GOSSAMER: A LIGHTWEIGHT APPROACH TO USING
MULTICORE MACHINES

by

Joseph Anthony Roback

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2010

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Joseph Anthony Roback

ACKNOWLEDGEMENTS

I cannot adequately express my gratitude to my advisor, Dr. Greg Andrews for his inspiration, support, encouragement, trust, and patience throughout my doctoral work. I have been very fortunate to have an advisor who helped me overcome many difficult situations and finish this dissertation. I am also very grateful to him for his generosity, coordination, and supervision, which made it possible for me to complete this dissertation in a distance of more than 1,000 miles (even after his official retirement). I wish Greg a long and healthy retirement with his wife Mary and their family.

I would also like to thank the other members of my committee. Saumya Debray was always willing to put up with my ignorance of compiler theory and to take my random questions as I walked by his office's open door. David Lowenthal and John Hartman also provided helpful advice and feedback.

I would also like to thank the National Science Foundation. This research was supported in part by NSF Grants CNS-0410918 and CNS-0615347.

A special thanks to my fellow graduate students from the department of computer science: Kevin Coogan, Somu Perianayagam, Igor Crk, Kyri Pavlou, and Varun Khare. They always were around for lunch or coffee when procrastination ranked higher than dissertation work.

Most importantly, I'd like to thank my family, who have been very supportive throughout my life and academic career. I would especially like to thank my mother for her endless support and for encouraging me to pursue my passion in computer science. I would like to thank my grandfather for his endless teachings about life, and my grandmother, Jean-Jean, who was always there to mend my torn clothes. Last but certainly not least, a big thanks to Jess, my patient and loving soul mate whom I love dearly.

DEDICATION

To Mom, Jean-Jean, and Gramp, for all their support and generosity.

TABLE OF CONTENTS

LIST OF FIGURES	7
LIST OF TABLES	8
ABSTRACT	9
CHAPTER 1 INTRODUCTION	10
1.1 Current Approaches	11
1.2 The Gossamer Approach and Contributions	12
1.3 Outline	13
CHAPTER 2 ANNOTATIONS	14
2.1 Task and Recursive Parallelism	14
2.2 Loop Parallelism	15
2.3 Domain Decomposition	16
2.4 Synchronization	20
2.5 MapReduce	21
CHAPTER 3 EXAMPLES	23
3.1 Quicksort	23
3.2 N-Queens	23
3.3 Bzip2	25
3.4 Matrix Multiplication	26
3.5 Sparse Matrix Multiplication	27
3.6 Jacobi Iteration	30
3.7 Run Length Encoding	31
3.8 Word Count	32
3.9 Multigrid	34
3.10 Gravitational N-Body Problem	34
3.11 Summary	37
CHAPTER 4 TRANSLATOR AND RUN-TIME SYSTEM	40
4.1 Recursive and Task Parallelism	42
4.2 Loop Parallelism	44
4.3 Domain Decomposition	47
4.4 Additional Synchronization	50

TABLE OF CONTENTS – *Continued*

4.5	MapReduce	54
4.6	Static Program Analysis and Feedback	55
4.7	Run-Time Program Analysis and Performance Profiling	57
4.8	Combining Multiple Levels of Parallelism	58
CHAPTER 5 PERFORMANCE		60
5.1	Testing Methodology and Hardware	60
5.2	Application Performance on Intel Xeon E5405 (8-core)	61
5.3	Application Performance on SGI Altix 4700 (32-core)	65
5.4	Application-Independent Gossamer Overheads	68
5.5	Summary	69
CHAPTER 6 RELATED WORK		70
6.1	Thread Packages and Libraries	70
6.2	Parallelizing Compilers	71
6.3	Parallel Languages	72
6.4	Stream and Vector Processing Packages	75
6.5	Annotation Based Languages	78
6.6	Performance of Gossamer versus Cilk++ and OpenMP	79
CHAPTER 7 CONCLUSION AND FUTURE WORK		82
APPENDIX A GOSSAMER ANNOTATIONS		85
A.1	Task Parallelism	85
A.2	Loop Parallelism	85
A.3	Domain Decomposition	86
A.4	Synchronization	86
A.5	MapReduce	87
APPENDIX B GOSSAMER MANUAL		88
B.1	Installing Gossamer	88
B.2	Compiling and Running Gossamer Programs	88
B.3	Debugging and Profiling Gossamer Programs	91
REFERENCES		94

LIST OF FIGURES

3.1	Example: Quicksort	24
3.2	Example: N-Queens Problem	24
3.3	Example: Bzip2	26
3.4	Example: Matrix Multiplication	27
3.5	Example: Sparse Matrix Multiplication	28
3.6	Example: Jacobi Iteration	29
3.7	Example: Run Length Encoding	31
3.8	Example: Word Count	33
3.9	Example: Multigrid	35
3.10	Example: N-Body	36
4.1	Gossamer Overview	41
4.2	Two Dimensional divide	48
4.3	MapReduce Hash Table	53
6.1	Matrix Multiplication Using Only One Filament Per Server Thread . .	81

LIST OF TABLES

2.1	Gossamer Annotations	14
3.1	Applications and the Annotations They Use	38
3.2	Dwarf Computational Patterns and Applications	39
5.1	Execution Times on Intel Xeon	62
5.2	Application Speedups on Intel Xeon	62
5.3	Execution Times on SGI Altix 4700	66
5.4	Application Speedups on SGI Altix 4700	66
5.5	Run-Time Initialization Overheads (CPU Cycles)	69
5.6	Filament Creation Overheads (Intel Xeon workstation)	69
6.1	Gossamer, Cilk++, and OpenMP: Execution Times	79
6.2	Gossamer, Cilk++, and OpenMP: Speedups	80

ABSTRACT

The key to performance improvements in the multicore era is for software to utilize the newly available concurrency. Consequently, programmers will have to learn new programming techniques, and software systems will have to be able to manage the parallelism effectively. The challenge is to do so simply, portably, and efficiently.

This dissertation presents a lightweight programming framework called Gossamer that is easy to use, enables the solution of a broad range of parallel programming problems, and produces efficient code. Gossamer supports task and recursive parallelism, iterative parallelism, domain decomposition, pipelined computations, and MapReduce computations. Gossamer contains (1) a set of high-level annotations that one adds to a sequential program to specify concurrency and synchronization, (2) a source-to-source translator that produces an optimized program, and (3) a run-time system that provides efficient threads and synchronization. The annotation-based programming model simplifies writing parallel programs by allowing the programmer to concentrate on the application and not the extensive book-keeping involved with concurrency and synchronization; moreover, the annotations never reference any particulars of the underlying hardware.

CHAPTER 1

INTRODUCTION

In recent years, a corollary of Moore’s law [47] that predicts processors will double in performance every couple of years has ended. Microprocessor manufacturers have encountered a clock frequency wall as energy consumption has limited the clock frequency scaling that was the norm for three decades. In 2004, Intel initially delayed, then cancelled its flagship 4 GHz chip. However, Dr. Gordon Moore’s original observation of doubling transistor count every 18 to 24 months is still holding true, and chip manufacturers are turning the excess transistors into multiple cores per chip. Intel’s commodity microprocessors currently have up to 8 cores, and within the next two years 12-core processors are expected. In laboratory tests, Intel has already demonstrated 80 cores on a single silicon chip [31].

With the emergence of multicore computer chips, improvements in application performance will depend not on faster clock speeds, but on an application’s ability to utilize the additional cores. Consequently, programmers will have to learn new programming techniques, and software systems will have to be able to manage parallelism effectively.

