socket Intel Nehalem

faster than GCC. The break-down of total time across all threads is shown in Figure 6.8. Despite the small working set of the program, compulsary cache misses induced by load balancing operations causes some work time inflation with all schedulers.

Fib, Figure 6.9, uses a cut-off to stop the creation of very small tasks, and thus has enough work in each task to amortize the costs of queue access. *CQ* yields performance 2-3% faster than *MTS* and

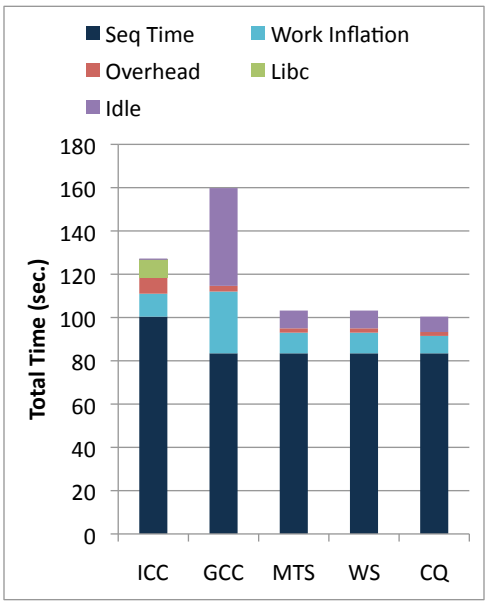


Figure 6.10: Total time over all threads on Fib using 32 threads.

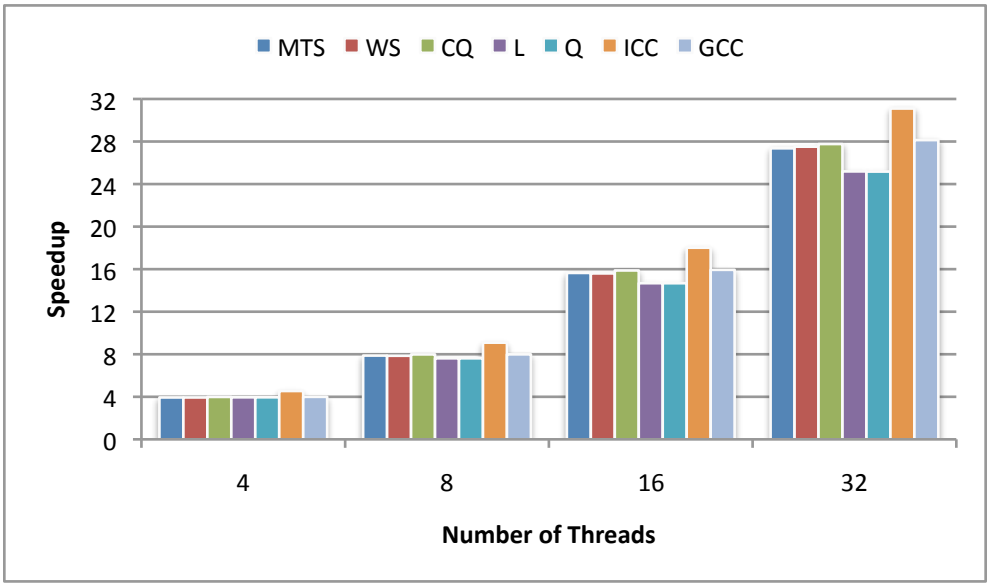


Figure 6.11: Alignment-`single` on 4-socket Intel Nehalem

the other versions of Qthreads, since load balance is good and no time is spent looking for work. The load balancing versions of Qthreads (26.1x - 26.7x) scale better than Intel at 24.9x. Both systems beat GCC substantially at only 15.8x. Overall, the scheduling improvements resulted in *MTS* running 26.5% faster than ICC and 28.8% faster than GCC but 2.0% slower than *CQ*. The break-down of total time across all threads is shown in Figure 6.10. As in NQueens, compulsary cache misses induced

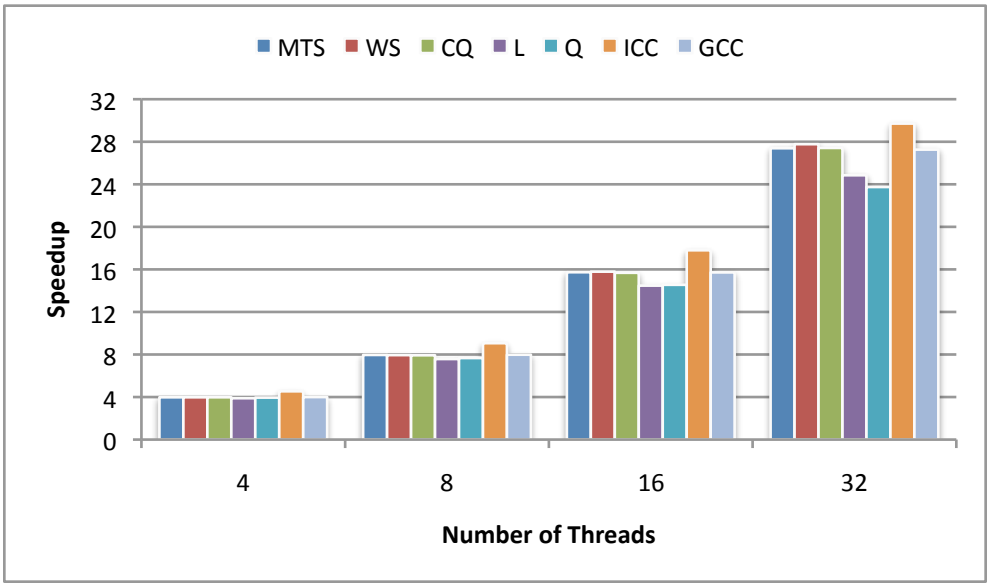


Figure 6.12: Alignment-`for` on 4-socket Intel Nehalem

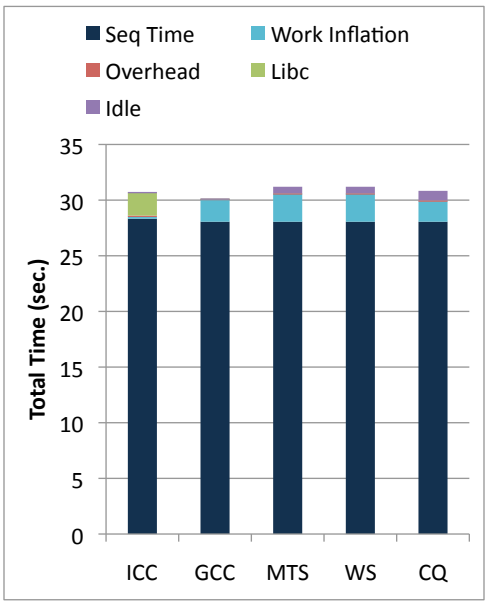


Figure 6.13: Total time over all threads on Alignment-`single` using 32 threads.

by load balancing operations causes some work time inflation with all schedulers. GCC exhibits excessive idleness.

The next two applications *Alignment* and *SparseLU*, each have two versions. For *Alignment*, Figures 6.11 and 6.12, speedup was near-linear for all versions and execution times between GCC and Qthreads were close (GCC +2.7% single initial task version; Qthreads +0.5% parallel loop

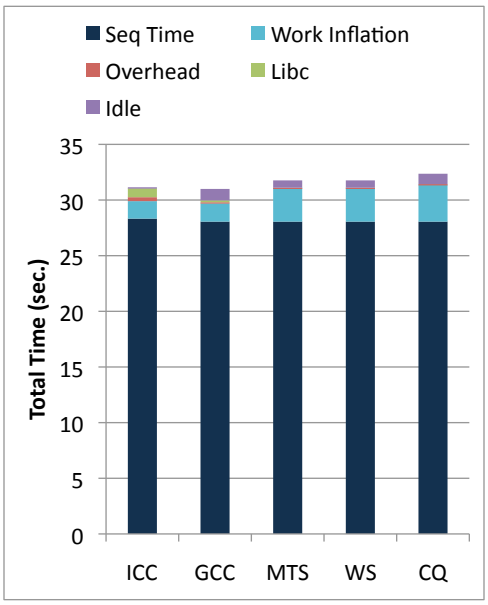


Figure 6.14: Total time over all threads on Alignment-`for` using 32 threads.

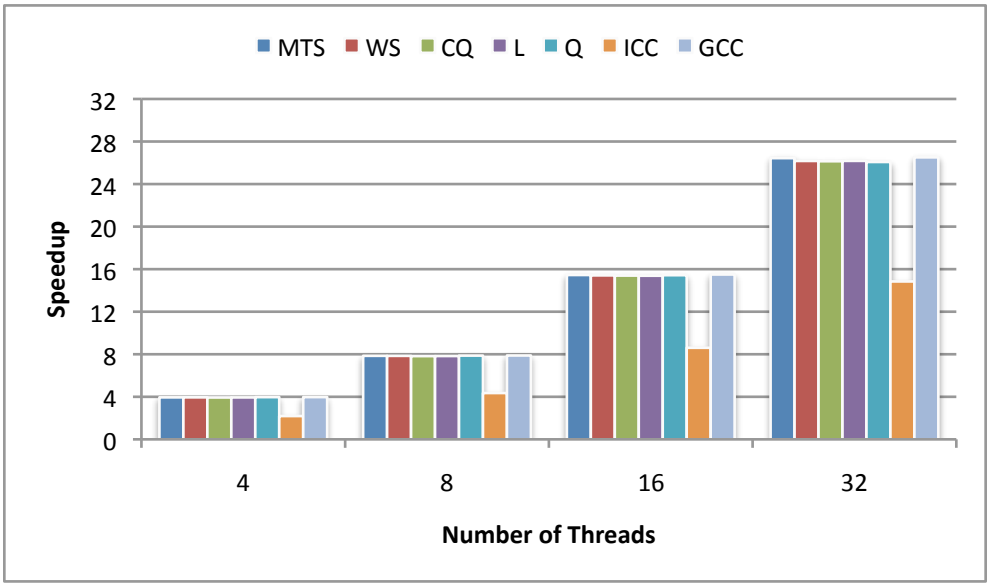


Figure 6.15: SparseLU-`single` on 4-socket Intel Nehalem

version). ICC scales better than GCC or Qthreads *MTS, WS, CQ*, with 12.4% lower execution time. Since *Alignment* has no `taskwait` synchronizations, we speculate that ICC scales better on this benchmark because it maintains fewer bookkeeping data structures in the absence of synchronization. The break-down of total time across all threads is shown in Figures 6.13 and 6.14. None of the schedulers exhibit excessive overheads or idle time, but all exhibit some work time inflation.

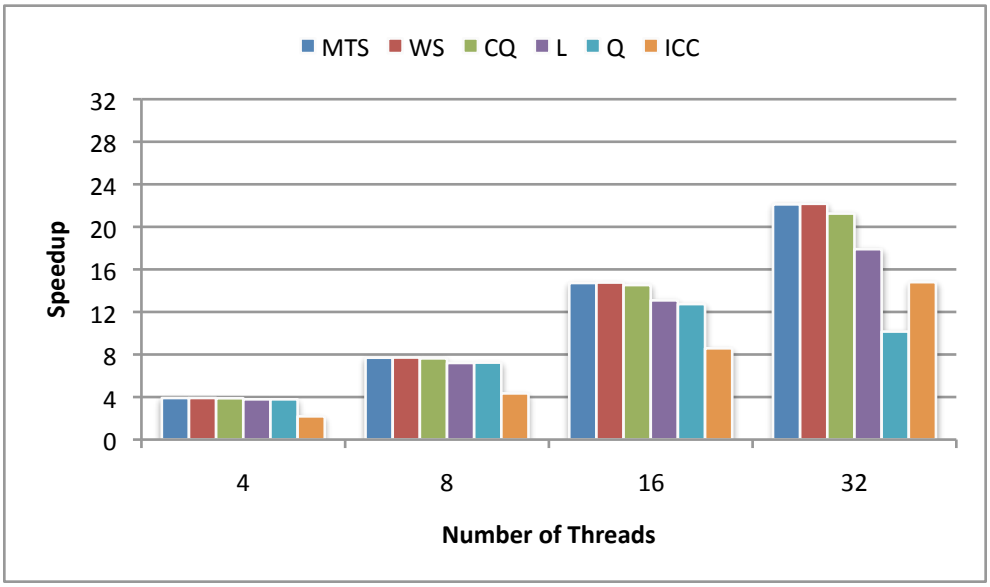


Figure 6.16: SparseLU-`for` on 4-socket Intel Nehalem

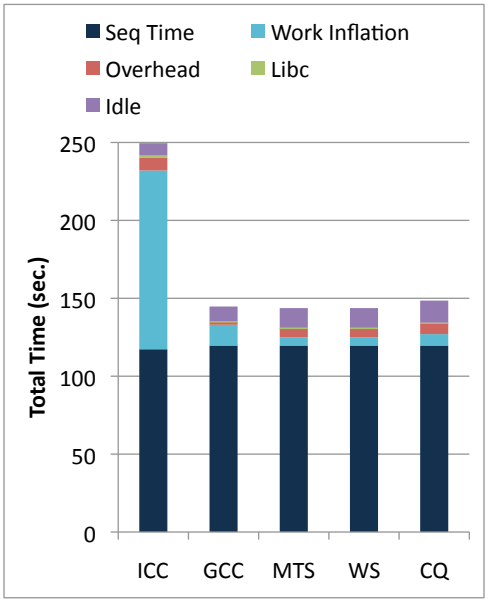


Figure 6.17: Total time over all threads on SparseLU-`single` using 32 threads.

On both *SparseLU* versions, ICC serial performance improved nearly 3x using the -ipo and -xHost flags rather than using -O2 alone. The flags also improved parallel performance, but by only 60%, so the improvement does not scale linearly. On *SparseLU*-`single`, Figure 6.15, the performance of GCC and the various Qthreads versions is effectively equivalent, with speedup reaching 26.2x. Due to the aforementioned scaling issues, ICC speedup reaches only 14.8x. The

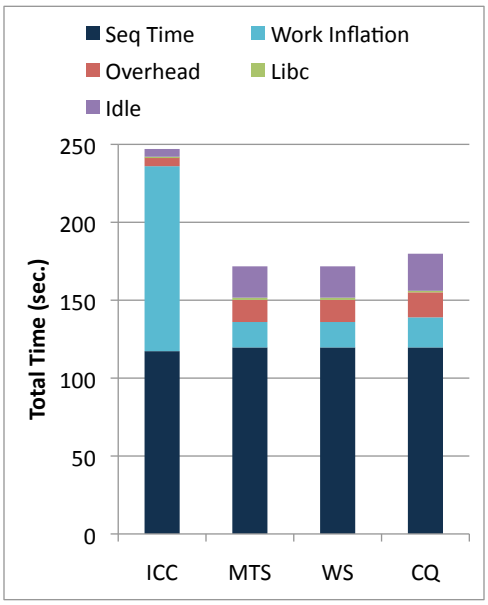


Figure 6.18: Total time over all threads on SparseLU-`for` using 32 threads.