Recent papers [4, 5] summarize the challenges resulting from the multicore era and describes twelve so-called dwarfs—types of computing and communication patterns that occur in parallel programs. One of the key points in these papers is that a general programming model must accommodate all of the patterns defined by the dwarfs, singly or in combination. The challenge is to do so both simply and efficiently.

1.1 Current Approaches

The most common way to write a parallel program is to use a sequential programming language with a threads library such as POSIX threads (Pthreads) [34, 49], Intel Threading Building Blocks (Intel TBB) [35], or Apple’s Grand Central Dispatch (GCD) [3]. This approach is general and efficient, but it is not simple because the programmer has to control every aspect of concurrency and synchronization using low-level APIs. At the other extreme, one could use a compiler that automatically parallelizes code or uses optimistic concurrency techniques [8, 12, 10]. This is simple and it works well for some kinds of parallelism (e.g., independent loop iterations), but it is not general. In between are languages such as C# and Java that contain explicit mechanisms for forking threads and synchronizing their execution. This approach is general, and it is easier to use than libraries, but the programmer still has to manage concurrency and synchronization, and these languages often impose significant performance overheads. A fourth approach is to use a parallel language such as Erlang [38], Fortress [56], X10 [54], or ZPL [17] that provide implicit parallelism through high-level abstractions. This makes parallel programs easier to write, and these languages have efficient implementations; however, they require that one use a new programming model, which complicates parallelizing existing applications.

The simplest way to write a parallel program is to start with a sequential program and annotate it with directives that specify concurrency and synchronization. This approach is supported by Cilk [26], Cilk++ [18], OpenMP [51], and Unified Parallel C (UPC) [57]. Cilk++ is the simplest with only three keywords that extend C++. OpenMP is the industry standard and supports C, C++, and Fortran. UPC extends C using the single program multiple data (SPMD) model of computation. (More specialized annotation based approaches include Nvidia CUDA [50], ATI Stream SDK [2], Intel Ct [27], OpenMP to GPGPU [39], and RapidMind [53]; these target stream computations and execution on general purpose graphical processing units (GPGPUs).) All these approaches have efficient implementations, but none supports

all the computational patterns (dwarfs) defined by Asanovic et al. [4, 5].

1.2 The Gossamer Approach and Contributions

This dissertation describes Gossamer, a threads package and concurrent programming model for multicore shared-memory machines. The design of Gossamer has been guided by three goals: generality, ease of use, and efficiency. Being general means being able to handle a variety of parallel computations, like the dwarf patterns defined by [4, 5]. Being easy to use means having only a few basic concepts that are easy to add to sequential programs. Being efficient means having low overhead from the added concurrency and synchronization and having performance that scales with the number of available cores.

Gossamer has three major components:

1. A set of fifteen high-level annotations that one adds to a sequential C program. Ten annotations are used to specify concurrency and synchronization; five are used to program MapReduce [22, 61] computations.
2. A source-to-source translator that takes an annotated sequential program and produces an optimized program that uses the Gossamer run-time threading library. The translator also provides feedback about suspect annotations (such as loop-carried dependencies in parallel loops and unprotected shared variables).
3. A run-time system that provides efficient fine-grained threads and high-level synchronization constructs—e.g., barriers, atomic actions, reductions, a thread ordering primitive, I/O buffering, and associative memory.

What sets Gossamer apart from earlier work is a more extensive set of annotations that enable solving a greater variety of applications. In addition to iterative, task, and recursive parallelism, Gossamer supports pipelined computations by a general ordering primitive, domain decomposition by means of replicated code patterns, and MapReduce computations by means of an efficient associative memory type.

Gossamer supports portability by using the POSIX threads library to implement its run-time system; moreover, annotations in Gossamer make no reference to the underlying hardware platform. Gossamer currently runs on a variety of machines; platforms including Intel x86, Intel Itanium, and Motorola PowerPC architectures running Linux or Mac OS X. The run-time library contains less than one hundred lines of machine-dependent assembly code, which makes it easy to port to new hardware architectures.

1.3 Outline

The rest of the dissertation is organized as follows. Chapter 2 introduces our programming model and annotations. Chapter 3 illustrates Gossamer's annotations by means of numerous examples. Chapter 4 describes the Gossamer translator and run-time system. Chapter 5 presents performance results. Chapter 6 discusses related work. Chapter 7 summarizes the dissertation and discusses future work. Appendix A gives a brief synopsis of the Gossamer annotations. Appendix B is the Gossamer user manual, which describes how to install and use Gossamer.

CHAPTER 2

ANNOTATIONS

Gossamer provides 15 simple annotations, as listed in Table 2.1: ten to specify concurrency and synchronization and five to program MapReduce computations. The concurrency and synchronization annotations have the property that if they are deleted from a program, what remains is the underlying sequential computation. The MapReduce annotations greatly simplify the bookkeeping required by MapReduce computations. The choice of annotations was driven by having the smallest set that was sufficient to program all the concurrency and synchronization patterns defined in [4, 5].

2.1 Task and Recursive Parallelism

The `fork` annotation supports task and recursive parallelism. Execution of

```
<variable> = fork function(arguments);
```

creates a new thread that is executed concurrently with the calling thread (the parent) and optionally returns a result when the thread terminates. All children created from the same parent are siblings. The parent can wait for one or more forked functions to terminate by executing

```
join(n);
```

Use	Annotations
Concurrency	<code>fork</code> , <code>parallel</code> , <code>divide/replicate</code>
Synchronization	<code>atomic</code> , <code>barrier</code> , <code>buffered</code> , <code>copy</code> , <code>join</code> , <code>ordered</code> , <code>shared</code>
MapReduce	<code>mr_space</code> , <code>mr_list</code> , <code>mr_put</code> , <code>mr_getkey</code> , <code>mr_getvalue</code>

Table 2.1: Gossamer Annotations

where the optional argument specifies the number of children for whom to wait (the default is all). In C, arrays are passed by reference, and this could lead to interference if two children update the same array. The `copy` annotation, when placed in front of an array argument to a forked function, gives the function its own copy of the array; dimensions are given to specify the number of elements to copy. The basic form of a `copy` annotation is

```
<variable> = fork function(copy arg[n][...]);
```

2.2 Loop Parallelism

The `parallel` annotation supports data parallelism that occurs when all iterations of a for loop are independent and hence can be executed concurrently. The `parallel` annotation can also be used with nested loops. The general form of a `parallel` loop is

```
parallel for (init-expr; test-expr; incr-expr) {
    statements
}
```

where each loop iteration is executed as separate thread. The parent thread waits for these threads to complete before executing the statement following the `parallel` for loop. The *init-expr* must declare and initialize a single variable of either integer or pointer type. If the variable was initialized outside the loop construct, no explicit *init-expr* is needed. Each thread receives a private copy of the *init-expr* variable and it is initialized to its current value within the loop expression. The *test-expr* must compare the *init-expr* variable to a termination value using one of the following comparison operators: `<`, `<=`, `!=`, `>=`, or `>`. The *incr-expr* must perform an increment or decrement assignment to the *init-expr* variable. The following mathematical operations are supported within the *incr-expr*: `+=`, `-=`, `*=`, `/=`, `++`, and `--`. Lastly, any variables appearing in the loop expression must not be modified within the loop body itself.

Local variables used within the loop body are made thread private by default.

They are initialized before entering the parallel loop and are undefined upon exiting the parallel loop. Local variables can be explicitly shared between threads by adding the `shared` annotation to a variable's declaration. For example, in

```
void myfunc(int n) {
    shared struct s;
    ... // s is initialized here
    parallel for (i = 0; i < n; i++) {
        statements
    }
}
```

the local variable `s` is shared among the iteration threads and any updates to variable may need to be synchronized. Global variables are shared by default and must be copied into a local variable if a thread-private copy is required.