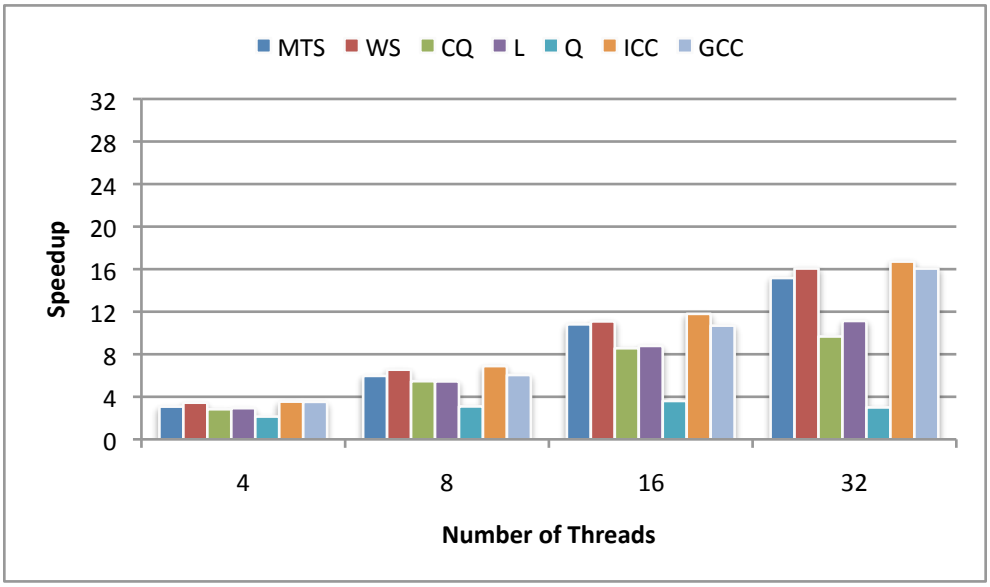


Figure 6.19: Strassen on 4-socket Intel Nehalem

execution times differ by 0.3% between GCC and *MTS* with both about 74.4% faster than ICC. On *SparseLU*-`for`, Figure 6.16, the GCC OpenMP runs were stopped after exceeding the sequential time; thus data is not reported. ICC again scales poorly (14.8x), and Qthreads speedup improves due to the LIFO work queue and work stealing, reaching 22.2x. *MTS* execution time is 46.3% faster than ICC. The break-down of total time across all threads is shown in Figures 6.17 and 6.18. The

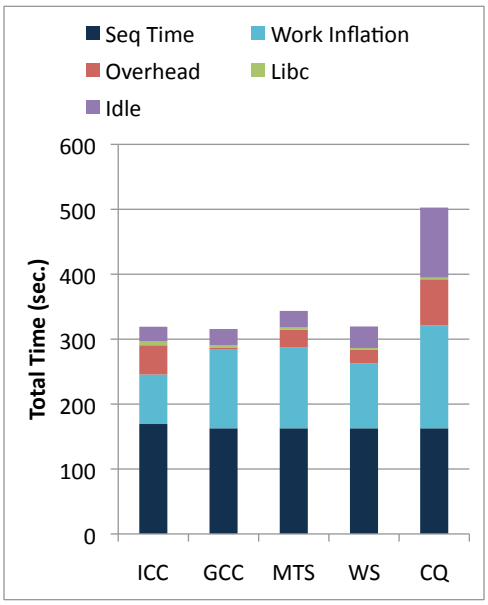


Figure 6.20: Total time over all threads on Strassen using 32 threads.

large amount of work time inflation in ICC is a results of the broken compiler optimizations. The other schedulers exhibit significantly more overhead time, idle time, and work time inflation in the execution of *SparseLU*-`for` than in the execution of *SparseLU*-`single`.

Strassen, Figure 6.19, performs recursive matrix multiplication using Strassen's method. We used the cutoff setting that gave the best performance for each implementation: coarser (128) for *CQ* and *MTS* and the default setting (64) for the others. The execution times of GCC, and *WS* are within 1% of each other on 32 cores, and Intel scales slightly better (16.7x vs 16.1x). For *MTS*, in which only 8 threads share a queue (rather than 32 as in *CQ*) the speedup reaches 15.2x. For *CQ*, however, the performance hit due to queue contention is substantial, as speedup peaks at 9.7x. *Q* performance suffers from the FIFO ordering: not enough parallel work is expressed at any one time, and speedup never exceeds 4x.

The break-down of total time across all threads is shown in Figure 6.20. All schedulers exhibit substantial work time inflation. *Strassen* uses an algorithm that has lower asymptotic time complexity ($O(N^{2.807})$) than standard matrix multiplication ($O(N^3)$). The matrices are recursively decomposed into submatrices, and only seven multiplication operations among the submatrices are required for each submatrix result (compared to eight). However, Strassen's algorithm has reduced reference locality, which leads to less effective cache use and high-latency remote memory accesses. Proposed

Configuration	Alignment (`single`)	Alignment (`for`)	Fib	Health	NQueens	Sort	SparseLU (`single`)	SparseLU (`for`)	Strassen
ICC 32 threads	4.4	2.0	3.7	2.0	3.2	4.0	1.1	3.9	1.8
GCC 32 threads	0.11	0.34	2.8	0.35	0.77	1.8	0.49	N/A	1.4
MTS 32 workers	0.28	1.5	3.3	1.3	0.78	1.9	0.15	0.16	1.9
WS 32 shepherds	0.035	1.8	2.0	0.29	0.60	0.90	0.060	0.24	3.0

Table 6.3: Variability in performance on the Intel Nehalem using ICC, GCC, MTS, and WS (standard deviation as a percent of the fastest time).

techniques to improve locality include switching to standard matrix multiplication at the base of the recursion and changing the data layout to the more cache-friendly Morton ordering, but data reordering itself incurs some costs (Thottethodi et al., 1998).

6.4.1.2 Variability

One interesting feature of a work stealing run time is an idle thread's ability to search for work and the effect that overhead has on performance in regions of limited parallelism or load imbalance. Table 6.3 gives the standard deviation of 10 runs as a percent of the fastest time for each configuration tested with 32 threads. Both Qthreads implementations with work stealing (*WS* and *MTS*) have very small variation in execution time for 3 of the 9 programs. For 8 of the 9 benchmarks, both *WS* and *MTS* show less variability than ICC.

In three cases (*Alignment*-`single`, *Health*, *SparseLU*-`single`), Qthreads *WS* variability was much lower than *MTS*. Since *MTS* enables only one worker thread per shepherd at a time to steal a chunk of tasks, we expect this granularity to be reflected in execution time variations. Overall, we see less variability with *WS* than *MTS* in 6 of the 9 benchmarks. We speculate that normally having all the threads looking for work leads to finding the last work quickest and therefore less variation in total execution time. However, for some programs (*Alignment*-`for`, *SparseLU*-`for`, *Strassen*), stealing multiple tasks and moving them to an idle shepherd results in faster execution during periods of limited parallelism. *WS* also shows less variability than GCC in 6 of the 8 programs for which we have data. No data is shown for *SparseLU*-`for` on GCC, as explained in the previous section.

Benchmark	MTS		WS	
	Steals	Failed	Steals	Failed
Alignment (`single`)	1016	88	3695	255
Alignment (`for`)	109	122	1431	286
Fib	633	331	467	984
Health	28948	10323	295637	47538
NQueens	102	141	1428	389
Sort	1134	404	19330	3283
SparseLU (`single`)	18045	8133	68927	24506
SparseLU (`for`)	13486	11889	68099	32205
Strassen	227	157	14042	823

Table 6.4: Number of remote steal operations during execution by Qthreads MTS and WS schedulers on Intel Nehalem.

6.4.1.3 Performance Analysis of MTS

Limiting the number of inter-chip load balancing operations is central to the design of our hierarchical scheduler (*MTS*). Consider the number of remote (off-chip) steal operations performed by *MTS* and by the flat work stealing scheduler *WS*, shown in Table 6.4. These counts exclude the number of on-chip steals performed by *WS*, and recall that *MTS* uses work stealing only between chips. We observe that *WS* steals more than *MTS* in almost all cases, and some cases by an order of magnitude. *Health* and *Sort* are two benchmarks where *MTS* wins clearly in terms of speedup. *WS* steals remotely over twice as many times as *MTS* on *Sort* and nearly twice as many times as *MTS* on *Health*. The number of failed steals is also significantly higher with *WS* than with *MTS*. A failed steal occurs when a thief's lock-free probe of a victim indicates that work is available but upon acquisition of the lock to the victim's queue the thief finds no work to steal because another thread has stolen it or the victim has executed the tasks itself. Thus, both failed and completed steals contribute to overhead costs.

The *MTS* scheduler aggregates inter-chip load balancing by permitting only one worker at a time to initiate bulk stealing from remote shepherds. Figure 6.21 shows how this improves performance on *Health*, one of the benchmarks sensitive to load balancing granularity. If only one task is stolen at time, subsequent steals are needed to provide all workers with tasks, adding to overhead costs. Our test machine has eight cores per socket, thus eight workers per shepherd, and a target of eight tasks stolen per steal request. This arrangement coincides with the peak performance: When the

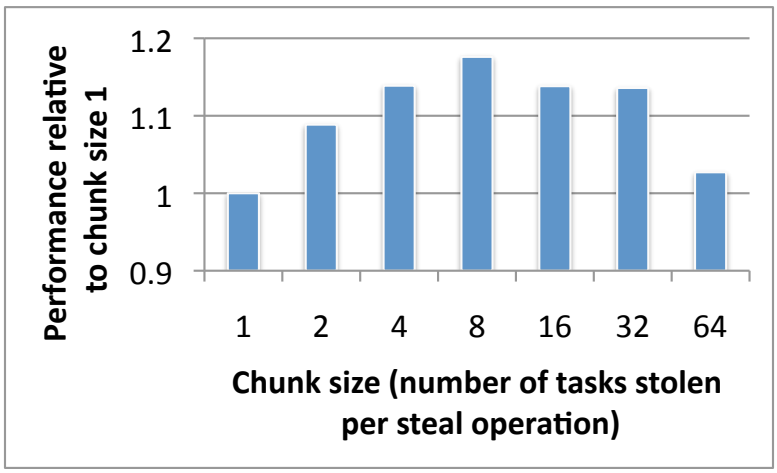


Figure 6.21: Performance on *Health* using MTS based on the choice of chunk size for stealing on Intel Nehalem.

Stolen Tasks	Alignment (`single`)	Alignment (`for`)	Fib	Health	NQueens	Sort	SparseLU (`single`)	SparseLU (`for`)	Strassen
Total	5900	450	2181	159386	423	5214	93117	38198	1355
Per Steal	5.8	4.1	3.4	5.5	4.1	4.6	5.1	2.8	6.0

Table 6.5: Tasks stolen and tasks per steal using the MTS scheduler. Average of ten runs.

Metric	MTS	WS	%Diff
L3 Misses	1.16e+06	2.58e+06	38
Bytes from Memory	8.23e+09	9.21e+09	5.6
Bytes on QPI	2.63e+10	2.98e+10	6.2

Table 6.6: Memory performance data for *Health* using MTS and WS. Average of ten runs on Intel Nehalem.

Metric	MTS	WS	%Diff
L3 Misses	1.03e+7	3.42e+07	54
Bytes from Memory	2.27e+10	2.53e+10	5.5
Bytes on QPI	4.35e+10	4.87e+10	5.6

Table 6.7: Memory performance data for *Sort* using MTS and WS. Average of ten runs on Intel Nehalem.

target number of tasks stolen corresponds to the number of workers in the shepherd, all workers in the shepherd are able to draw work immediately from the queue as a result of the steal.

Frequently the number of tasks available to steal is less than the target number to be stolen. Table 6.5 shows the total number of tasks stolen and the average number of tasks stolen per steal operation, given a target of 8 tasks per steal. Across all benchmarks, the range of tasks actually stolen

per steal is 2.8 to 6.0. The numbers skew downward due to a scarcity of work during start-up and near termination, when only one or few tasks are available at a time. Note the lower number both of total steals and tasks per steal for the `for` versions of *Alignment* and *SparseLU* compared to the `single` versions. Loop parallel initialization provides good initial load balance so that fewer steals are needed, and those that do occur sporadically are near termination and synchronization phases.

Another benefit of the *MTS* scheduler is better L3 cache performance, since all workers in a shepherd share the on-chip L3 cache. The *WS* scheduler exhibits poorer cache performance, and subsequently, more reads to main memory. Tables 6.6 and 6.7 show the relevant metrics for *Health* and *Sort* as measured using hardware performance counters, averaged over ten runs. They also show more traffic on the Quick Path Interconnect (QPI) between chips for *WS* than for *MTS*. QPI traffic occurs when data is requested and transferred from either remote memory or remote L3 cache, i.e., attached to a different socket of the machine. Not only are remote accesses higher latency, but they also result in remote cache invalidations of shared cache lines and subsequent coherence misses. Increased QPI traffic in *WS* reflects more remote steals and more accesses to data in remote L3 caches and remote memory. In summary, *MTS* gains advantage by exploiting locality among tasks executed by threads on cores of the same chip, making good use of the shared L3 cache to access memory less frequently and avoid high latency remote accesses and coherence misses.

6.4.2 Performance on AMD Magny Cours

We also evaluate the Qthreads schedulers against ICC and GCC on a 2-socket AMD Magny Cours system, one node of a cluster at Sandia National Laboratories. Each socket hosts an Opteron 6136 multi-chip module: two quad-core chips that share a package connected via two internal HyperTransport (HT) links. The remaining two HT links per chip are connected to the chips in the other socket, as shown in Figure 6.22. Each chip contains a memory controller with 8GB attached DDR3 memory, a 5MB shared L3 cache, and four 2.4 MHz cores with 64kb L1 and 512kb L2 caches. Thus, the machine has a total of 16 cores and 32GB memory, evenly divided among four HyperTransport-connected NUMA nodes (one per chip, two per socket). The system runs Cray compute-node Linux kernel 2.6.27, and we used the GCC 4.6.0 with -O3 optimization and ICC 12.0 with -O3 -ipo -msse3 -simd optimization.

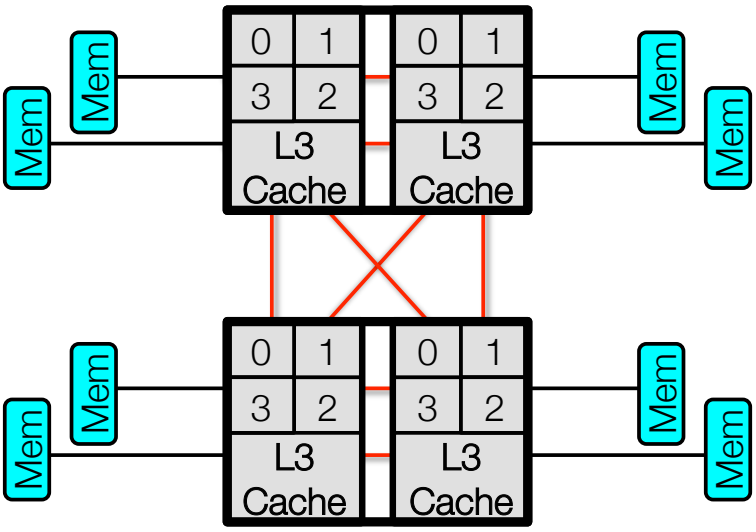


Figure 6.22: Topology of the 2-socket/4-chip AMD Magny Cours.

Configuration	Alignment	Fib	Health	NQueens	Sort	SparseLU	Strassen
ICC	23.93	107.9	10.18	60.56	18.51	156.0	214.9
GCC	29.77	105.0	10.67	58.16	17.72	153.4	211.1

Table 6.8: Sequential execution times using ICC and GCC on the AMD Magny Cours.

We ran the same benchmarks with the same parameters used in the Intel Nehalem evaluation. We report sequential execution times in Table 6.8. Again, interprocedural optimization (-ipo) in ICC was essential to match the GCC performance; execution time was more than 500 seconds without it. The greatest remaining difference between the sequential times is on *Alignment*, where GCC is 20% slower than ICC.

We obtained speedup results using 16 threads for Qthreads configurations with one shepherd per chip, *MTS (4Q)*; one shepherd per socket, *MTS (2Q)*; one shepherd per core (flat work stealing), *WS*; ICC; and GCC. At least one of the Qthreads variants matches or beats ICC and GCC on all but one of the benchmarks. Moreover, the Qthreads schedulers achieve near-linear to slightly super-linear speedup on 6 of the 9 benchmarks, shown in Figure 6.23: the two versions of *Alignment*, *Fib*, *NQueens*, and the two versions of *SparseLU*. Of those, speedup using ICC is 22% and 23% lower than Qthreads on the two versions of *Alignment*, 10% and 18% lower on the two versions of *SparseLU*, 9% lower on *NQueens* and 7% lower on *Fib*. GCC is 42% lower than Qthreads on Fib,

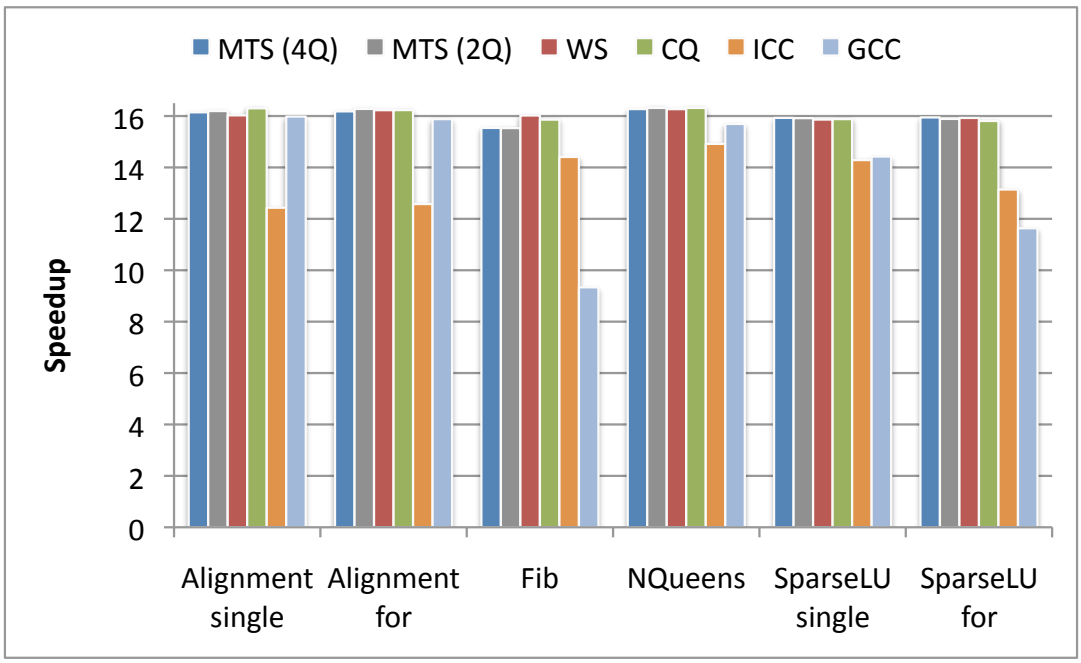


Figure 6.23: Six BOTS benchmarks on 2-socket AMD Magny Cours using 16 threads that show linear or near-linear speedup using Qthreads.