2.3 Domain Decomposition

The third concurrency annotation, `divide/replicate`, supports data parallelism that results from domain decomposition. In particular, domain decomposition is used when shared arrays can be divided into independent regions, and the same code is to be executed on each region. The general form of `divide/replicate` is

```
divide array1[size]<[size]> <, array2[size]<[size]>, ...>
replicate {
    statements
}
```

The `divide` annotation defines how arrays, one or two dimensional, are to be divided among threads. Multiple arrays are separated by commas. Array dimensions are required in order for the run-time to know how big each array is, and they can be any integer variable or constant. The `replicate` annotation defines the block of code which all threads will execute over the divided arrays. The parent thread waits for all replicated threads to complete before executing the statement following the `replicate` block.

As an example,

```
divide data[size] replicate {
    statements
}
```

divides array `data` into `size/NCPU` equal-size regions, where `NCPU`¹ is the number of processors, and creates a thread on each processor that executes the replicated statements.

Dividing in two dimensions works in a similar way. For example, consider the following code:

```
divide grid[n][m] replicate {
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            do_something(grid[i][j]);
        }
    }
}
```

Here, the `grid` is divided up into rows and columns as follows:

$$\text{rows} = n / \text{floor}(\text{sqrt}(\text{NCPU}))$$

$$\text{columns} = m / (\text{NCPU} / \text{floor}(\text{sqrt}(\text{NCPU})))$$

By default, a two dimensional array is divided among both dimensions. An optional caret (^) can be used inside the array dimensions to control precisely which dimensions are divided. For example,

```
divide grid[n^][m^] replicate {
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            do_something(grid[i][j]);
        }
    }
}
```

¹`NCPU` is a predefined macro in Gossamer and is always equal to the number of processors available to the run-time.

```
}
```

divides `grid` explicitly in both dimensions.

To divide only along the first dimension to produce horizontal strips, the caret is specified only in the first dimension. For example,

```
divide grid[n^][m] replicate {
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      do_something(grid[i][j]);
    }
  }
}
```

divides the first dimension among the total number of available processors—`NCPU`. This results in `NCPU` threads each having a subarray consisting of `n/NCPU` rows and `m` columns.

To divide only along the second dimension and produce vertical strips, the caret is specified only in the second dimension. For example,

```
divide grid[n][m^] replicate {
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      do_something(grid[i][j]);
    }
  }
}
```

divides the second dimension among the total number of available processors. This results in `NCPU` threads each having a subarray consisting of `n` rows and `m/NCPU` columns.

An optional `where` clause can be added to `divide/replicate` for one dimensional divides to specify a boolean condition that must be true at divide points; the effect is to adjust the array boundaries to the right, as necessary, until the boolean condition is satisfied. As an example,

```

divide data[n]
    where data[divide_left] != data[divide_right]
replicate {
    statements
}

```

uses a **where** clause to specify that the values of partition boundaries cannot be equal to one another. Pre-defined symbols **divide_left** and **divide_right** refer to the indices to the left and right of a divide point. (See Chapter 4 for details on how this is implemented.) As a second example,

```

divide data[n]
    where isspace(data[divide_right])
replicate {
    statements
}

```

specifies that the element after each split must be whitespace; this might be used, for example, if the data contains words.

Barrier synchronization among the replicated threads is specified by executing **barrier** within a replicated code block. For example,

```

divide data[size] replicate {
    // step 1
    statements
    barrier;
    // step 2
    statements
}

```

says that all threads created by **divide/replicate** will complete the first step of the **replicate** block before moving on to the second step.

All local variables used within the **divide/replicate** annotation are made thread private by default. Thread-private variables are initialized before entering a replicated region and are undefined upon exiting the replicated region. Local variables

can be explicitly shared between threads by adding the **shared** annotation to a variable's declaration. Global variables are shared by default and must be copied into a local variable if a thread-private copy is required.

2.4 Synchronization

Gossamer contains seven annotations for synchronization. Four were introduced in the previous section: (1) the **join** annotation used explicitly with **fork** and implicitly with **parallel**, (2) the **copy** annotation used to make copies of arguments of forked functions, (3) the **barrier** annotation used to provide barrier synchronization for **divide/replicate** threads, and (4) the **share** annotation to explicitly share a local variable among threads. The three remaining synchronization annotations are discussed below.

The **ordered** annotation is used to serialize execution of code among all sibling threads created by the same parent. The general form is

```
ordered {
    statements
}
```

In particular, **ordered** delays a thread until its predecessor sibling has finished executing its instance of **statements**.² Ordering is often needed to serialize access to I/O streams. The **ordered** annotation can be used in any concurrent code that requires it (unlike in OpenMP, which only permits ordered within parallel for loops.)

The **buffered** annotation causes the output of the write calls used within its code block to be written to a local buffer that is flushed on thread termination. The general form is

```
buffered (<ordered> <, buffer size>) {
    write statements
}
```

²The predecessor sibling of a filament is the filament created before the current one by the same parent.

The optional `ordered` says that the buffers are flushed in thread-sibling order, similar to the `ordered` annotation described above. The optional `buffer size` specifies the size of internal buffers to use; the default is the run-time system's memory page size.

The `atomic` annotation allows a set of statements to be executed atomically. The form of an `atomic` statement is

```
atomic <using mutex | using spinlock> {
    statements
}
```

By default, an `atomic` block is protected using a mutex lock. It can be explicitly specified by appending `using mutex` to the `atomic` annotation. Spinlocks are used if `using spinlock` is appended to an `atomic` annotation. If the only statement(s) inside an `atomic` block are assignments, and each assignment is mathematically associative and commutative, then the assignment is performed as a reduction (see Chapter 4 for details).

2.5 MapReduce

Gossamer supports the MapReduce programming model by providing two built-in associative memory types and three operations on those types. All the MapReduce annotations interact directly with the associative memory implementation, whereas all concurrency and synchronization in MapReduce computations come from the annotations introduced above.

The `mr_space` type defines an associate memory space for (key,value) pairs. MapReduce spaces are declared using the following form

```
mr_space variable-name (key-type, value-type);
```

Key and value types can be any C primitive type. For example, a MapReduce space, called `myspace`, with an string key type and integer value would be declared as:

```
mr_space myspace (char *, int);
```

The `mr_list` annotation is an opaque type that holds a list of values for a particular key from a space. It is declared using the standard C declaration syntax for

a variable; it has the following form

```
mr_list variable-name;
```

The `mr_put()` operation puts a (key, value) pair into an `mr_space`. It has the general form

```
void mr_put (mr_space space ,  
             key-type key, value-type value);
```

where the `key-type` and `value-type` must match the types defined for `space`.

The `mr_getkey()` operation removes the next available key from an `mr_space` and returns the key and its associated list of values. It has the general form

```
bool mr_getkey(mr_space space ,  
               key-type &key, mr_list &values);
```

where the `key-type` must match the key type defined for `space`. If there are keys remaining in `space`, `mr_getkey()` extracts the key and an `mr_list` containing all its associated values and returns true; otherwise it returns false.

The `mr_getvalue()` operation removes the next value from an `mr_list` and returns it. It has the general form

```
bool mr_getvalue(mr_space space ,  
                 mr_list values, value-type &value);
```

where the `value-type` must match the value type defined for `space`. If there are items remaining in `values` list, `mr_getvalue()` extracts the value into `&value` and returns true; otherwise it returns false.

CHAPTER 3

EXAMPLES

This section presents examples of classic parallel programs, as well as sequential programs annotated for parallelism. The Gossamer annotations add only a few extra lines of code to each program. In the examples, Gossamer annotations are highlighted in boldface.

3.1 Quicksort

Quicksort is a classic divide and conquer algorithm. It sorts by dividing a list into two sub-lists and then recursively sorting each sub-list. Since the sub-lists are independent, they can be sorted in parallel.

An annotated version of the quicksort algorithm is shown in Figure 3.1. The Gossamer program sequentially partitions the array passed to `qsort()`; the last element is chosen as the pivot and then the array is reordered such that all elements with values less than the pivot come before the pivot and all elements with values greater than the pivot come after it. When the partition step is finished, quicksort is forked for each subarray. A `join` is used to ensure each of the two subarrays have finished sorting before returning from `qsort`.

3.2 N-Queens

N-Queens is a depth-first backtracking algorithm [21]. It tries to solve the problem of placing `N` chess queens on an `NxN` chessboard such that no one queen can capture any other using the standard chess queen's moves. The problem can be solved by placing a queen on a row and column and then recursively testing the rest of the board starting on the next row.

```

void qsort(int *begin, int *end)
{
    if (begin != end) {
        int *middle;
        end--;
        middle = partition(begin, end, *end);
        myswap(end, middle);
        fork qsort(begin, middle);
        fork qsort(++middle, ++end);
        join;
    }
}

```

Figure 3.1: Example: Quicksort

```

int solutions = 0;
void putqueen(char **board, int row)
{
    int j;
    if (row == n) {
        atomic { solutions++; }
        return;
    }
    for (j = 0; j < n; j++) {
        if (isOk(board, row, j)) {
            board[row][j] = 'Q';
            fork putqueen(copy board[n][n], row+1);
            board[row][j] = '-';
        }
    }
    join;
}

```

Figure 3.2: Example: N-Queens Problem

An annotated version of the N-Queens algorithm is shown in Figure 3.2. In this example, each attempt at placing a queen on the board checked in parallel using the recursive `putqueen()` function. If all the rows have been checked, the base case has been reached and the solutions count is incremented; otherwise, each column in the current row is checked for horizontal, vertical, and diagonal conflicts by the `isOk` function. If there are no conflicts, a queen is placed on the board at the current position and `putqueen()` is forked and the next row is checked in parallel. The queen is then removed from the current position and the next column is checked. A `join` is used to ensure that all parallel recursive checks have completed before returning.

Two issues arise from parallelizing `putqueen()`. First, the global variable `solutions` is incremented every time a solution is found. The `atomic` annotation is used to ensure that updates are executed one at a time. Second, the two-dimensional array, `board`, is passed by reference to each call to `putqueen()`, which would cause the board contents to be shared among the `putqueen()` threads. The `copy` annotation is used to ensure that each `putqueen()` thread receives its own copy of the `board`, not just a copy of the pointer to the board.

3.3 Bzip2

Bzip2 [55] compression uses the Burrows-Wheeler block-sorting text transform algorithm, followed by a move-to-front transform, then a run length encoding transform, and finally Huffman coding. Compression is performed on independent blocks of data, which lends itself naturally to block-level task parallelism.

A parallel version of bzip2 using Gossamer annotations is shown in Figure 3.3. The `main()` function reads blocks of data from a file and forks the function `compressBlock()` to compress each block in parallel. The `copy` annotation is used to give each `compressBlock()` thread a copy of the data block to be compressed. Once all the input data blocks have been read and distributed among `compressBlock()` threads, the parent thread uses `join` to wait for all its children to finish before terminating the program. Moreover, Bzip2 decompression assumes that compressed

```

int main(int argc, char **argv) {
    ...
    while (!feof(infile)) {
        insize = fread(inblock, 1, BLKSIZE, infile);
        fork compressBlock(copy inblock[insize]);
    }
    join;
    ...
}

void compressBlock(char *inblock, int insize) {
    ...
    BZ2_bzBuffToBuffCompress(inblock, insize,
                             outblock, &outsize);
    ordered {
        fwrite(outblock, 1, outsize, outfile);
    }
    ...
}

```

Figure 3.3: Example: Bzip2

blocks are output in the same order that uncompressed blocks are read from input, thus `fwrite()` calls in `compressBlock()` must be enclosed in an `ordered` annotation to ensure that the previously forked thread has written its compressed block before the current thread writes its block.

3.4 Matrix Multiplication

Matrix multiplication is an example of iterative data parallelism: each iteration of the two outer-most for loops can be executed in parallel. Both of the outer for loops in Figure 3.4 are thus prefixed with the `parallel` annotation. This results in n^2 tasks. Alternatively, one could prefix just the outer-most loop by `parallel`, which would result in n tasks and a slightly more efficient implementation. It would not, however, be efficient to prefix just the second loop by `parallel`, because threads would then be created and destroyed every time around the outer loop. (It would not be correct to try to parallelize the inner-most loop, because it has a loop-carried dependency—

```
double **A, **B, **C;
int i, j, k;
// initialize A,B arrays
parallel for (i = 0; i < n; i++) {
    parallel for (j = 0; j < n; ++j) {
        for (k = 0; k < n; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Figure 3.4: Example: Matrix Multiplication

the assignment to `C[i][j]` is dependent on the previous loop iteration. In this case the Gossamer translator issues a warning. Further details on the translator and the feedback it provides are presented in Chapter 4.)

3.5 Sparse Matrix Multiplication

Sparse matrix multiplication is another example of iterative data parallelism: each iteration of the two outer-most for loops can be executed in parallel. The number of tasks created is directly proportional to how dense the `A` and `B` matrices are, in contrast to the previous example where the number of tasks is based on the matrices' dimensions. The sparse data layout can often lead to unbalanced workloads if individual tasks are too coarse, so parallelizing both outer-most for loops will ensure good distribution of the workload by using the finest grain tasks possible.

Figure 3.5 shows a parallel sparse matrix multiplication implementation using row and column linked list representation for the matrices [24]. The outer loops are executed in parallel and create `A->num_rows * B->num_cols` threads. The inner while loop avoids scanning each list on every iteration by moving down both the row and column lists in tandem, searching for elements in the row list that have the same index as elements in the column list. Since the lists are kept in order by index, this can be done in one scan through the lists. Each iteration through the loop moves forward at least one position in one of the lists, so the loop terminates

```

typedef struct _sa_cell_t {
    unsigned int index;
    double value;
    struct _sa_cell_t *next;
} sa_cell_t;

typedef struct {
    unsigned int num_rows;
    sa_cell_t **rows;
    unsigned int num_cols;
    sa_cell_t **cols;
} sa_t;

sa_t *A, *B, *C;
int i, j;
// initialize A,B sparse arrays
parallel for (i = 0; i < A->num_rows; i++) {
    for (j = 0; j < B->num_cols; j++) {
        sparse_array_cell_t *row_p = A->rows[i];
        sparse_array_cell_t *col_p = B->cols[j];
        double sum = 0.0f;

        while ((row_p != NULL) && (col_p != NULL)) {
            if (row_p->index == col_p->index) {
                // A[i][k] * B[k][j]
                sum += row_p->value * col_p->value;
                row_p = row_p->next;
                col_p = col_p->next;
            }
            else if (row_p->index < col_p->index) {
                row_p = row_p->next;
            }
            else {
                col_p = col_p->next;
            }
        }
        if (sum != 0.0f) {
            atomic { sparseArraySetCell(C, i, j, sum); }
        }
    }
}

```

Figure 3.5: Example: Sparse Matrix Multiplication