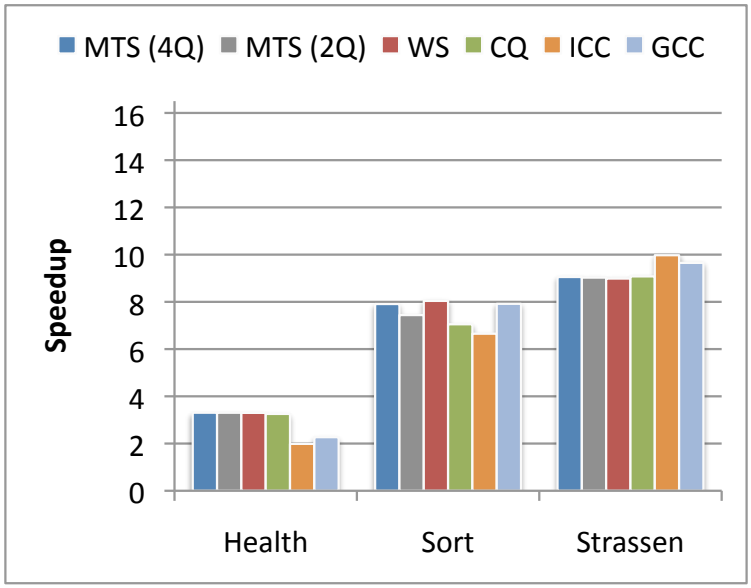


Figure 6.24: Three BOTS benchmarks on 2-socket AMD Magny Cours using 16 threads showing sub-linear speedup.

9% and 27% lower on the two versions of *SparseLU*, and close on *NQueens* and both versions of *Alignment*.

Config	Alignment (`single`)	Alignment (`for`)	Fib	Health	NQueens	Sort	SparseLU (`single`)	SparseLU (`for`)	Strassen
ICC	2.2	0.80	1.3	14	1.1	8.2	0.62	0.31	2.5
GCC	0.035	0.27	5.4	0.38	0.96	3.5	0.016	0.025	1.1
MTS (4Q)	0.25	0.63	1.5	0.17	0.13	1.1	0.012	0.16	0.98
MTS (2Q)	0.46	0.68	1.4	0.069	0.24	0.30	0.015	0.081	0.87
WS	0.21	1.3	1.5	0.15	0.13	1.8	0.036	0.094	1.4

Table 6.9: Variability in performance on AMD Magny Cours using 16 threads (standard deviation as a percent of the fastest time).

On three of the benchmarks, shown in Figure 6.24, no run time achieves ideal speedup. *Strassen* is the only benchmark on which ICC and GCC outperform Qthreads, and even ICC falls just short of 10X. On *Sort*, the best performance is with Qthreads *WS*, *MTS(4Q)*, and GCC all at roughly 8x. Speedup is lower with Qthreads *MTS (2Q)* and still lower with *CQ*, indicating that centralized queueing beyond the chip level is counterproductive. ICC speedup lags behind the other schedulers on this benchmark. Speedup on *Health* peaks at 3.3x on this system using the Qthreads schedulers, with even worse speedup using ICC and GCC.

The variability in execution times is shown in Table 6.9. The standard deviations for all benchmarks on the *MTS* and *WS* Qthreads implementations are below 2% of the best case execution time. On all but two of the benchmarks, the *MTS* standard deviation is less than 1%.

The Magny Cours results demonstrate that the competitive, and in many cases superior, performance of our Qthreads schedulers against ICC and GCC is not confined to the Intel architecture. Differences in performance using the various Qthreads configurations seem less pronounced than they were on the four socket Intel machine. However, those differences were strongest on the Intel machine at 32 threads, and the AMD system only has 16 threads. Some architectural differences go beyond the difference in core count. *MTS* is designed to leverage locality in shared L3 cache, but the Magny Cours has much less L3 cache per core than the Intel system (1.25MB/core versus 2.25MB/core). Less available cache also accounts for worse performance on the data-intensive *Sort* and *Health* benchmarks. Considering the whole of the Magny Cours results, the key observations are that the performance of the Qthreads schedulers is not limited to Intel platforms and that the hierarchical scheduler does not degrade performance.

Configuration	Alignment	Fib	Health	NQueens	Sort	SparseLU	Strassen
GCC	53.96	139.2	45.60	63.62	33.59	632.7	551.3

Table 6.10: Sequential execution times on the SGI Altix.

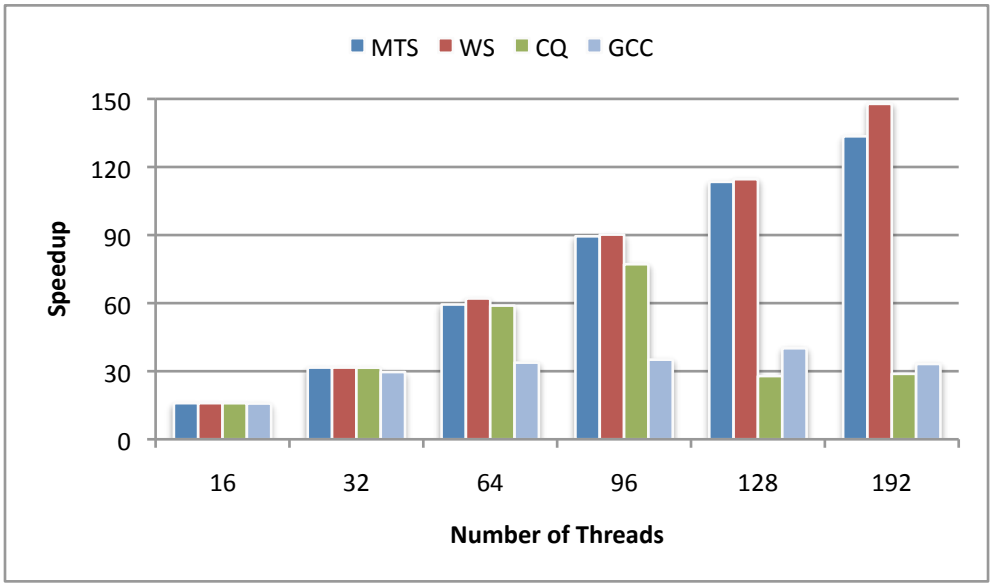


Figure 6.25: NQueens on SGI Altix

6.4.3 Performance on SGI Altix

We evaluate scalability beyond 32 threads on an SGI Altix 3700. Each of the 96 nodes contains two 1.6MHz Intel Itanium2 processors and 4GB of memory, for a total of 192 processors and 384GB of memory. The nodes are connected by the proprietary SGI NUMALink4 network and run a single system image of SuSE Linux kernel 2.6.16. We used the GCC 4.5.2 compiler as the native compiler for our ROSE-transformed code and the GCC OpenMP run time for comparison against Qthreads. The version of ICC on the system is not recent enough to include support for OpenMP tasks. Sequential execution times, given in Table 6.10, are slower than those of the other machines, because the Itanium2 is an older processor, runs at a lower clock rate, and uses a different instruction set (IA64).

The best observed performance on any of the benchmarks is on *NQueens*, shown in Figure 6.25. *WS* achieves 115x on 128 threads (90% parallel efficiency) and reaches 148x on 192 threads. *MTS* reaches 134x speedup. On this machine, the *MTS* configuration has two threads per shepherd to match the two processors per NUMA node. *CQ* tops out at 77x speedup on 96 threads, beyond

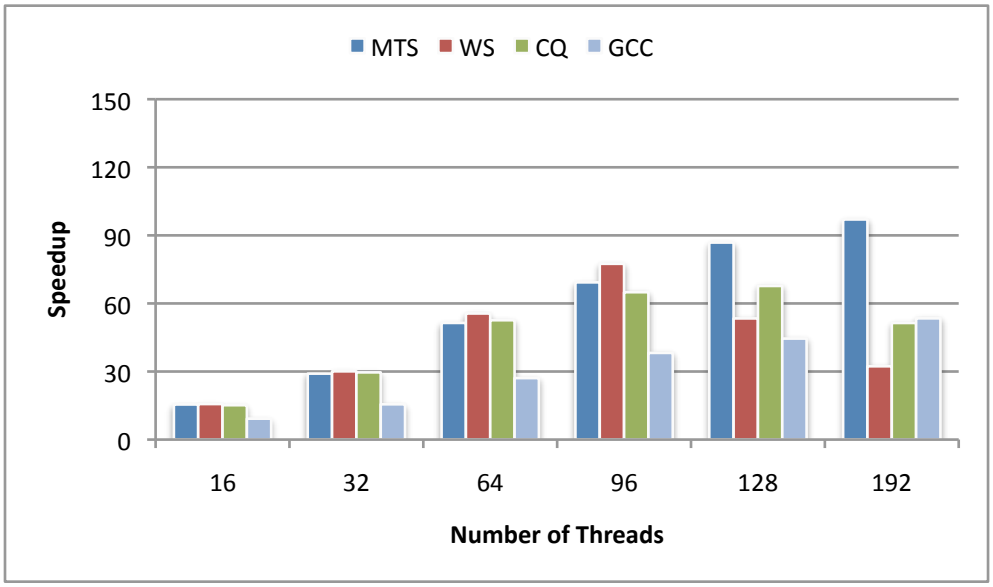


Figure 6.26: Fib on SGI Altix

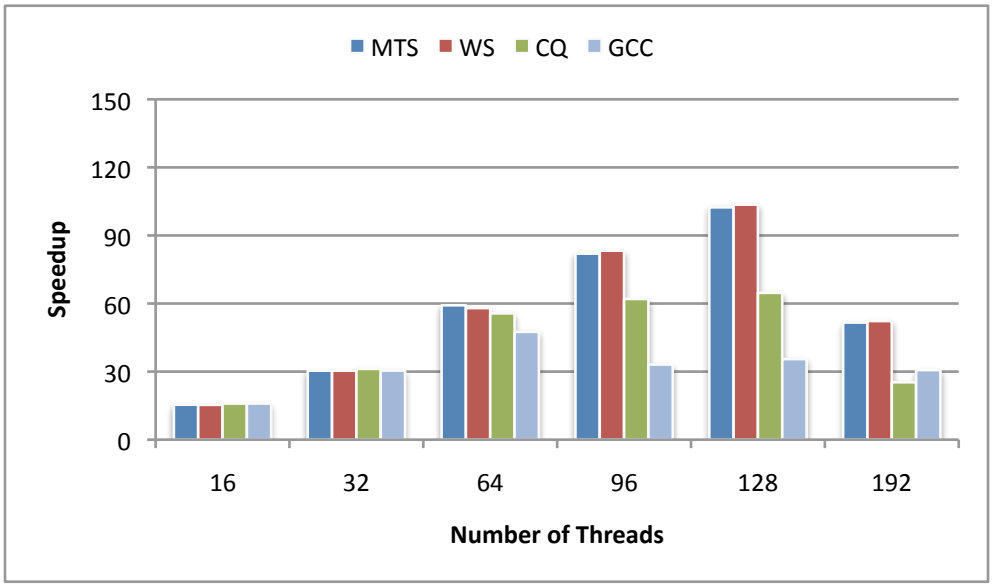


Figure 6.27: Alignment-`single` on SGI Altix

which overheads from queue contention become overwhelming. GCC gets up to only 40x speedup. Although no run time achieves linear speedup on the full machine, they all reach 30x to 32x speedup with 32 threads; this underscores the importance of testing at higher processor counts to evaluate scalability. On the *Fib* benchmark, shown in Figure 6.26, *MTS* almost doubles the performance of *CQ* and GCC on 192 threads, with a maximum speedup of 97x. *CQ* peaks at 68x speedup on 128

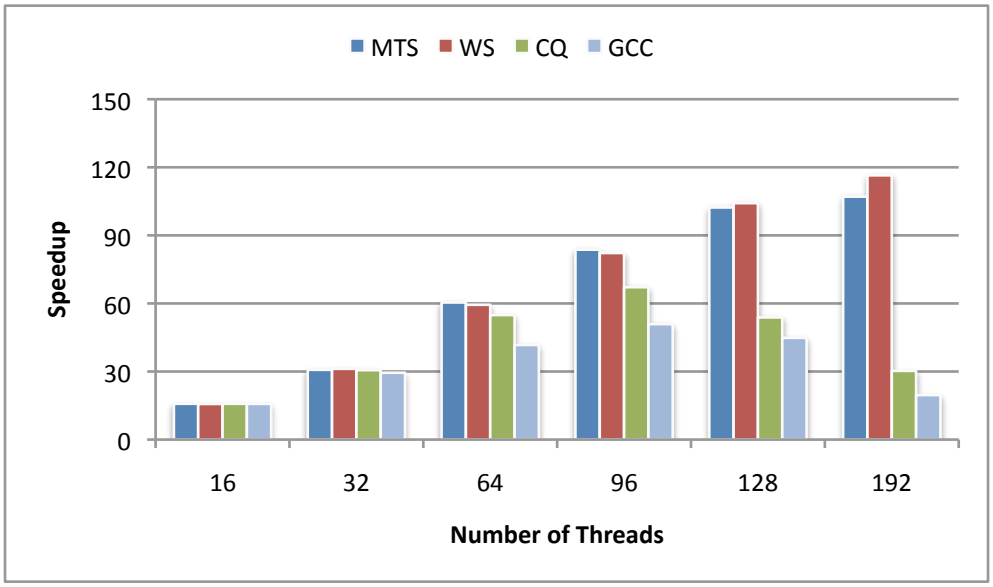


Figure 6.28: Alignment-`for` on SGI Altix

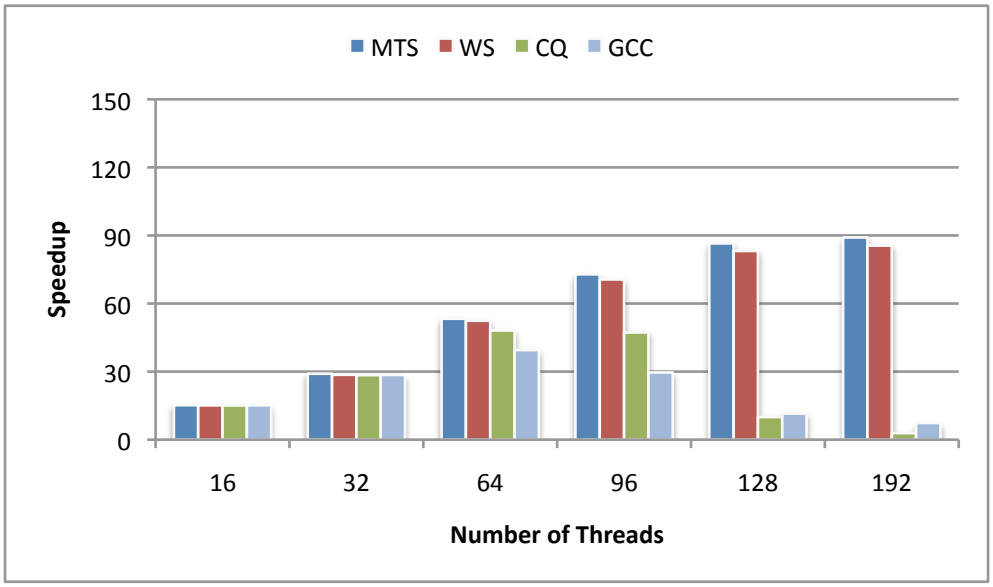


Figure 6.29: SparseLU-`single` on SGI Altix

threads and *WS* exhibits its worst performance relative to *MTS*, maxing out at 77x speedup on 96 threads.

We see better peak performance on *Alignment*-`for` (Figure 6.28) than *Alignment*-`single` (Figure 6.27). *WS* reaches 116x speedup on 192 threads and *MTS* reaches 107x, with *CQ* and GCC performing significantly worse. On the other hand, *SparseLU*-`single` (Figure 6.29) scales better

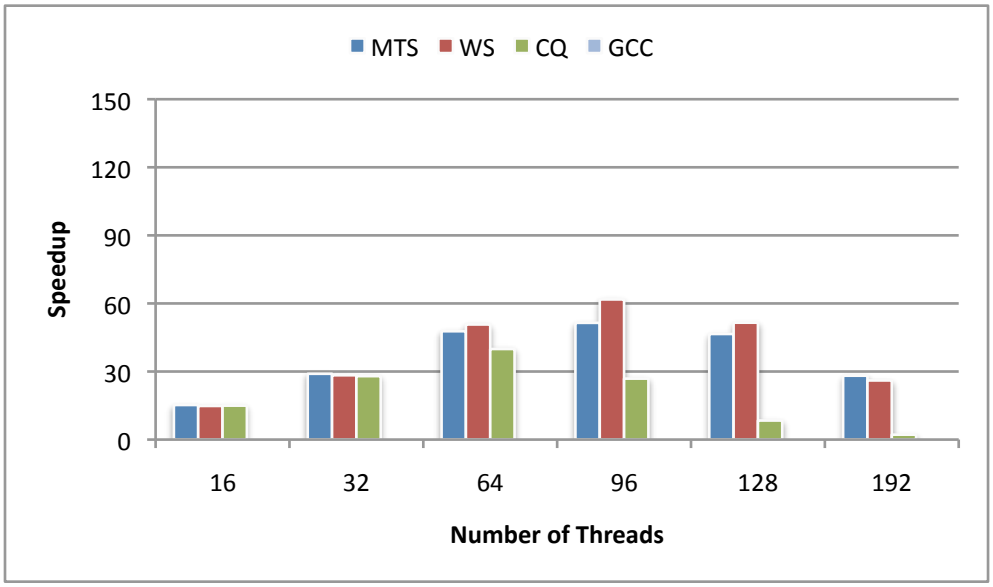


Figure 6.30: SparseLU-`for` on SGI Altix

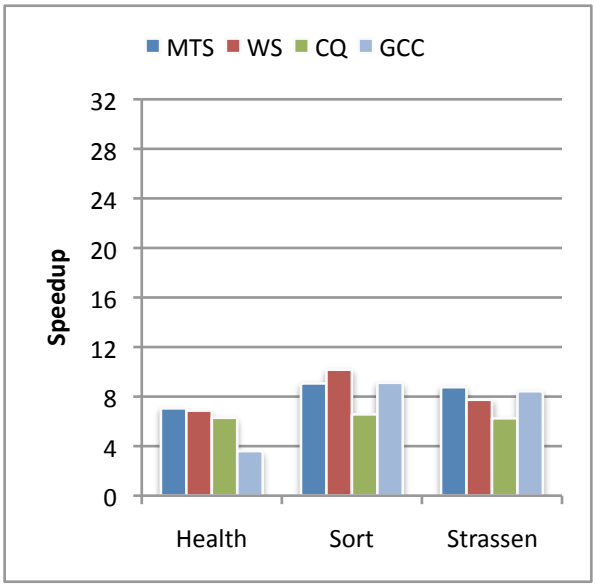


Figure 6.31: Health, Sort, and Strassen on SGI Altix using 32 threads

than *SparseLU*-`for` (Figure 6.30). Peak speedup on *SparseLU*-`single` is 89x with *MTS* and 86x with *WS*, while *SparseLU*-`for` achieves a peak speedup of 60x. As was the case on the 4-socket Intel machine, GCC is unable to complete after a timeout equal to the sequential execution time.

For three of the benchmarks, no improvement in speedup was observed beyond 32 threads: *Health*, *Sort*, and *Strassen*. As shown in Figure 6.31, none exceed 10x speedup on the Altix. These

benchmarks were also observed to be the most challenging on the four-socket Intel and two-socket AMD systems. Still, the Qthreads *MTS* and *WS* schedulers match or exceed GCC in performance.

6.5 Summary

Through the combination of shared LIFO queues and work stealing, our hierarchical scheduler maintains good load balance while supporting effective cache performance and limiting overhead costs. With our *MTS* scheduler implementation, applications using the Qthreads OpenMP run time achieve speedup as high or higher on Intel Nehalem and AMD Magny Cours machines than commonly available production OpenMP 3.0 implementations. Measurements using hardware performance counters confirm improved cache performance and decreased inter-socket traffic in executions using hierarchical scheduling. The scalability results on the SGI Altix go beyond previous BOTS evaluations (Duran et al., 2009) that only present results on up to 32 cores. Several benchmarks reach speedup of 90X-150X on 196 cores. Overall, our performance study reveals the strengths and limitations of hierarchical scheduling on multi-socket multicore architectures and demonstrates that mirroring the hierarchical nature of the hardware in the run time scheduler can indeed improve application performance. As we shall see in Chapter 7, our hierarchical run time system also provides a structural foundation for additional strategies to exploit locality in challenging data-intensive applications such as the *Health* benchmark.

CHAPTER 7

LOCALITY-BASED SCHEDULING

In this chapter, we propose a framework for locality-based scheduling of task parallel programs using programmer annotations to the application code. We extend Qthreads *MTS* to implement a scheduler that honors such annotations, enabling performance improvements in applications that otherwise exhibit *work time inflation* due to excessive non-local data accesses.

7.1 Work Time Inflation in *Health* and *Heat*

Recall that we introduced the concept of *work time inflation* in Chapter 4. In this section, we study the impact of *work time inflation* on two example applications, in particular as a function of the number of threads engaged in the computation.

First consider the *Health* benchmark from the BOTS (Duran and Teruel, 2011) suite of task parallel benchmark applications, a simulation of a nation-wide heath system, to demonstrate how NUMA effects can increase work time. This simulation uses a divide-and-conquer approach to simulate events in small villages and to propagate the effects across wider and wider geographic areas over time. The program is parallelized using the task model from OpenMP 3.0 by encapsulating each recursive call as an explicit task for execution by an available thread. On the four-socket Intel Nehalem-EX system using the Intel OpenMP compiler and run time system, we observe a sequential execution time of 15.1 seconds and a parallel execution time of about 1.7 seconds. In Chapter 6, we observed that using our hierarchical scheduling strategy as implemented in the Qthreads *MTS* scheduler improved the performance of this application, but the parallel execution is still well short of ideal speedup.

Figure 7.1, shows a break-down of the execution time across all threads for parallel executions of the health simulation with the manual cutoff set. The time is summed across threads so perfect

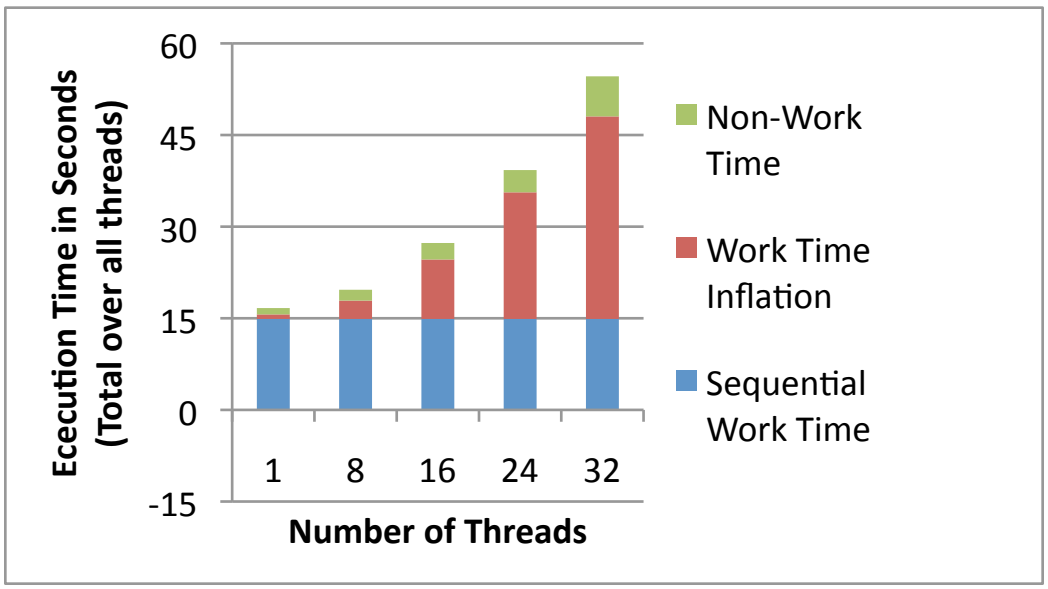


Figure 7.1: Total time over all threads on *Health*

scaling would require this time to be constant as the number of threads increases. We observe instead that work time dominates the total overall time. We can view the work time as the sum of two parts: The amount of time equal to the sequential execution time, and the amount of time spent in excess of the sequential execution time. This excess time, the *work time inflation*, accounts for the observed performance gap in the health simulation. Work time increases substantially as we use more threads although we use the same input problem size throughout, i.e., we evaluate strong scaling.

Figure 7.2 shows the execution time break-down for another application, a two-dimensional heat diffusion simulation, *Heat*, from the Cilk example set. The simulation uses five-point stencil computations and, like the health simulation, is parallelized by generating tasks in a divide-and-conquer problem decomposition with a cutoff threshold for granularity control. Similarly to the health simulation, this heat simulation achieves speedup (11.2X) and parallel efficiency (35%) on a 32-thread execution that are well short of the ideal values. We observe some work time inflation (2.8 seconds) even in a single-threaded execution of the OpenMP executable. This inflation is an artifact of parallel code limiting the efficacy of compiler optimizations, a cause of work time inflation that is outside the scope of this dissertaton. The remaining work time inflation, which increases as we use more threads, is again due to NUMA effects and memory performance.

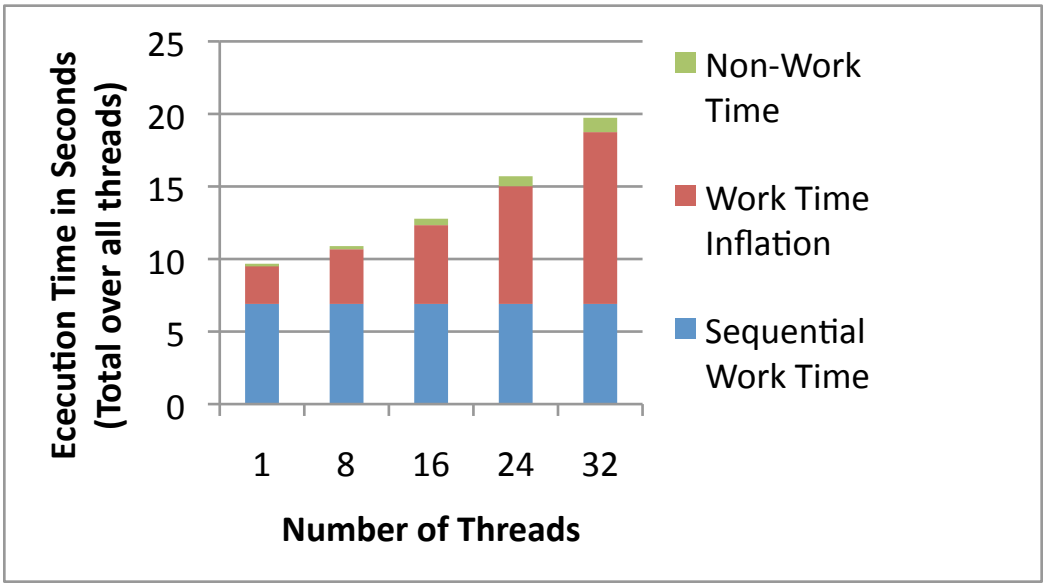


Figure 7.2: Total time over all threads on *Heat*

Consider some measurements we gathered using hardware performance counters during a sequential and a 32-thread parallel execution of the 2D heat simulation. Both times, the number of bytes read from memory was 13.6GB. However, the parallel execution generates 43GB of interconnect traffic, over 500 times the volume seen in the sequential execution. This interconnect traffic represents expensive remote loads and cache invalidations that cause work time inflation.

7.2 First Touch and Scheduling

OpenMP offers no inherent means to express locality for data to threads or tasks. Thus, performance-oriented users must rely on non-portable solutions such as operating system tools (e.g., libnuma in Linux), third party libraries (e.g., hwloc (Broquedis et al., 2010b)), or heuristics. Integrated OpenMP locality support would overcome the inconvenience of these methods. However, a common programming idiom for NUMA systems exploits the first-touch page placement policy (Bolosky et al., 1991; Nikolopoulos et al., 2001; Marathe and Mueller, 2006). We now examine how this heuristic is used with parallel loops and how current task schedulers fail to support an equivalent idiom for task parallel programs, a role that our locality framework fills.

```
#pragma omp parallel
{
   #pragma omp for schedule(static)
      for (i = 0; i < n; i++)
         init(data[i]);

   for (step = 0; step < nsteps; step++)
      #pragma omp for schedule(static)
         for (i = 0; i < n; i++)
            compute(data[i]);
}
```

Figure 7.3: Simple first-touch initialization under OpenMP.

This idiom assumes an identical schedule of loop iterations in the initialization and computational loops. The first-touch policy places a memory page in memory attached to the socket of the processor that executes the thread that first accesses the page. If the data use in the computational loop iterations mirrors the data use in the initialization loop iterations and if we bind threads to processors with the `OMP_PROC_BIND` environment variable, memory accesses are local. We show an example of this pattern in Figure 7.3, where each thread initializes n/p elements of `data` and computes on those elements during each simulation step. This results in primarily local data accesses on NUMA systems with first-touch policy.

Figure 7.4 shows the analogous code for an OpenMP task parallel divide-and-conquer program with tree-structured data, which uses barriers to ensure that the initialization tasks complete before the compute tasks execute. Task parallelism in `recursive_init()` spreads the data across memory banks. However, we have no mechanism (not even a non-portable one) to guarantee that `recursive_compute()` task placement mirrors that of `recursive_init()`. Thus, a corresponding `recursive_compute()` task may be scheduled on a different thread that runs on a different socket, in which case we incur non-local accesses and, thus, work time inflation. The lack of a task scheduling analog is unfortunate, since task parallel computations allow easy expression of key computation patterns, such as oct-tree decompositions, for which a flat 1-D loop distribution is not well suited.

```
void recursive_init(data_t *data) {
   init(data);
   if (!is_leaf(data)) {
      #pragma omp task
         recursive_init(data->left);
      #pragma omp task
         recursive_init(data->right);
      #pragma omp taskwait
   }
}

void recursive_compute(data_t *data) {
   if (!is_leaf(data)) {
      #pragma omp task
         recursive_compute(data->left);
      #pragma omp task
         recursive_compute(data->right);
      #pragma omp taskwait
   }
   compute(data, data->left, data->right);
}

#pragma omp parallel
{
   #pragma omp single
      #pragma omp task
         recursive_init(top);
   #pragma omp barrier

   for (step = 0; step < nsteps; step++) {
      #pragma omp single
         #pragma omp task
            recursive_compute(top);
      #pragma omp barrier
   }
}
```

Figure 7.4: Analogous initialization for OpenMP tasks.

7.3 A Framework for Locality-Based Scheduling

Our framework for locality-based scheduling builds on the notion of a *locality domain* consisting of several components:

- Logically, one or more threads and associated storage;
- Physically, one or more cores and physical memory close to them (e.g., a multicore chip and its directly attached memory) to which the system maps the threads;
- An associated (set of) task queue(s) that are distinct from the queues associated with other locality domains;
- A unique integer identifier, its *locality domain identifier*.

The remainder of this section describes our extensions to OpenMP to support locality-based scheduling for task-based programs and our prototype implementation of those extensions.

7.3.1 A Concise API for Programmer-Specified Scheduling

We add the following API calls to OpenMP to allow the programmer to access information about locality domains and to specify placement of tasks on them.

- `omp_child_task_affinity(`*`locality_domain_id`*`)` sets an internal control variable (ICV) that indicates the locality domain on which the run time should place tasks that the currently executing task generates. The *`locality_domain_id`* identifier specifies the locality domain. This call overrides default placement of child tasks, which is the parent task's locality domain.
- `omp_get_num_locality_domains()` returns the total number of locality domains in the system.
- `omp_get_locality_domain_num()` returns the locality domain identifier on which the task is executing.

The code in Figure 7.5 shows how a programmer can use these calls to distribute tasks and data according to a predictable locality-based schedule for the tasks: [1]

The code above omits the `recursive_compute()` function, which is similar in structure to `recursive_init()`. If the run time presents four locality domains, then each iteration places the top task on locality domain 0, the first split tasks on locality domains 0 and 1, and the second split tasks on locality domains 0, 1, 2, and 3. We place each remaining task on the locality domain of the task that generates it. Thus, a thread in the same domain during initialization and during each simulation time step executes the subtree of tasks that each second split task generates. While this pattern is one possible task layout, our run time calls can also schedule more irregular and dynamic ones, such as those generated by adaptive methods.

7.3.2 Run Time Scheduling Policy and Implementation

The OpenMP specification (OpenMP Architecture Review Board, 2008) places few restrictions on the scheduling of tasks. Thus, implementers have flexibility in run time scheduler design and implementation. Our extensions naturally suit a scheduler design that has a logical queue per locality domain. Our implementation uses physically separate queues for each locality domain. A scheme in which all locality domains share a single central queue with each task marked for and dispatched to the appropriate locality domain is possible but is unlikely to scale well.