```

double **old, **new;
int i, j, n, m, iters;
new++;
old++;
n -= 2;


old[n^][m], new[n^][m]
    replicate {
        for (iters = 0; iters < MAXITERS; iters += 2) {
            for (i = 0; i < n; i++) {
                for (j = 1; j < m; j++) {
                    // compute new values
                    new[i][j] = (old[i-1][j] + old[i+1][j] +
                                old[i][j-1] + old[i][j+1]) * 0.25;
                }
            }
            barrier;
            for (i = 0; i < n; i++) {
                for (j = 1; j < m; j++) {
                    // compute old values
                    old[i][j] = (new[i-1][j] + new[i+1][j] +
                                new[i][j-1] + new[i][j+1]) * 0.25;
                }
            }
            barrier;
        }
    }


```

Figure 3.6: Example: Jacobi Iteration

after at most $R + C$ iterations (where R is the number of elements in the row list and C the number of elements in the column list). If the row and column have the same index, their values are multiplied and stored in the local variable `sum`. After the inner while loop terminates, if the resulting sum is non-zero, it is stored in the sparse matrix `C` by using an $O(n)$ operation called `sparseArraySetCell()`; The set cell operation inserts a row and column cell for the corresponding i and j indices. Moreover, the set operation must be synchronized using the `atomic` annotation to ensure proper insertion of the new cell within the row and column linked lists.

3.6 Jacobi Iteration

Laplace's equation in two dimensions is the partial differential equation $\nabla^2(\Phi) = 0$. Given boundary values for a region, its solution is the steady values of interior points. These values can be approximated using a finite difference method such as Jacobi iteration. In particular, discretize the region using a grid of equally spaced points, and initialize each point to some value. Then repeatedly compute a new value for each grid point, where the new value is the average of the values of a point's four neighbors from the previous iteration. The computation terminates when enough of iterations have occurred to produce convergence.

Because Jacobi iteration uses two grids, all new values can be computed in parallel. Figure 3.6 shows an annotated program. In this case, we use the `divide/replicate` annotation because we want to execute the original algorithm on sub-regions of the `old` and `new` grids. The computation first computes `new` values using the `old` grid points. A `barrier` annotation is used to ensure that all threads have completed this step before continuing. Then the computation computes `old` values using the `new` grid points. Again a `barrier` annotation is used to ensure that all threads have completed this step before continuing. The update process is repeated for `MAXITERS` iterations.

In Figure 3.6, both grids are divided into strips on the first dimension. To divide the grids vertically, the `divide/replicate` annotation would be specified as follows:

```
...


divide

 old[n][m^] new[n][m^] replicate {
...

```

To divide the grids into blocks, the `divide/replicate` annotation would be specified as

```
...


divide

 old[n^][m^], new[n^][m^] replicate {
...

```

or alternatively, the carets could be emitted altogether to get the default division

```

FILE *out_fp;
char *data;
int size, run, val;



data[size]
  where data[divide_left] != data[divide_right]
  replicate {
    while (size > 0) {
      val = *data++;
      size--;
      run = 1;
      // find run length of current value
      while (val == *data && size > 0) {
        run++; data++; size--;
        if (run == RUNMAX) { break; }
      }
      buffered (ordered) {
        fwrite(&val, sizeof(int), 1, out_fp);
        fwrite(&run, sizeof(int), 1, out_fp);
      }
    }
  }


```

Figure 3.7: Example: Run Length Encoding

by blocks.

3.7 Run Length Encoding

Run length encoding (RLE) is a form of data compression in which sequences of the same data value are stored as pairs. Figure 3.7 shows an RLE implementation that scans an array byte-by-byte recording each run and writing the (value,count) pair to output. The **divide/replicate** annotation is used to divide **data** into chunks—each thread receives a private copy of **data** and **size**; the private copies correspond to each thread’s **data** starting address and smaller chunk **size**. The original sequential code inside the **replicate** block is executed concurrently on each chunk. Note that the replicated code is a while loop over the input, whereas typically data parallelism is programmed using a for loop.

The **where** annotation is used in RLE to ensure that the data is not split at points that would break up runs of identical values. (Without the **where** clause, the output would be a legal encoding, but it might not be the same as the output of the original sequential program.) The **buffered** annotation is used to buffer all (value,count) **fwrite()** calls in thread local storage, which greatly improves efficiency. RLE's output requirements are similar to Bzip2 shown in Figure 3.3. Output is written out in the same order as it was read in by using the **ordered** option on the **buffered** annotation. In particular, the **ordered** option ensures that thread buffers flushed at thread termination occur in the same order in which the threads were created with regard to their parent. The buffer's initial size can be specified as a option to the **buffered** annotation if the size is known a priori as a performance optimization. For example, if the variable **bufsize** contained the initial buffer size to be used by RLE, the **buffered** annotation would be as follows:

```
buffered (ordered, bufsize) {
    fwrite(&val, sizeof(int), 1, out_fp);
    fwrite(&run, sizeof(int), 1, out_fp);
}
```

3.8 Word Count

Word count calculates the number of occurrences of each word in a set of data files. This problem can be solved using the MapReduce programming style. The map phases finds all the words in the files; the reduce phase counts the number of instances of each word. Figure 3.8 gives a Gossamer implementation that uses the **fork** and **join** annotations for concurrency and the MapReduce annotations to implement intermediate storage.

The map phase forks a thread for each input file. Each thread reads the data in a file and puts words in the **mr_space wordcount** using the **mr_put** annotation. The reduce phase forks a thread for each word in **wordcount** it retrieves using the **mr_getkey** annotation. Each reduce thread extracts all of a word's values from **wordcount** using the **mr_getvalues** annotation, sums them, and writes the final count

```
mr_space wordcount(char *, int);

main(void) {
    char *key;
    mr_list values;

    for (i = 0; i < n; i++)
        fork map(file[i]);
    join;

    while (mr_getkey(wordcount, &key, &values))
        fork reduce(key, values);
    join;
}

void map(char *file) {
    char *word;
    while ((word = getnextword(file)) != NULL)
        mr_put(wordcount, word, 1);
}

void reduce(char *key, mr_list values) {
    int val, count = 0;
    while (mr_getvalue(wordcount, values, &val))
        count += val;
    printf("word: %s, count: %d\n", key, count);
}
```

Figure 3.8: Example: Word Count

to output. When the program terminates, `wordcount` is empty.

3.9 Multigrid

Multigrid, like Jacobi, is used to approximate solutions to partial differential equations. It is more complex than Jacobi, but it converges *much* faster by using multiple grid sizes. Multigrid starts with a fine-meshed grid and smooths the values using one the relaxation methods such as Jacobi or Gauss-Seidel. Then multigrid restricts the values on the fine grid to a coarser grid with twice the mesh size. This process of smoothing and restricting is continued until a desired coarsest grid size is reached. The relaxation method is then applied to the coarsest grid until it converges. Finally, a series of interpolation and smoothing steps are applied to move from coarser to finer grids until the original grid is reached. There are many variations to this basic method.

Figure 3.9 gives pseudocode for a multigrid algorithm that uses what is called a four-level V cycle. The main function, `multigrid`, divides the grid among the available processors and replicates code that implements the V cycle. This consists of a sequence of calls to the `relax`, `restrict`, and `interpolate` functions. Constant `V_SMOOTHING` specifies the number of iterations to perform in each relaxation step that smooths a fine grid. Variable `maxiters` specifies the number of times to iterate on the coarsest grid. Notice that each of `relax`, `restrict`, and `interpolate` ends with a `barrier` annotation; this ensures that the step is completed on all processors before any proceed to the next step. Similarly to Jacobi iteration shown earlier, the caret syntax can be used to control how the grid is divided among the processors.

3.10 Gravitational N-Body Problem

An N-Body simulation tracks the motion of a group of astronomical bodies that are subject to gravitational interaction. Each body in the simulation has a mass, position, and velocity. A simulation steps through time; in each step it calculates the forces on each body, and then updates their velocities and positions.