We need a scheduling policy between and within locality domains. In general, the API call `omp_child_task_affinity()` only specifies initial task placement, but the task could be migrated to another locality domain for load balancing purposes. However, such migrations may lead to non-local data accesses and, thus, work time inflation. Thus, we provide a *strict* mode that disallows migrations between locality domains although it does not preclude load balancing within each locality domain.

Our prototype implementation extends the Qthreads hierarchical task scheduler that we described in Chapter 6. Qthreads probes the hardware topology of the system on which it runs using one of several portable or architecture-specific libraries. We configure it to use hwloc (Broquedis et al., 2010b), a library with good support for x86 NUMA machines. Recall that using the Qthreads *MTS*

[1] The `log2` and `pow` functions actually require a cast to int, but we omit it for simplicity and readability.

```
void recursive_init(data_t *data, int nsplits) {
   init(data);
   if (!is_leaf(data)) {
      #pragma omp task
         recursive_init(data->left, nsplits-1));
      if(nsplits > 0) {
         int currDom = omp_get_locality_domain_num();
         int nextDom = currDom + pow(2, nsplits);
         omp_child_task_affinity(nextDom);
      }
      #pragma omp task
         recursive_init(data->right, nsplits-1));
      #pragma omp taskwait
   }
}

int ndomains = omp_get_num_locality_domains();
int nsplitsTotal = log2(ndomains);

#pragma omp parallel
{
   #pragma omp single
   {
      omp_child_task_affinity(0);
      #pragma omp task
         recursive_init(top, nsplitsTotal);
   }
   #pragma omp barrier

   for (step = 0; step < nsteps; step++) {
      #pragma omp single
      {
         omp_child_task_affinity(0);
         #pragma omp task
            recursive_compute(top, nsplitsTotal);
      }
      #pragma omp barrier
   }
}
```

Figure 7.5: A task parallel program using locality-based scheduling.

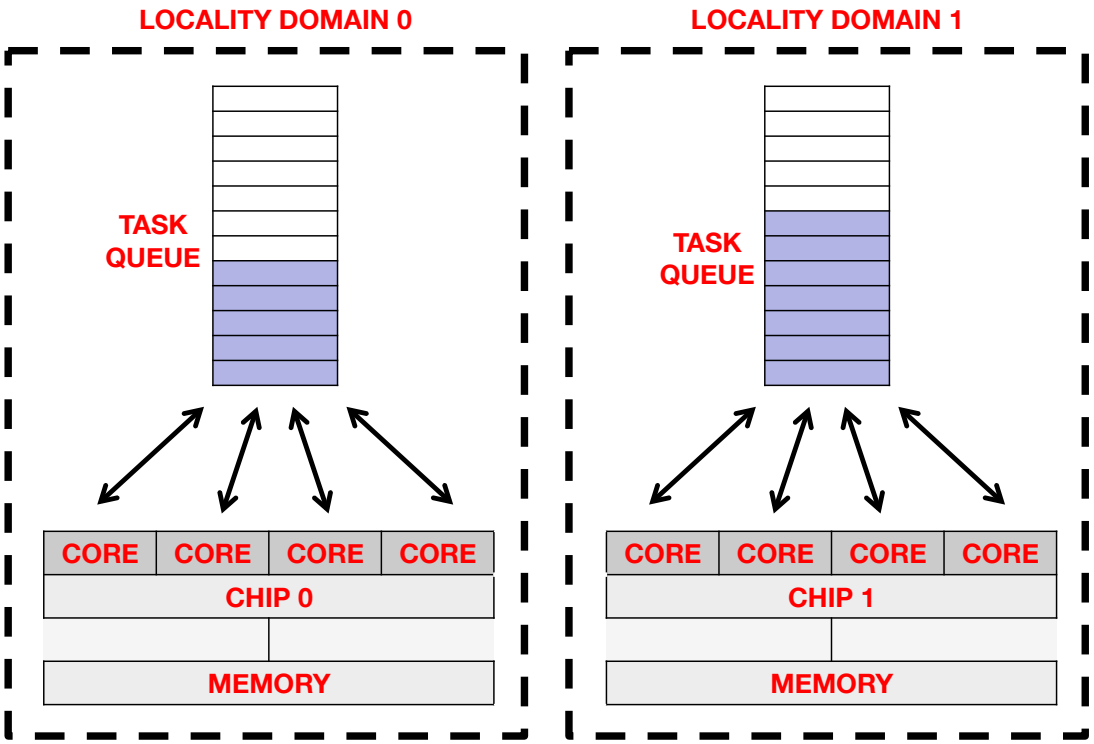


Figure 7.6: A mapping of locality domains to a two-socket system.

hierarchical scheduler, all threads that run on cores on the same chip share a task queue scheduled in a LIFO discipline, which promotes constructive L3 cache sharing among those threads and provides natural load balance among those threads. Load balancing between chips in *MTS* is accomplished by work stealing.

We map locality domains to the Qthreads representation of the NUMA topology and use per-socket shared queues, as shown in Figure 7.6. By default, when a task is generated, it is queued on the shared queue that the thread executing the parent task is using. However, if the child task affinity ICV of the parent task's ICV has been set to another locality domain, the new task is placed on that domain's queue. If the *strict* mode is set, then we disable work stealing between the task queues. Otherwise, tasks may migrate between the queues. The run time calls to return the total number of locality domains and the current locality domain map directly to existing Qthreads functions.

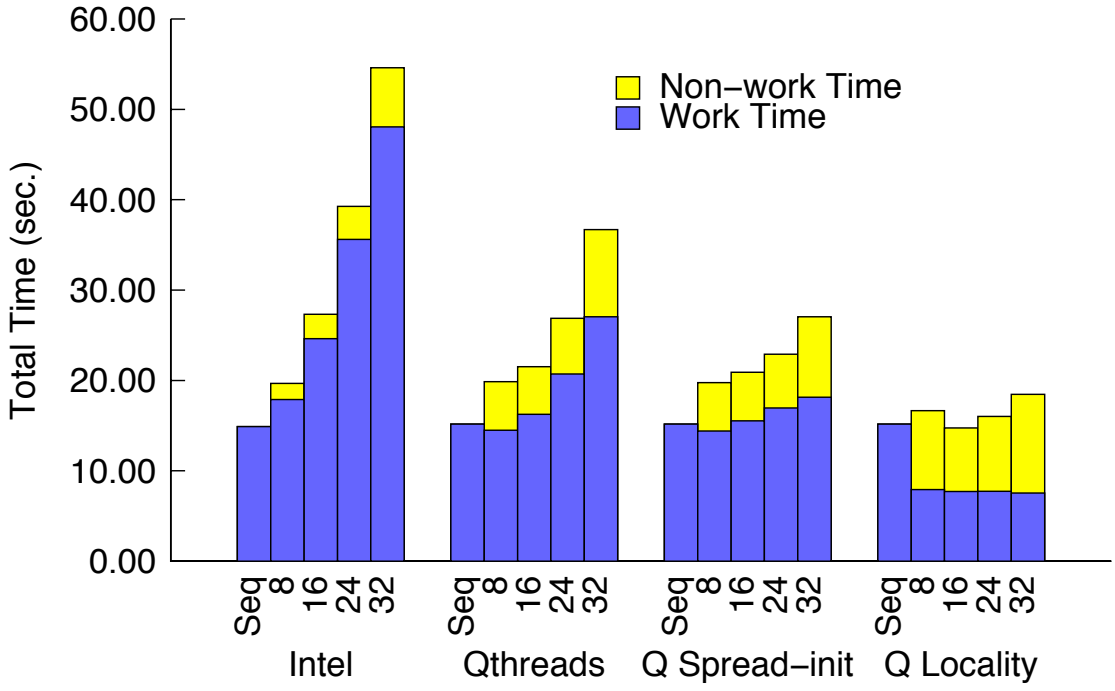


Figure 7.7: Total time over all threads for *Health*.

7.4 Evaluation

To evaluate our framework, we apply the techniques described in Section 7.3 to the example problems described in Section 7.1. The test system for our experiments is the four-socket Intel Nehalem-EX machine used in Chapter 6, the Dell PowerEdge M910 quad-socket blade with four Intel x7550 2.0GHz 8-core processors.

7.4.1 Performance and Speedup

We use the GNU C/C++ 4.4.4 compiler with $-O2$ optimization to obtain sequential times to compare against our framework, and as the native compiler for the code transformed by ROSE 0.9.5a. We use the Intel 11.1 compiler with $-O2\ -xHost\ -ipo$ optimization and the Intel OpenMP run time system for comparison. The difference in sequential time between the two compilers is negligible on the health simulation and more pronounced on the 2D heat diffusion simulation. For a comparison of parallel executions on absolute terms, we give a comparison of elapsed execution time

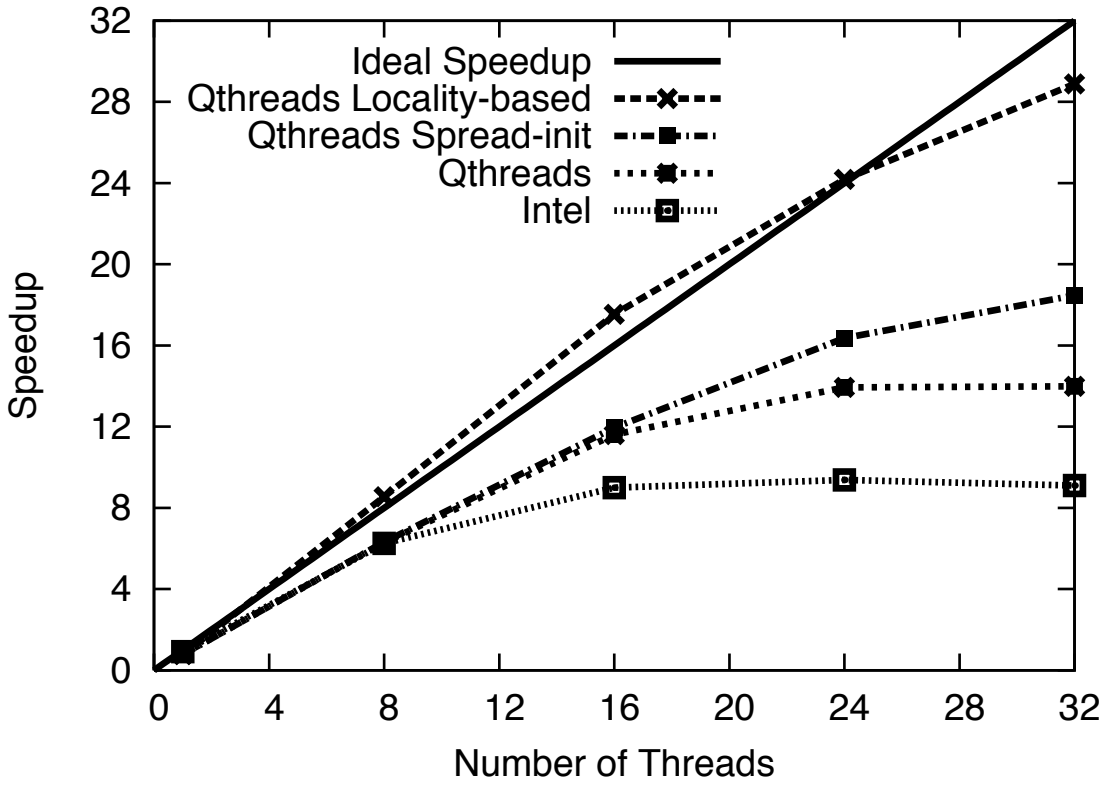


Figure 7.8: Speedup on *Health*.

for each application. Timing and speedup results represent the best of ten trials for each configuration, unless otherwise noted.

Figure 7.7 shows the execution time break down for the health simulation using several different configurations. The first set of bars shows results for executions using the locality-oblivious Intel run time, as we showed previously in Figure 7.1. The second set, labeled *Qthreads*, shows results obtained using the Qthreads *MTS* hierarchical scheduler with no locality-based scheduling. The sequential times for Intel and GCC are very close for this problem (14.9 and 15.2 seconds, respectively). The timing results for the parallel executions show more non-work time spent, i.e., combined idle and overhead time, compared to the Intel run time. However, the work time inflation is much lower for Qthreads (12 seconds), due to the hierarchical scheduler, than for Intel (33 seconds). The third set, labeled *Q Spread-init*, shows that we achieve further improvement by using our framework to spread the initialization of the program data across locality domains but without the use of locality-based scheduling in the simulation. This change further reduces work time inflation through better use of

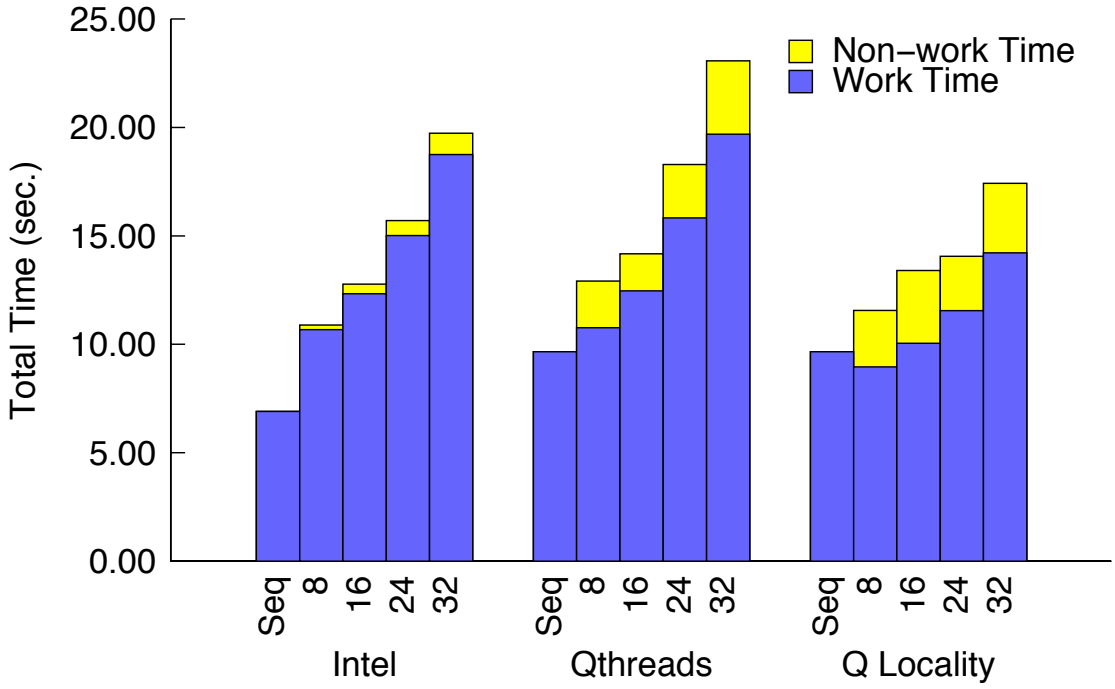


Figure 7.9: Total time over all threads for *Heat*.

Threads	8	16	24	32
Intel	2.46	1.71	1.64	1.69
Q Locality	1.83	0.890	0.646	0.540
% Decrease	25.6	48.0	60.6	68.0

Table 7.1: Run times (elapsed) on *Health*.

Threads	8	16	24	32
Intel	1.36	0.799	0.654	0.617
Q Locality	1.30	0.763	0.566	0.520
% Decrease	4.4	4.5	13.5	15.7

Table 7.2: Run times (elapsed) on *Heat*.

memory concurrency. The fourth set, *Q Locality*, uses the spread initialization and also locality-based scheduling in the *strict* mode to ensure that each task is scheduled on a thread in the locality domain to which the data that the task uses is associated. The full and contention-free use of all available cache and memory bandwidth without undue L3 cache invalidations results in work time equivalent to roughly half of the sequential time. The tradeoff is increased non-work time due to a lack of load balancing between locality domains.

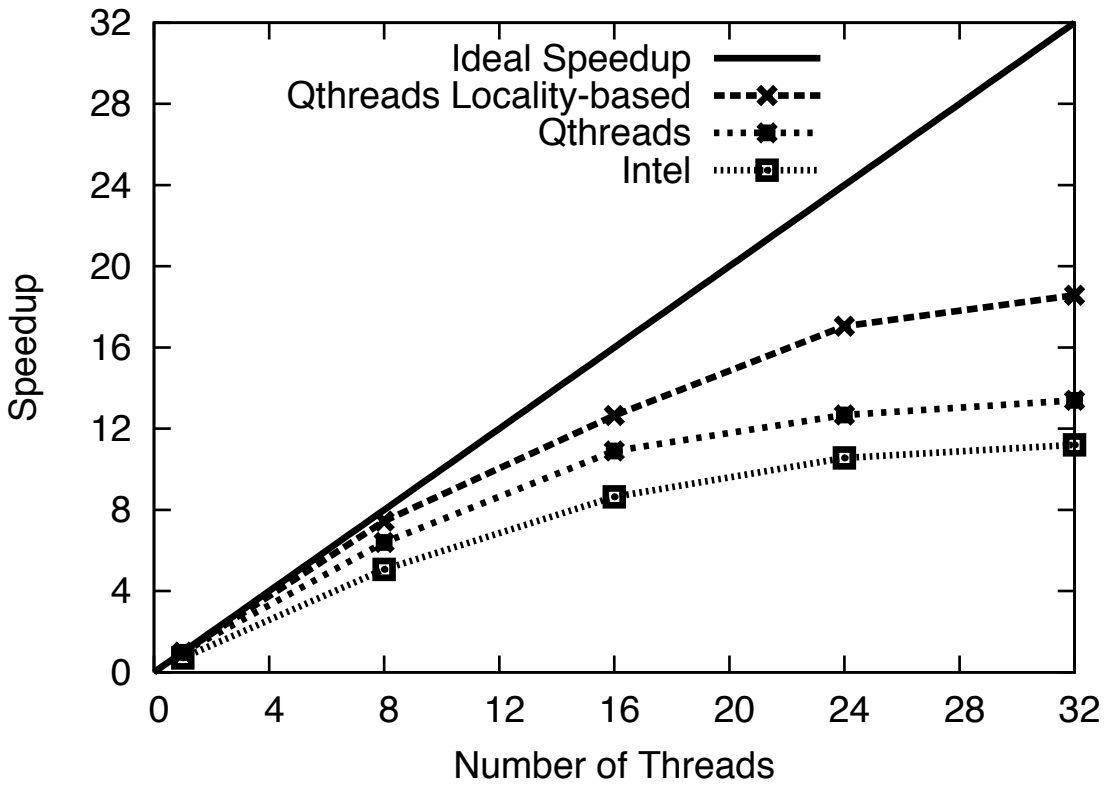


Figure 7.10: Speedup on *Heat*.

Figure 7.7 presents the sum of the total times spent by all threads in an execution. Alternatively, Figure 7.8 shows the parallel speedup achieved, using ICC and GCC sequential times for the Intel and Qthreads schedulers respectively, and the elapsed times for the parallel executions. The speedup using Intel flattens at 16 threads. The speedup using the Qthreads locality-oblivious scheduler is better, but still flattens at 24 threads. The use of spread initialization in Qthreads results in further improvement, but even it reaches only 18X speedup on 32 threads. Only the Qthreads locality-based scheduling achieves near-linear speedup, even exhibiting slightly super-linear speedup at 16 threads. Table 7.1 gives a comparison of the elapsed times for the Intel and locality-based schedulers.

Figure 7.9 shows a time break down for the 2D heat diffusion simulation. In this case, sequential execution is significantly faster using ICC (6.9 sec.) than GCC (9.7 sec.). Despite this gap, work times for parallel executions that use the same number of threads with Intel and the Qthreads locality-oblivious scheduler are close. Work time inflation on 32 threads is close to 100% using Qthreads and 170% using the Intel run time. Again, non-work time is higher using Qthreads. The heat simulation

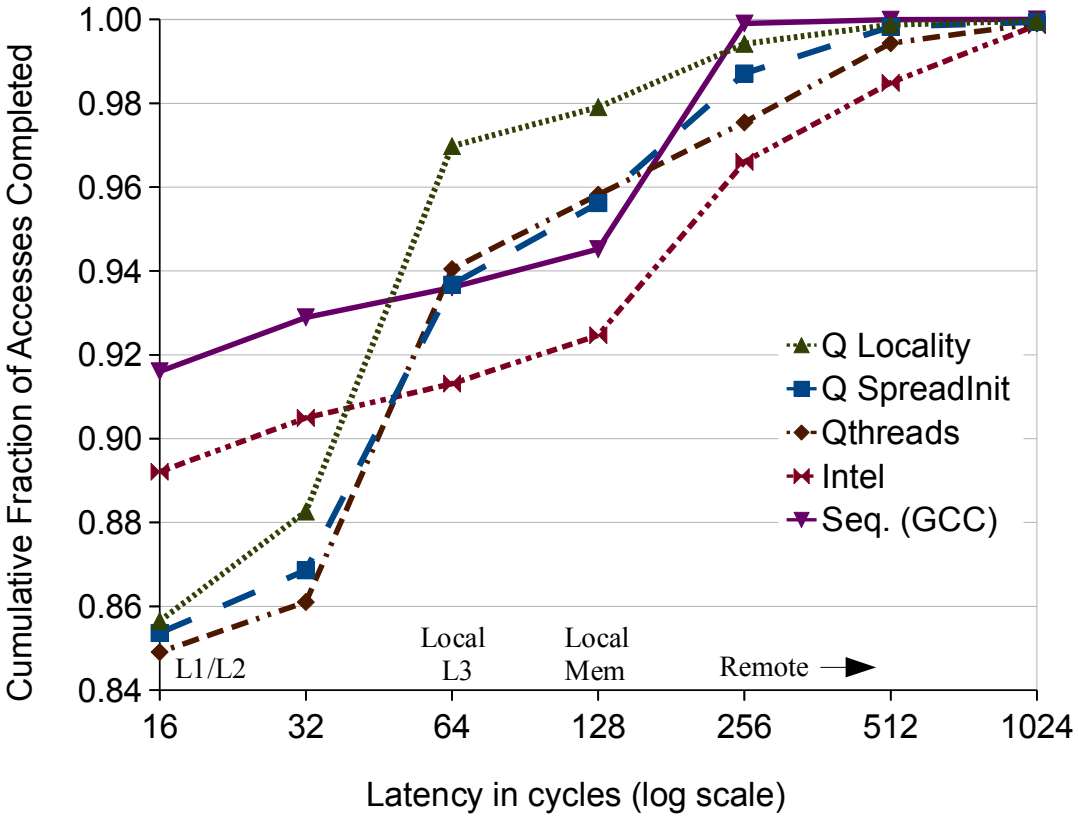


Figure 7.11: CDF showing data access latency for *Health*.

uses parallel initialization to spread the data, so the vehicle for further improved performance must be locality-based scheduling during the simulation to schedule tasks near the data that they access. We observe from the *Q Locality* results that this scheduling eliminates some but not all work time inflation, with similar non-work times. On 32 threads, it reduces work time inflation by more than half compared to the locality-oblivious scheduler. Since the heat simulation uses stencils that require exchanges of data at the boundaries of the grid, some non-local data is inevitably accessed. Thus, even with locality-based scheduling some work time inflation remains. Despite the difference in sequential times using GCC and ICC, the total execution times using locality-based scheduling are lower across the board, as shown in Table 7.2. The speedup improvements, which Figure 7.10 shows, are significant although none of the schedulers achieve ideal speedup.

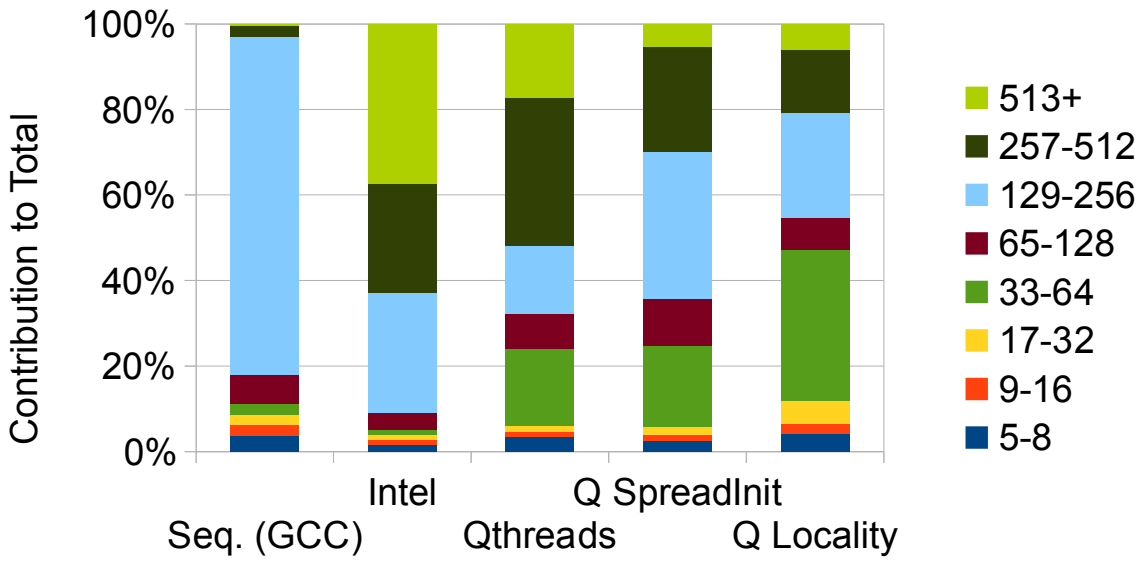


Figure 7.12: Ranges of load latencies (cycles) for *Health*.

7.4.2 Performance Counter Measurement

To obtain a more complete understanding of run time behavior, we used hardware counters to measure performance characteristics of executions that use locality-oblivious and locality-based scheduling. Since we posit that memory latency incurred on non-local accesses is the driving force behind work time inflation, we measured load latency during the execution of our two examples. This metric is measured using Intel's Precise Event Based Sampling (PEBS). Figure 7.11 plots a cumulative distribution function (CDF) for latency using each scheduler with 32 threads, along with results from a sequential execution. The interpretation is as follows: for a point (x, y) on the graph, y is the cumulative fraction of loads that have completed in x cycles or less. We observe that for all the schedulers, at least 85% of loads complete in 16 cycles or less. Micro-benchmarking studies of the Nehalem cache hierarchy and memory subsystem by Molka et al. (Molka et al., 2009) indicate that latencies in this range represent hits in the core's L1 and L2 caches. As we proceed across the x-axis, we note the regimes in which accesses represent on-chip L3 cache hits (and hits in L2 caches of cores on the same chip), accesses to memory on the same socket, and finally, accesses to remote L3 caches and memory on other sockets.

Starting from the left side of the graph, the first difference between the results for the different schedulers is that the sequential execution best uses the per-core L1 and L2 caches. Since multi-

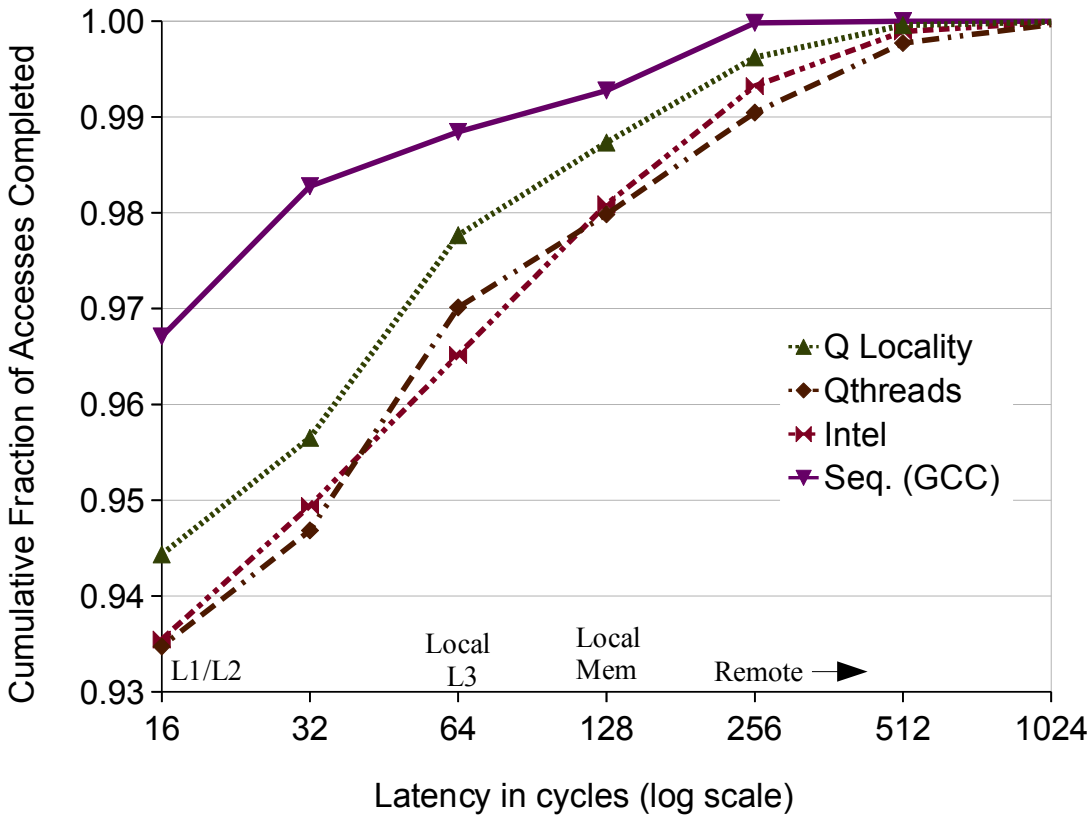


Figure 7.13: CDF showing data access latency for *Heat*

threaded executions introduce coherence misses, we expect this phenomenon. Moreover, threads on the same chip in the Qthreads schedulers share a queue, and shared queues are cache-friendly for shared caches, e.g., the Nehalem L3, but not but individual caches, e.g., L1 and L2. Although we cannot examine its run time source code, we suspect that Intel uses per-core work stealing, which is cache-friendly for individual caches. The benefit of the per-chip shared queues to exploit the shared L3 caches manifests in the sharp increase in accesses completed by 64 cycles using Qthreads. The locality-aware scheduler achieves a higher number of cumulative completions in this range than even the sequential execution, as it effectively exploits the full L3 cache of the machine with minimal coherence misses between sockets. This scheduler also outperforms Intel's and sequential executions in the range of local memory accesses by engaging the full memory bandwidth of the machine with minimal accesses to memory on remote sockets. The data points at 256 cycles and beyond represent remote memory accesses. The sequential execution, since it is only executed on a single core on a single socket, only shows a negligible number of such remote memory accesses, which can be

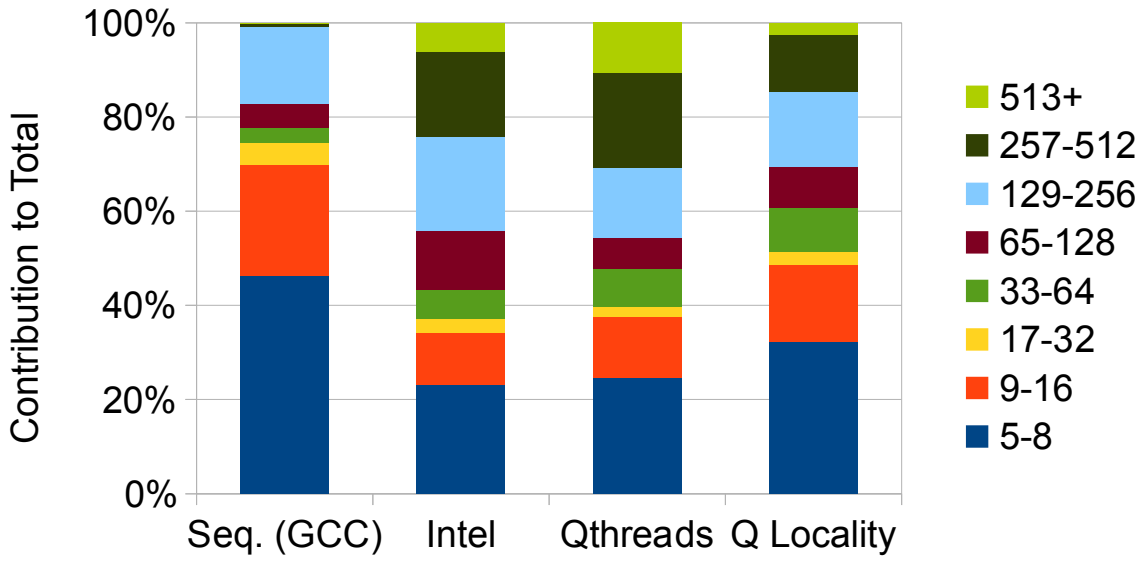


Figure 7.14: Ranges of load latencies (cycles) for *Heat*.

attributed to I/O or system processes. The completion rates of the Qthreads and Intel schedulers line up in the same order as the speedup graphs, led by the locality-based scheduler. However, can the remaining accesses, which account for less than 2% of the total number of accesses sampled, really matter?