```

// declaration and initialization of grid[n][m];

void relax(int h, int iterations) {
    for (count = 0; count < iterations; count++) {
        perform one update cycle;
        barrier;
    }
}

void restrict(int fine, int coarse) {
    restrict to next smaller grid;
    barrier;
}

void interpolate(int coarse, int fine) {
    interpolate to next larger grid;
    barrier;
}

void multigrid() {
    grid++;
    n -= 2;
    divide grid[n][] replicate {
        relax(1, V_SMOOTHING);
        restrict(1, 2);
        relax(2, V_SMOOTHING);
        restrict(2, 4);
        relax(4, V_SMOOTHING);
        restrict(4, 8);
        relax(8, maxiters);
        interpolate(8, 4);
        relax(4, V_SMOOTHING);
        interpolate(4, 2);
        relax(2, V_SMOOTHING);
        interpolate(2, 1);
        relax(1, V_SMOOTHING);
    }
}

```

Figure 3.9: Example: Multigrid

```

point_t *f, *m, *p, *v;
node_t *tree;
int numBodies;

void nbody() {
    divide f[numBodies], m[numBodies],
    p[numBodies], v[numBodies] replicate {
        for (ntimes = 0; ntimes < ITERS; ntimes++) {
            if (THR_ID == 0 && time to build) {
                build new tree;
            }
            barrier;
            for (i = 0; i < numBodies; i++) {
                compute forces acting on body;
            }
            barrier;
            for (i = 0; i < numBodies; i++) {
                update position and velocity;
            }
            barrier;
        }
    }
}

```

Figure 3.10: Example: N-Body

The brute force method for computing forces requires $O(n^2)$ calculations. The Barnes-Hut method reduces this to $O(n \log n)$ by considering groups of remote bodies to be a single larger body [7]. In particular, Barnes-Hut constructs a quadtree by recursively dividing the space being simulated into smaller and smaller sets of four boxes until each box contains just one body. Barnes-Hut then makes an upward pass through the quadtree during which it calculates the total mass and center of mass for the bodies in each subtree and stores that information in the root of each subtree.

Figure 3.10 gives pseudocode for a Gossamer program that uses the Barnes-Hut algorithm. The Gossamer program divides the force, mass, position, and velocity vectors among the available processors and replicates code that builds the quadtree, calculates forces (using the quadtree), and moves bodies.

Each iteration of the simulation first checks whether it is time to build a new quadtree using only the first thread. A **barrier** is used to ensure the entire quadtree is built before any threads continue. Bodies move slowly, so the quadtree continues to be fairly accurate for a number of time steps. Moreover, the quadtree build in Figure 3.10 is a sequential bottleneck, so it should be done as infrequently as possible.

A **barrier** is inserted between the two inner for loops, because bodies cannot be moved until all force calculations have been completed. Likewise, a **barrier** is needed after the second for loop, since the quadtree cannot be rebuilt until all bodies have been moved.

3.11 Summary

The examples above collectively use the 15 Gossamer annotations. Table 3.1 lists the examples and indicates which annotations each uses.

Table 3.2 lists the computational patterns defined in [4, 5] and matches the example applications to their corresponding pattern. In addition to the examples shown in this chapter, Table 3.2 also lists Fast Fourier Transform, which has been

Application	Annotations Used
Quicksort	fork, join
N-Queens	fork, join, copy, atomic
Bzip2	fork, join, copy, ordered
Matrix Multiplication	parallel
Sparse Matrix Multiplication	parallel
Jacobi Iteration	divide/replicate, barrier
Run Length Encoding	divide/replicate where, buffered (ordered)
Word Count	fork, join, mr_space, mr_list, mr_put, mr_getkey, mr_getvalue
Multigrid	divide/replicate, barrier
N-Body	divide/replicate, barrier

Table 3.1: Applications and the Annotations They Use

programmed using Gossamer. The Berkeley paper classifies Bzip2’s Huffman encoding [36] as a finite state machine, but in Gossamer it is actually programmed using block-level task parallelism.

Circuits, dynamic programming, and unstructured grid dwarf patterns are not covered currently by the annotations and run-time. Cyclic redundancy checks (CRC) [52] are commonly used in programs that contain the circuits dwarf pattern. The generic CRC algorithm involves a loop that has a loop-carried dependency on the variable holding the checksum value. This requires the program’s algorithm (if possible) to be rewritten in order to achieve any parallelism.

Dynamic programming [23] patterns break problems up into smaller steps, store the results of the smaller problems in a table and use that table to find solutions to later problems. If the table is to be updated in parallel, it needs to be done atomically. This introduces circular dependencies and the possibility of deadlock, since Gossamer’s threads do not support preempt-able blocking synchronization. Dynamic programming algorithms could be more easily realized if Gossamer provided a heavier weight thread that supported preemption and is discussed in Chapter 7.

Unstructured grid computations perform updates on irregular meshes or grids and involve complex algorithms that are often hand-tuned for a given program and

Computational Pattern (dwarf)	Application
Finite State Machine	Run Length Encoding, Bzip2
Circuits	n/a
Graph Algorithms	Quicksort
Structured Grid	Jacobi Iteration, Multigrid
Dense Matrix	Matrix Multiplication
Sparse Matrix	Sparse Matrix Multiplication
Spectral	Fast Fourier Transform
Dynamic Programming	n/a
Particle Methods	N-Body
Backtrack/Branch+Bound	N-Queens
Graphical Models	Word Count
Unstructured Grid	n/a

Table 3.2: Dwarf Computational Patterns and Applications

machine architecture. This kind of computational pattern tends to impose difficult and elaborate solutions that are often unable to be solved by annotations alone. All three patterns not currently included in Gossamer are left for future work.

CHAPTER 4

TRANSLATOR AND RUN-TIME SYSTEM

The Gossamer translator converts an annotated C program into an executable program. It first parses the annotated program and performs analyses at the basic block level to determine which variables need to be moved into thread local storage. The translator then transforms the annotations into code that uses the Gossamer run-time library. Finally, the translator produces C source code, transparently pipes it into the GNU C Compiler, and links the result with the Gossamer run-time library. Figure 4.1 summarizes the process.

The Gossamer run-time system is implemented using the POSIX threads (Pthreads) library. In particular, Gossamer uses Pthreads to create one server thread per processor on the underlying hardware platform. Each server thread is initialized at program startup with an execution stack and a run queue, which contains application-level threads.

Application-level threads in Gossamer are called *filaments* [42]. Each filament is a lightweight structure represented by a function pointer, its arguments, and a pointer to where to store a return value, if any. Each filament descriptor also contains a pointer back to its parent and a count of the number of children it has forked; these are used to perform explicit join operations or implicit joins that result when exiting a `parallel` loop. Memory for the filament's descriptor comes from a pool on the processor that executes the fork; this is done to avoid overheads associated with allocating small chunks of memory using `malloc()`.

Filaments do not have private stacks; instead each filament shares the stack of the server thread on which it is executed. (Typical thread packages allocate a private stack for each new thread, which makes creation and context-switching slow while wasting memory due to relatively large default stack sizes.) The lack of a

Figure 4.1: Gossamer Overview

private stack means a filament cannot be preempted.¹ Once started, a filament executes until it terminates.² This avoids the overhead of requiring machine-dependent context-switching code. If a filament needs to lock a shared data structure, it waits until it acquires the lock. This cannot cause deadlock, because the lock must be held by another filament, which must already be executing and will not be preempted and hence will eventually release the lock.

Filaments are executed in FIFO order. New filaments are enqueued at the tail of a run queue; executing filaments are dequeued from the head of the run queue. A server thread with an empty run queue searches the run queues of other server threads looking for filaments to execute before dropping into an idle loop. When a filament terminates, it flushes any I/O buffers associated with the filament, decrements its parent's child count, and returns the filament descriptor to the appropriate memory pool.

Below we give more details on how concurrency annotations are translated, scheduled, and executed within the Gossamer framework; how synchronization anno-

¹Consequently, Gossamer cannot support applications such as multithreaded event handlers, file servers, and some traditional concurrent programming problems (e.g. readers-writers).

²Filaments performing a `join` operation will execute any children enqueued on server threads before busy waiting. See Sections 4.1 and 4.3 for details.

tations are translated and implemented; how MapReduce annotations are handled; the feedback provided by the translator; and the analysis and profiling provided by the translator and run-time.

4.1 Recursive and Task Parallelism

Each `fork` annotation creates a new filament to execute the function call. If the function returns a value, the memory address of the return variable is stored in the filament data structure; its value is undefined at run-time until the corresponding `join` operation is completed. An array argument prefixed with the `copy` annotation is copied into memory allocated at run-time before the filament is placed on a server thread's run queue; the allocated memory address is stored in the filament's descriptor instead of the original array reference for that argument.

The translator creates a wrapper function for each function that is forked in a program. The executing filament invokes the function's wrapper and not the actual forked function itself. The following code snippet illustrates the wrapper function generated for the `putqueen()` function in the n-queens example shown in Figure 3.2:

```
void __putqueen_wrapper(struct filament *f) {
    char **board = *((char**)((f->args)+0U));
    int row = *((int*)((f->args)+8U));
    int tmp;
    // unpacking arguments while invoking function call
    putqueen(board, row);
    // free the copied 'board' 2D array argument
    for (tmp = 0; tmp < n; tmp++) {
        free(board[tmp]);
    }
    free(board);
}
```

The wrapper function is responsible for (1) unpacking the arguments stored in the filament descriptor, (2) invoking the original function call using the unpacked ar-

guments, and (3) freeing any memory allocated for `copy` arguments. If the forked function returns a value, the wrapper is also responsible for copying the value into the memory address of the return variable stored in the filament descriptor.

Forked filaments are placed on server thread run queues using a round-robin algorithm to promote load balancing. This is a simple and efficient way to quickly spread work among all the server threads.

Gossamer translates a `join` annotation into a run-time library call. The join operation continues executing filaments until the current filament's child count is zero. If the server's run queue becomes empty and remaining children are currently executing on other server threads, then the join operation executes a busy wait until its child count becomes zero. This works correctly and is deadlock free because filament parent/child relationships are ordered; moreover, children are always created after their parent and terminate before their parent. This strict ordering, combined with the property that filaments cannot be preempted and once started execute to completion, guarantees a deadlock-free join operation.

To achieve good performance for recursive fork calls, the translator *prunes* repeated calls to `fork`. The pruning mechanism is dynamic, and it allows the run-time system to avoid overheads due to excessive filament creation and synchronization. A threshold is used to determine when enough work (filaments) have been created to keep all processors busy, and then forks are executed sequentially. When the number of active filaments drops below the threshold, forked calls dynamically switch back to creating filaments until enough work is generated. The threshold's default value is twice the number of server threads; it can also be set by the user at run-time.

The following code snippet shows how the recursive function used to calculate Fibonacci numbers is modified by the translator to allow for dynamic pruning:

```
int fib(int n) {
    if (global_runtime_prune_flag) {
        int x, y;
        if (n <= 2) { return 1; }
        x = fib(n - 1);
```

```

        y = fib(n - 2);
        return (x + y);
    } else {
        int x, y;
        if (n <= 2) { return 1; }
        x = fork fib(n - 1);
        y = fork fib(n - 2);
        join;
        return (x + y);
    }
}

```

Each call to `fib()` checks the pruning flag. If it is not set, the run-time executes the original annotated function body; otherwise it executes a sequential copy of the `fib()` function body. The sequential function body is a copy of the original function body with all annotations are removed from the code.³

4.2 Loop Parallelism

For a `parallel` annotation, one filament is created for each value of the loop iteration variable; that value is a filament's argument. Consider the following matrix multiplication example where the outermost loop is annotated with `parallel`:

```

int i, j, k;
parallel for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

```

³The exception is the `atomic` annotation, it is not removed in order to guarantee atomicity among all server threads since one or more server threads can be executing the sequential code branches concurrently.

Gossamer creates a function containing the `parallel` loop body, it replaces the original loop body with code to create a filament using the newly created function. The iteration variable `i` is the filament's single argument; the inner loops iteration variables, `j` and `k`, do not need to be copied into the filament since they are not live at the time of creation. The following code is generated by the translator for the above example:

```
int i;
// original outer loop now creates 'n' filaments
for (i = 0; i < n; i++) {
    create_filament(__gs_run_for0, i);
}
implicit_join(); // wait for all iterations to finish

// loop body function auto-generated by translator
void __gs_run_for0(struct filament *f) {
    // get loop var 'i' from filament
    int i = *((int*)((f->args)+0U));
    int j, k;
    // run original loop body (inner j,k loops)
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Filaments created from `parallel` loops are enqueued on servers threads in chunks of n/P , where n is the number of iterations and P is the number of processors. This is done to ensure good memory and cache performance since most loops operate on large data sets and exhibit spatial locality.

Nested `parallel` for loops that have no statements between the outermost parallel loops are optimized by pushing the creation of filaments down into the innermost

parallel loop. A nested `parallel` for has the form:

```
parallel for (i = 0; i < n; i++) {
    parallel for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Not performing any optimization would result in the outermost loop creating `n` filaments, which in turn would create `n` filaments and `n` implicit joins. Pushing filament creation down into the innermost parallel loop from the previous example would produce generated code of the following form:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        create_filament(__gs_run_for0, i, j);
    }
}

implicit_join();

void __gs_run_for0(struct filament *f) {
    int i = *((int*)((f->args)+0U));
    int j = *((int*)((f->args)+4U));
    int k;

    for (k = 0; k < n; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

Each filament now contains two arguments, one for each iteration variable; moreover, the optimization eliminates `n` implicit joins that would have normally been created by the outermost parallel for loop. This optimization can be performed on an

arbitrary number of nested `parallel` loops.

4.3 Domain Decomposition

The `divide/replicate` annotation creates a filament for the `replicate` code block on each processor and initially divides the specified arrays evenly among the filaments. The divide operation currently supports one and two dimensional division. Dividing on the first dimension results in each filament receiving a pointer to its chunk of the original array and the new length of that chunk. Suppose the following code was executed on a four processor machine:

```
divide data[size] replicate {
    i = 0;
    while (i < size) {
        do_something(data[i]);
        i++;
    }
}
```

If `size` was originally equal to eight, then each of the four filaments would receive a copy of `size` with the value of two. The value of `data` in each filament would point to the first, third, fifth, and seventh element in the original array, respectively. In the case of uneven division, the remainder is added to the `size` value of last filament created by the divide.

An array with sizes for both dimensions is divided into blocks. Figure 4.2 shows a 4x4 grid being divided among four processors. Division is optionally controlled by using the caret (^) syntax. Specifying carets in both dimensions is the same as not using the caret syntax. For example,