The relative impact of accesses in each range is a function of not only the number of accesses in the range, but also the number of cycles per access in that range. For example, one access that requires 256 cycles to complete contribute the same number of cycles to the total as 16 accesses of 16 cycles. Figure 7.12 shows the relative contribution of the different ranges of access times, using a conservative estimate: number of accesses $times$ the low end of the range, e.g., assume all accesses in the range of 33-64 cycles take 33 cycles. The vast majority of access time in the sequential execution is spent on loads in the local memory regime (129-256). For the locality-oblivious schedulers, accesses of greater than 256 cycles in duration account for almost half of the total latency using Qthreads and over 60% using Intel. In addition to low contributions to total latency from high latency loads, locality-based scheduling also shows a much higher contribution of L3 regime loads (about 40%) than any of the other schedulers, and again betters the sequential execution in that range.

Figure 7.13 shows a CDF of load latency sampled during executions of the 2D heat diffusion simulation. The sequential execution experiences the lowest latency in all regimes. While the

	Seq. (gcc)	Intel	Qthreads	Q Locality
Health	0.067	34	26	1.0
Heat	0.077	31	31	0.34

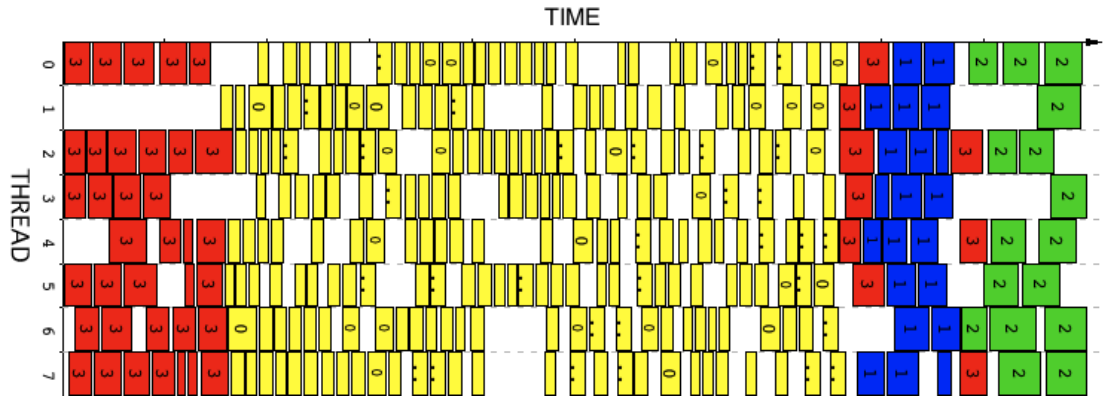
Table 7.3: Data Transferred (GB) over QPI between sockets during sequential and 32-thread executions.

latencies incurred with locality-based scheduling are much higher, they are still significantly lower than those incurred with the locality-oblivious schedulers (Qthreads and Intel). This observation is consistent with our speedup results: locality-based scheduling outperforms locality-oblivious scheduling but falls short of ideal speedup. Figure 7.14 shows the relative contribution of the different latency ranges. A marked difference between the health and heat simulations is evident just from comparing the sequential results: Contributions from L1 and L2 accesses are less than 10% in executions of the heath simulation but 70% in the executions of the heat simulation. The locality-based Qthreads scheduler sees a lower contribution of high latency loads then the locality-oblivious schedulers.

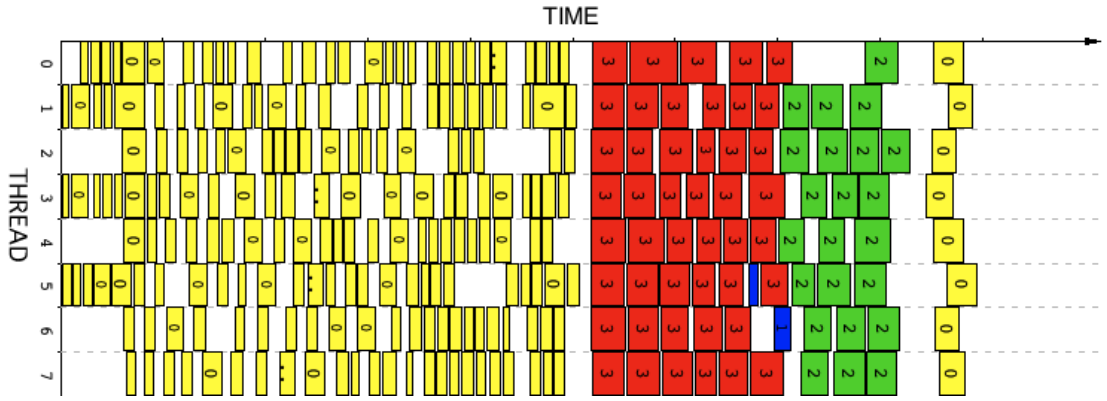
Another interesting metric is QPI traffic, i.e., the amount of data transferred between sockets during execution. Table 7.3 shows the measured volumes in gigabytes. The interconnect is quiet during sequential execution, as we might expect. The significant observation is the order of magnitude reduction in QPI traffic from the locality-oblivious schedulers of Intel and Qthreads to the locality-based Qthreads scheduler.

7.4.3 Visualizing Observed Task Schedules

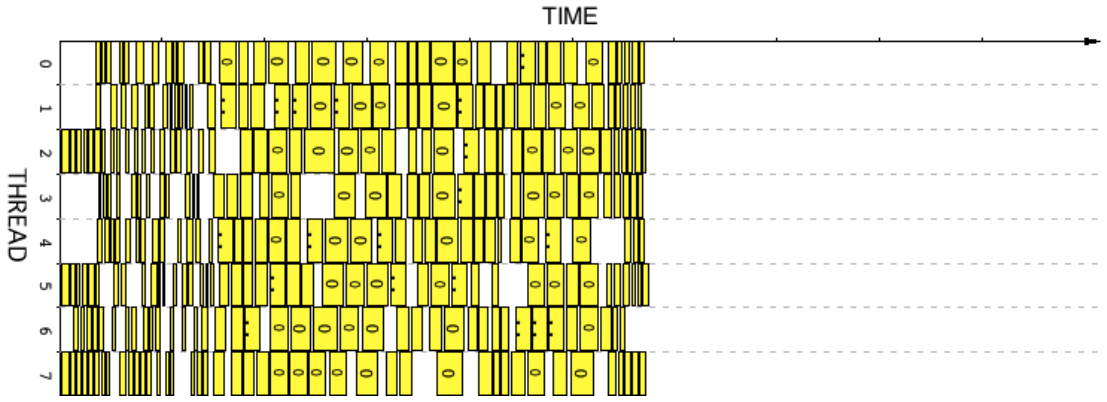
Instrumentation to log actual schedules gives a micro-level view of scheduler behavior to allow a detailed evaluation of the impact of scheduler decisions. We use the Jedule tool (Hunold et al., 2010) to generate visualizations of task schedules, as Figure 7.15 shows. The three recorded partial schedules are taken from 32 core executions of the health simulation using three different schedulers, with a uniform time axis spanning about 3.1 milliseconds. In each execution, we record the start and end times of each leaf task in the 100th iteration of the simulation loop, along with the thread number on which the task executes and the locality domain in which the data it uses is located. The vertical axis represents the threads (for the first eight of 32 threads) and the horizontal axis represents time. The threads shown (threads 0-7) are all tied to cores on the chip in socket 0. Each



(a) Locality-oblivious scheduling



(b) Non-strict locality-based scheduling



(c) Strict locality-based scheduling

Figure 7.15: Observed schedules of tasks over time on 8 threads on the same chip.

box is a task. The numbers and colors on each task indicate the number of the socket that is directly linked to the memory bank that contains the data used in that task: yellow, blue, green, and red for sockets 0, 1, 2, and 3, respectively. Thus, the tasks numbered 0 and colored yellow use data that resides in memory attached to socket 0 and perform only local memory and cache accesses when executed on threads 0-7. The tasks bearing other numbers and colors use data local to the other three sockets (not shown in the diagram) and the other threads (8-15, 16-23, and 24-31, respectively).

Figure 7.15(a) illustrates locality-oblivious scheduling. In the early part of the execution, the threads in this locality domain work on tasks that access data that is local to locality domain 3. Later in the execution, tasks that access local data are stolen. These tasks generally execute more quickly, as indicated by the smaller boxes. Eventually, the threads steal more tasks that access data in locality domains 1, 2, and 3.

Figure 7.15(b) shows a schedule from locality-based scheduling in *non-strict* mode, i.e., with work stealing allowed between locality domains. This schedule completes in slightly less time than the locality-oblivious schedule. Initially, the threads execute tasks that access data local to the locality domain. Just more than half way through, a thread on another locality domain steals the remaining work, and in turn thread 4 steals tasks from locality domains 3 and 2, and a couple from 1. This behavior occurs due to instantaneous work imbalances that occur even in computations that are on the whole balanced, especially in the later stages of execution. Near the end of the computation, thread 3 reacquires some tasks that access local data, but they require more time now, because their data has been cached in the L3 caches in the other locality domains and must now be invalidated.

Strict locality-based scheduling, shown in Figure 7.15(c), results in a much faster execution. Threads 0-7 only execute tasks that access local data. When a thread goes idle and no tasks are on the queue, the thread is forbidden to steal from other locality domains and must wait for one of the other threads in its own locality domain to generate more work. The short tasks near the beginning and end of the execution indicate another benefit of executing only local tasks: better locality between the simulation steps. Local caches are not polluted by non-local data, and, conversely, local data does not end up in remote caches. In contrast, with locality-oblivious scheduling, and even *non-strict* locality-based scheduling, each simulation step can begin with a significant amount of interconnect cross-traffic and cache invalidations.

7.5 Summary

The locality-based task scheduling framework that we propose comprises the concept of distinct locality domains, programmer annotations to application source code, and a scheduler that honors those annotations. The example applications to which we applied the model in our evaluation have important similarities and differences. As simulation codes, both applications have significant data reuse across iterations of the outer simulation loop. In both cases, much of their work time inflation is attributable to non-local data accesses. For both applications, global dynamic load imbalance is a much smaller contributor to performance loss than work time inflation, enabling effective use of the *strict* scheduling mode. In the health simulation, data accesses are almost completely confined to the bounds of the locality domain. This access pattern enables a doubling of speedup using locality-based scheduling. In the heat simulation, some data must be shared between the locality domains when the stencil is applied to points near the boundary of the subspaces assigned to two locality domains. Accordingly, a smaller performance improvement is observed on *Heat* than on *Health*.

Applicability. Locality-based scheduling does not improve performance of all task parallel applications. The BOTS *Sort* benchmark is memory-bandwidth bound and performs unavoidable data exchanges on the whole input. For such an application, essentially no scheduling strategy can improve performance beyond the limits of memory concurrency on the target machine (Mandal et al., 2010). For these applications, and those limited by similar shared resource constraints, e.g., I/O or network bandwidth, reducing the number of worker pthreads on each locality domain and powering down the cores on which they run may enable power savings without degrading performance. Applications with little data reuse, such as UTS and the BOTS *Fib* benchmark, do not benefit from locality-based scheduling because the proximity of tasks to local data is not a primary concern compared to other issues such as overheads and load imbalance.

Limitations and Future Work. In *Health* and *Heat*, the assignment of tasks to locality domains follows the same pattern in each iteration. In our future work, we will consider applications in which tasks are assigned dynamically to locality domains as the simulation progresses. Hybrid Reciprocal Velocity Obstacle (HRVO) is an algorithm for multi-agent navigation designed to minimize collisions and oscillation (Snape et al., 2011). In this problem, n agents are required to navigate between individual start and end positions on the plane. Each individual agent acts independently to reach

its goal and avoid other agents in its environment. In each time step of the simulation, each agent determines which other agents are near enough to influence its choice of velocity, calculates a new velocity based on the HRVO algorithm, updates its velocity and position, and determines if the goal has been reached.

Consider a 2D decomposition of the plane into a grid, enabling a quick division along the x and y axes to assign grid cells to locality domains. Each agent belongs to the grid cell corresponding to its position. In each time step, we create a task for each agent to compute its new velocity based on the velocities of nearby agents. Then a synchronization ensures that all agents have finished computing their new velocities before a second task is created for each agent to update its velocity and position. In a locality-oblivious implementation, tasks are scheduled onto threads according to the scheduling policy of the run time system. In a locality-based implementation, each task is assigned to the locality domain corresponding to its grid cell. The tasks corresponding to each agent in different iterations of the simulation loop are reassigned to different locality domains when the agent crosses from one quadrant to another in that time step.

Applications such as the adaptive Fast Multipole Algorithm (FMA) for n-body simulation have a changing and unbalanced locus of work that requires global dynamic load balancing. The imbalance in such applications precludes the use of the *strict* scheduling mode, and the assignment of subgraphs of the task graph to locality domains may be irregular and changing. Future work will extend our framework to overcome these limitations. A weighting function could be devised to control stealing based on the trade-off between load balance and locality for each application. Alternatively, particular tasks could be reassigned between simulation iterations to different locality domains than surrounding tasks to improve load balance and respond to a shift in the work.

CHAPTER 8

CONCLUSIONS AND FUTURE DIRECTIONS

In defense of our thesis, the preceding chapters present our case that schedulers using hardware topology information and user-specified information about application locality can improve the performance of task parallel applications. Our analysis of load balancing overheads and work time inflation illustrates the impact of those issues. Our *MTS* hierarchical scheduler provides transparent performance improvements by limiting remote load balancing applications and exploiting shared caches. Our locality-based scheduling framework, while nontransparent, provides a concise programmer API to specify explicit task placement, enabling performance improvements in applications in which remote data accesses are the primary sources of performance loss and data reuse is significant. Our implementation of these schedulers in the Qthreads run time allows comparison with prior scheduling policies and production run time systems. These evaluations using OpenMP applications and actual hardware demonstrate the performance improvements possible with hierarchical and locality-based scheduling.

8.1 Future Directions

This dissertation has focused on scheduling task parallelism on shared memory NUMA systems. Locality and overhead costs can have an even higher impact on distributed memory systems, where network latency costs are incurred for communication between nodes. We discussed several task parallel languages and libraries for distributed memory in Chapter 2. Languages such as Chapel and X10 would be most suited for a distributed memory adaptation of our scheduling methods because they present a Partitioned Global Address Space (PGAS). While developing a scalable distributed memory implementation of the UTS benchmark with user-level work stealing in UPC, we introduced several strategies for dynamic load balancing, including efficient stealing protocols and termination

detection (Olivier and Prins, 2008). We also observed a benefit from bulk stealing, in common with the *MTS* scheduler. To schedule tasks on a cluster of multicore nodes, hierarchical and locality-based scheduling could be extended to include a new top level of the hierarchy differentiating between cluster nodes.

Graphics Processing Units (GPUs) and other accelerators have become key components of HPC systems. Latency costs to transfer data across the bus between the host processor (CPU) and accelerator can cause major performance bottlenecks. As part of the trend of relaxing constraints on accelerator programming, individual words in host memory are now accessible by pointers in GPU code where explicit copies were previously required. This system-managed pointer access effectively makes the CPU and GPU NUMA nodes in a shared memory system from the application programmer's point of view. Hierarchical scheduling and locality-based scheduling may improve performance on these systems for applications structured to use task parallelism on the CPUs with leaf tasks that execute on the GPU. Some GPUs also support the creation of new tasks on the GPU, so hierarchical scheduling and locality-based scheduling could be extended onto the GPU itself.

8.2 The Big Picture

This dissertation reflects the reality that parallel computing is an exercise in balancing elegance and performance. Existing task scheduling techniques, while elegant and sound in theory, assume a simple model of the memory hierarchy that does not match the complexity of multicore NUMA systems. Our schedulers incorporate some of that complexity into the run time system. The use of only minimal language constructs, e.g., `task` and `taskwait` or `spawn` and `sync`, to express parallelism in the task parallel programming model underscores its elegance, as well its accessibility. However, it provides little information to the run time system about data usage among the tasks. Our concise API for explicit tasks placement adds slightly to the complexity of the application code but enables the users to provide information to enable locality-based scheduling. Beyond this particular work, we posit that the trade-off between elegance and complexity will be a recurring theme of parallel computing research well into the future.

BIBLIOGRAPHY

Acar, U. A., Blelloch, G. E., and Blumofe, R. D. (2000). The data locality of work stealing. In *SPAA '00: Proc. 12th ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12. ACM.

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701.

Agarwal, S., Barik, R., Bonachea, D., Sarkar, V., Shyamasundar, R. K., and Yelick, K. (2007). Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA '07: Proc. 19th ACM symposium on Parallel Algorithms and Architectures*, pages 229–240. ACM.

Alverson, G. A., Alverson, R., Callahan, D., Koblenz, B., Porterfield, A., and Smith, B. J. (1992). Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *ICS '92: Proc. 6th ACM Intl. Conference on Supercomputing*, pages 188–197. ACM.

AMD Inc. (2006). Performance guidelines for AMD Athlon(TM) 64 and AMD Opteron(TM) cc-NUMA multiprocessor systems. http://support.amd.com/us/Processor%5FTechDocs/40555.pdf.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proc. 1967 Spring Joint Computer Conference*, pages 483–485. ACM.

Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. (1991). Scheduler activations: Effective kernel support for the user-level management of parallelism. In *SOSP '91: Proc. 13th ACM Symposium on Operating Systems Principles*, pages 95–109. ACM.

Arora, N. S., Blumofe, R. D., and Plaxton, C. G. (1998). Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proc. 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129. ACM.

Arora, N. S., Blumofe, R. D., and Plaxton, C. G. (2001). Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144.

Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418.

Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., and Teruel, X. (2007). An experimental evaluation of the new OpenMP tasking model. In Adve, V. S., Garzarán, M. J., and Petersen, P., editors, *Language and Compilers for Parallel Computers (LCPC)*, volume 5234 of *Lecture Notes in Computer Science*, pages 63–77. Springer.

Baldeschwieler, J. E., Blumofe, R. D., and Brewer, E. A. (1996). ATLAS: An infrastructure for global computing. In *EW 7: Proc. 7th ACM SIGOPS European Workshop*, pages 165–172. ACM.

Blelloch, G. E. and Gibbons, P. B. (2004). Effectively sharing a cache among threads. In *SPAA '04: Proc. 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244. ACM.

Blelloch, G. E., Gibbons, P. B., and Matias, Y. (1999). Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1996). Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69.

Blumofe, R. D. and Leiserson, C. E. (1998). Space-efficient scheduling of multithreaded computations. *SIAM Journal of Computing*, 27(1):202–229.

Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46:720–748.

Blumofe, R. D. and Lisiecki, P. A. (1997). Adaptive and reliable parallel computing on networks of workstations. In *ATEC '97: Proc. USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA. USENIX Association.

Blumofe, R. D. and Papadopoulos, D. (1998). The performance of work stealing in multiprogrammed environments. Technical Report TR-98-13, Department of Computer Science, University of Texas at Austin, Austin, TX, USA.

Bolosky, W. J., Scott, M. L., Fitzgerald, R. P., Fowler, R. J., and Cox, A. L. (1991). NUMA policies and their relation to memory architecture. In *ASPLOS '91: Proc. 4th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221. ACM.

Brent, R. P. (1974). The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206.

Broquedis, F., Aumage, O., Goglin, B., Thibault, S., Wacrenier, P.-A., and Namyst, R. (2010a). Structuring the execution of OpenMP applications for multicore architectures. In *IPDPS 2010: Proc. 25th IEEE Intl. Parallel and Distributed Processing Symposium*. IEEE.

Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010b). hwloc: A generic framework for managing hardware affinities in HPC applications. In *PDP 2010: Proc. 18th Euromicro Intl. Conference on Parallel, Distributed and Network-Based Processing*, pages 180–186, Pisa, Italia. IEEE Computer Society.

Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., and Taşirlar, S. (2010). Concurrent collections. *Scientific Programming*, 18:203–217.

Cave, V., Zhao, J., Shirako, J., and Sarkar, V. (2011). Habanero-Java: The new adventures of old X10. In *PPPJ 2011: 9th Intl. Conference on the Principles and Practice of Programming in Java*. ACM.

Chamberlain, B., Callahan, D., and Zima, H. (2007). Parallel programmability and the Chapel language. *Intl. Journal of High Performance Computing Applications*, 21(3):291–312.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proc. 20th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 519–538. ACM.

Chase, D. and Lev, Y. (2005). Dynamic circular work-stealing deque. In *SPAA '05: Proc. 17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28. ACM.

Chen, S., Gibbons, P. B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G. E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T. C., and Wilkerson, C. (2007). Scheduling threads for constructive cache sharing on CMPs. In *SPAA '07: Proc. 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115. ACM.

Dinan, J., Krishnamoorthy, S., Larkins, D. B., Nieplocha, J., and Sadayappan, P. (2008). Scioto: A framework for global-view task parallelism. In *ICPP '08: Proc. 37th Intl. Conference on Parallel Processing*, pages 586–593. IEEE.

Dinan, J., Larkins, D. B., Sadayappan, P., Krishnamoorthy, S., and Nieplocha, J. (2009). Scalable work stealing. In *SC09: ACM/IEEE Supercomputing 2009*, pages 1–11. ACM.

Duran, A., Corbalán, J., and Ayguadé, E. (2008a). An adaptive cut-off for task parallelism. In *SC08: ACM/IEEE Supercomputing 2008*, pages 1–11. IEEE.

Duran, A., Corbalán, J., and Ayguadé, E. (2008b). Evaluation of OpenMP task scheduling strategies. In Eigenmann, R. and de Supinski, B. R., editors, *IWOMP '08: Proc. Intl. Workshop on OpenMP*, volume 5004 of *Lecture Notes in Computer Science*, pages 100–110. Springer.

Duran, A. and Teruel, X. (2011). Barcelona OpenMP Tasks Suite (BOTS) Benchmark Project. http://nanos.ac.upc.edu/projects/bots.

Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguadé, E. (2009). Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP '09: Proc. 38th Intl. Conference on Parallel Processing*, pages 124–131. IEEE.

Eager, D., Zahorjan, J., and Lazowska, E. (1989). Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423.

Eastlake, D. and Jones, P. (2001). US Secure Hash Algorithm 1 (SHA-1). RFC 3174, Internet Engineering Task Force.

Faxén, K.-F. (2009). Wool - A work stealing library. *SIGARCH Computer Architecture News*, 36(5):93–100.

Feng, M. and Leiserson, C. E. (1997). Efficient detection of determinacy races in Cilk programs. In *SPAA '97: Proc. 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 1–11. ACM.

Friedman, D. and Wise, D. (1978). Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289 –296.

Frigo, M., Halpern, P., Leiserson, C. E., and Lewin-Berlin, S. (2009). Reducers and other Cilk++ hyperobjects. In *SPAA '09: Proc. 21st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90. ACM.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-oblivious algorithms. In *FOCS '99: Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE.

Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223. ACM.

Funk, M. and Peterson, R. (2010). Of NUMA on POWER7 in IBM i. *IBM Performance Management Resource Library*, http://www.ibm.com/systems/resources/pwrsysperf%5FP7NUMA.pdf.

Fürlinger, K. and Skinner, D. (2009). Performance profiling for OpenMP tasks. In *IWOMP '09: Proc. 5th Intl. Workshop on OpenMP*, volume 5568 of *Lecture Notes in Computer Science*, pages 132–139, Berlin, Heidelberg. Springer.

Gautier, T., Besseron, X., and Pigeon, L. (2007). Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07: Proc. 2007 Intl. Workshop on Parallel Symbolic Computation*, pages 15–23. ACM.

Goldstine, H. H. and Goldstine, A. (1946). The electronic numerical integrator and computer (ENIAC). *Mathematics of Computation*, 2(15):97–110.

Guo, Y., Barik, R., Raman, R., and Sarkar, V. (2009). Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS '09: Proc. 2009 IEEE Intl. Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE.

Guo, Y., Zhao, J., Cave, V., and Sarkar, V. (2010). SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 341–342. ACM.

Halstead, Jr., R. H. (1985). MULTILISP: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538.

Harris, T. (1963). *The Theory of Branching Processes*. Springer.

He, Y., Leiserson, C. E., and Leiserson, W. M. (2010). The Cilkview scalability analyzer. In *SPAA '10: Proc. 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 145–156. ACM.

Hendler, D., Lev, Y., Moir, M., and Shavit, N. (2006). A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18:189–207.

Huang, L., Jin, H., Yi, L., and Chapman, B. M. (2010). Enabling locality-aware computations in OpenMP. *Scientific Programming*, 18(3-4):169–181.

Hunold, S., Hoffmann, R., and Suter, F. (2010). Jedule: A tool for visualizing schedules of parallel applications. In *PSTI 2010: Proc. First Intl. Workshop on Parallel Software Tools and Tool Infrastructures, in conjunction with 39th Intl. Conference on Parallel Processing*, pages 169–178. IEEE Computer Society.

IEEE (1995). Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE.

Intel Corp. (2010). Intel Cilk Plus™. http://software.intel.com/en-us/articles/intel-cilk-plus/.

Intel Corp. (2011). Intel VTune™. http://www.intel.com/software/products/vtune.

Johnson, T., Davis, T. A., and Hadfield, S. M. (1996). A concurrent dynamic task graph. *Parallel Computing*, 22:327–333.

Kambadur, P., Gupta, A., Ghoting, A., Avron, H., and Lumsdaine, A. (2009). PFunc: modern task parallelism for modern high performance computing. In *SC09: ACM/IEEE Supercomputing 2009*, pages 43:1–43:11. ACM.

Kleinrock, L. (1976). *Queueing Systems, Volume II: Computer Applications*. Wiley Interscience.

Kukanov, A. and Voss, M. (2007). The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4).

Kumar, S., Hughes, C. J., and Nguyen, A. D. (2007). Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In Tullsen, D. M. and Calder, B., editors, *ISCA '07: 34th Intl. Symposium on Computer Architecture*, pages 162–173. ACM.

LaGrone, J., Aribuki, A., Addison, C., and Chapman, B. M. (2011). A runtime implementation of OpenMP tasks. In Chapman, B. M., Gropp, W. D., Kumaran, K., and Müller, M. S., editors, *IWOMP'11: 7th Intl. Workshop on OpenMP*, volume 6665 of *Lecture Notes in Computer Science*, pages 165–178. Springer.

Lea, D. (2000). A Java fork/join framework. In *JAVA '00: Proc. ACM Conference on Java Grande*, pages 36–43. ACM.

Leijen, D., Schulte, W., and Burckhardt, S. (2009). The design of a task parallel library. *SIGPLAN Notices: OOPSLA '09: 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, 44(10):227–242.

Leskovec, J., Kleinberg, J., and Faloutsos, C. (2005). Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proc. 11th ACM SIGKDD Int'l Conf. Know. Disc. Data Mining (KDD '05)*, pages 177–187.

Levinthal, D. (2009). Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors (Intel Corp. White paper). http://software.intel.com/ sites/products/collateral/hpc/vtune/performance%5Fanalysis%5Fguide.pdf.

Liao, C., Quinlan, D. J., Panas, T., and de Supinski, B. R. (2010). A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In Sato, M., Hanawa, T., Müller, M. S., Chapman, B. M., and de Supinski, B. R., editors, *IWOMP 2010: Proc. 6th Intl. Workshop on OpenMP*, volume 6132 of *Lecture Notes in Computer Science*, pages 15–28. Springer.

Lin, Y. and Mazurov, O. (2009). Providing observability for OpenMP 3.0 applications. In *IWOMP '09: Proc. 5th Intl. Workshop on OpenMP*, volume 5568 of *Lecture Notes in Computer Science*, pages 104–117. Springer.

Mandal, A., Fowler, R., and Porterfield, A. (2010). Modeling memory concurrency for multi-socket multi-core systems. In *ISPASS 2010: IEEE Intl. Symposium on Performance Analysis of Systems and Software*, pages 66 – 75. IEEE Computer Society.

Marathe, J. and Mueller, F. (2006). Hardware profile-guided automatic page placement for ccNUMA systems. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–99.

Michael, M. M. (2003). CAS-based lock-free algorithm for shared deques. In Kosch, H., Böszörményi, L., and Hellwagner, H., editors, *Euro-Par 2003: Proc. 9th Euro-Par Conference on Parallel Processing*, volume 2790 of *LNCS*, pages 651–660. Springer.

Molka, D., Hackenberg, D., Schone, R., and Muller, M. (2009). Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *PACT '09. 18th Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 261 –270.

Narlikar, G. J. (1999). Scheduling threads for low space requirement and good locality. In *SPAA '99: Proc. 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 83–95. ACM.

Narlikar, G. J. and Blelloch, G. E. (1998). Pthreads for dynamic and irregular parallelism. In *SC98: 1998 ACM/IEEE Conference on Supercomputing*, pages 1–16. IEEE.

Nikolopoulos, D. S., Artiaga, E., Ayguadé, E., and Labarta, J. (2001). Exploiting memory affinity in OpenMP through schedule reuse. *SIGARCH Computer Architecture News*, 29(5):49–55.

Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., and Tseng, C.-W. (2007). UTS: An unbalanced tree search benchmark. In Almási, G., Cascaval, C., and Wu, P., editors, *LCPC 2006: Proc. 19th Intl. Workshop on Languages and Compilers for Parallel Computing*, volume 4382 of *LNCS*, pages 235–250. Springer.

Olivier, S. and Prins, J. (2008). Scalable dynamic load balancing using UPC. In *ICPP '08: Proc. 37th Intl. Conference on Parallel Processing*, pages 123–131. IEEE.

Olivier, S. L., Porterfield, A. K., Wheeler, K. B., and Prins, J. F. (2011). Scheduling task parallelism on multi-socket multicore systems. In *ROSS '11: Proc. Intl. Workshop on Runtime and Operating Systems for Supercomputers*, pages 49–56. ACM.

Olivier, S. L. and Prins, J. F. (2009). Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In Muller, M. S., de Supinski, B. R., and Chapman, B. M., editors, *IWOMP '09: Proc. 5th Intl. Workshop on OpenMP*, volume 5568 of *Lecture Notes in Computer Science*, pages 63–78, Berlin, Heidelberg. Springer.

Olivier, S. L. and Prins, J. F. (2010). Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs. *Intl. Journal of Parallel Programming*, 38(5-6):341–360.

Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. (1996). The case for a single-chip multiprocessor. In *ASPLOS-VII: Proc. 7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM.

OpenMP Architecture Review Board (2008). OpenMP API, Version 3.0.

Porterfield, A., Fowler, R., Horst, P., OBrien, D., Olivier, S., Wheeler, K., and Viviano, B. (2011). Scheduling OpenMP for Qthreads with MAESTRO. Technical Report TR-11-2, Renaissance Computing Institute, Chapel Hill, NC.

Prins, J., Huan, J., Pugh, W., Tseng, C.-W., and Sadayappan, P. (2003). UPC implementation of an unbalanced tree search benchmark. Technical Report TR03-034, University of North Carolina at Chapel Hill.

Reinders, J. (2007). *Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, Sebastopol, CA.

Sanchez, D., Yoo, R. M., and Kozyrakis, C. (2010). Flexible architectural support for fine-grain scheduling. In *ASPLOS '10: Proc. 15th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 311–322. ACM.

Smith, B. J. (1981). Architecture and applications of the HEP multiprocessor computer system. In *4th Symposium on Real-Time Signal Processing*, pages 241–248. SPIE.

Snape, J., van den Berg, J., Guy, S. J., and Manocha, D. (2011). The Hybrid Reciprocal Velocity Obstacle. *IEEE Transactions on Robotics*, 27(4):696 –706.

Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., and Petersen, P. (2002). Compiler support of the workqueuing execution model for Intel SMP architectures. In *EWOMP '02: Proc. 4th European Workshop on OpenMP*.

Sundell, H. and Tsigas, P. (2005). Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In Higashino, T., editor, *OPODIS 2004: 8th Intl. Conference on Principles of Distributed Systems*, volume 3544 of *LNCS*, pages 240–255. Springer.

Tallent, N. R. and Mellor-Crummey, J. M. (2009). Effective performance measurement and analysis of multithreaded applications. In *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240. ACM.

Tallent, N. R., Mellor-Crummey, J. M., and Porterfield, A. (2010). Analyzing lock contention in multithreaded applications. In *PPoPP '10: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 269–280. ACM.

Thibault, S., Namyst, R., and Wacrenier, P.-A. (2007). Building portable thread schedulers for hierarchical multiprocessors: The BubbleSched framework. In *EuroPar 2007: Proc. 13th Intl. Euro-Par Conference on Parallel Processing*, Rennes,France. ACM.

Thottethodi, M., Chatterjee, S., and Lebeck, A. R. (1998). Tuning Strassen's matrix multiplication for memory efficiency. In *SC98: Proc. 1998 ACM/IEEE Conference on Supercomputing*, pages 1–14. IEEE.

van Nieuwpoort, R., Kielmann, T., and Bal, H. E. (2000). Satin: Efficient parallel divide-and-conquer in Java. In *Euro-Par '00: Proc. 6th Intl. Euro-Par Conference on Parallel Processing*, pages 690–699, London, UK. Springer.

Wheeler, K. B., Murphy, R. C., and Thain, D. (2008). Qthreads: An API for programming with millions of lightweight threads. In *IPDPS 2008: Proc. 22nd IEEE Intl. Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE.

Wheeler, K. B. and Thain, D. (2010). Visualizing massively multithreaded applications with ThreadScope. *Concurrency and Computation: Practice and Experience*, 22:45–67.

Yan, Y., Zhao, J., Guo, Y., and Sarkar, V. (2010). Hierarchical place trees: A portable abstraction for task parallelism and data movement. In Gao, G. R., Pollock, L. L., Cavazos, J., and Li, X., editors, *LCPC 2009: 22nd Intl. Workshop on Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 172–187. Springer.