```
divide grid[n][m] replicate {
    statements
}
```

is the same as

Figure 4.2: Two Dimensional divide

```
divide grid[n^][m^] replicate {
  statements
}
```

Block division requires that each filament gets a copy of the first dimension of the original array with the pointers rewritten to match their corresponding position; otherwise, any subarrays starting in the middle of the second dimension will not have contiguous rows unless their first dimension pointers are updated. The translator puts code at the start of each filament created from the **divide/replicate** annotation to concurrently allocate and assign values to its subarray's first dimension copy.

Horizontal division is specified by using the caret syntax in only the first dimension. The parent thread simply divides the first dimension among the replicated threads by giving each filament a pointer to the appropriate position in the original array.

Conversely, vertical division is specified by using the caret syntax in only the second dimension. In this case, the array structure is being divided up among its columns, all filaments need copies of the first dimension in order to ensure contiguous rows. Like with block division, the translator puts code at the start of each filament to concurrently allocate and assign values to its subarray's first dimension copy.

The translator places divide operations containing a **where** clause at the beginning of each filament to determine the start and end boundaries of the division. Consider the following **divide/replicate** block:

```

divide data[size]
  where data[divide_left] != data[divide_right] replicate {
    i = 0;
    while (i < size) {
      do_something(data[i]);
      i++;
    }
  }

```

Here, the **where** clause restricts the division of **data** such that the boundary elements cannot be equal. Each filament determines its chunk boundaries concurrently. First, the special symbols **divide_left** and **divide_right** are assigned to the indices of the left-side division point. Then, each filament evaluates the **where** condition. If it evaluates to true, the division point has been successfully determined; otherwise the **divide_left** and **divide_right** indices are incremented and the process is repeated until either the expression evaluates to true or the right end of the array is reached.⁴

After each filament determines its left-side boundary index, the right-side boundary index is set to one less than the left boundary index of the filament to its right. Once both indices are determined, the divide operation continues as it would normally. Note that there are cases that lead to very unequal data allocations; for example, if the **where** clause is never true, then the first filament ends up being responsible for the entire array.

Barrier synchronization points inside a **divide/replicate** annotation are implemented using a dissemination barrier [32] for scalability. Barrier synchronization is a spin-waiting operation since the number of filaments participating in a barrier is equal to the number of server threads. The translator allocates a barrier structure for each **barrier** annotation as part of Gossamer's initialization routines. Each barrier structure is cache-aligned and contains the flags used for a dissemination barrier.

⁴The filaments testing the first and last chunks are special cases; the first chunks's start index is always the start of the array, and the last chunks's end index is always the end of the array.

4.4 Additional Synchronization

Code is placed before and after an **ordered** block to serialize execution of the code block among all filaments that are created by the same parent. Each child filament is assigned a unique identifier by their parent when it is created; after the parent assigns the identifier, it increments it by one. Thus, identifiers tell the order a filament was created with respect to its parent, and it is this exact ordering which the **ordered** annotation uses.

The translator statically allocates a data structure for each **ordered** annotation in a program. The structure contains a mutex, a condition variable, and an integer. The integer field is used to store the identifier of the last filament to complete the corresponding **ordered** block. When a filament enters an **ordered** block, it compares its identifier to the one stored in the data structure. If the identifier is that of the current filament's predecessor, then the current filament continues executing the code inside the **ordered** block; otherwise the filament blocks using the condition variable. When a filament exits an **ordered** block, it updates the identifier field and signals the condition variable. All filaments waiting on the condition variable are awakened and check the identifier field; the next filament defined by sibling order is executed if it has already reached the **ordered** block (the other filaments, if any, go back to sleep).

A **buffered** annotation is implemented by replacing all **write()** and **fwrite()** calls inside the **buffered** block with functions from the run-time library. The library routines buffer output on a per filament, per file descriptor basis. During initialization, the run-time queries the operating system for the maximum number of open file descriptors a process can have. When the first **buffered** write is encountered, an array of size max open file descriptors is allocated in the current filament descriptor. Subsequent writes are buffered to the array indexed by the write's integer file descriptor. Multiple buffered writes to the same descriptor are stored as a singly linked list of buffers. A pointer to the tail of the list is also kept in the filament descriptor in order to achieve $O(1)$ inserts. When the filament terminates, the buffer

array is traversed and buffers are flushed using the `write()` system call. Ordered `buffered` calls are handled similarly, except that before the buffers are flushed, a check is made to ensure the previous sibling filament has flushed its buffers before preceding.

If a buffered file descriptor is closed before a filament terminates, the corresponding filament buffer must be flushed before closing the file descriptor. To handle this case, the translator replaces all `close()` and `fclose()` calls in a program with functions in the run-time library, whether or not they are in a `buffered` block. The library functions check for buffered output and flush any output before closing the file descriptor.

An `atomic` annotation wraps the code block with mutexes or architecture dependent spinlocks. The number of locks used in a program is determined by partitioning the variables updated inside `atomic` blocks into sets. The translator starts by creating one set for each `atomic` block in the program; it contains the variables updated in that `atomic` block. Next, all sets are put on a work list. Then, one by one, each set is pulled from the work list and intersected with each set in the result list (initially the empty set). If the intersection is not empty (meaning the two sets share at least one variable in common), then the working set is combined with the current result set. If no common variables are found in any result set, then the working set becomes its own result set. This process is repeated until the work list is empty. All `atomic` blocks in the same result set share the same lock. This is a compromise in lock granularity between having one global lock used for every `atomic` block and having a separate lock for every variable. Using one global lock can increase lock contention when accessing `atomic` blocks from multiple filaments; using a separate lock for each variable can increase overheads [46].

As an example, consider the following atomic statements:

```
atomic {
    a++;
    b++;
}
```

```

atomic {
    b++;
    c++;
}

```

```

atomic {
    d++;
}

```

Variables **a** and **b** belong to the same **atomic** block, and therefore belong to the same set, and likewise with variables **b** and **c**. Since the first two **atomic** blocks above both contain **b**, variables **a**, **b**, and **c** are all in the same result set. The variable **d** belongs to its own result set. The first and second **atomic** blocks share the same lock and the third **atomic** containing just **d++** has its own lock; this ensures that independent **atomic** blocks do not use the same lock. The above example would use two locks and be translated as follows:

```

lock(mutex1);
a++;
b++;
unlock(mutex1);

lock(mutex1);
b++;
c++;
unlock(mutex1);

lock(mutex2);
d++;
unlock(mutex2);

```

When the only statement inside an **atomic** block is an assignment and the state-

Figure 4.3: MapReduce Hash Table

ment is mathematically associative and commutative, then the assignment is implemented by a reduction operation instead of using a lock. A reduction is implemented by giving each server thread a cache-aligned private copy of the variable that is being updated; the reduction variable is cache-aligned to ensure that a server thread's private copy does not reside in the same cache line as another server thread's copy. The assignments inside the `atomic` annotation updates the private variable on the server on which a filament is executed; this does not require locking. When the filaments perform a join operation, either implicitly or explicitly, the private values are combined and assigned to the original shared variables. This combining operation requires locking; there is one global lock per reduction variable.

It is possible to have more complex cases where variables appear in both lock-based `atomic` blocks and reduction-based `atomic` blocks. In this case, all `atomic` blocks in the same set are protected using a lock, forgoing any reductions operations that might have otherwise been selected by the translator. It is also possible for statements inside an `atomic` block to call functions that assign values to reference arguments or global variables. In this case, the `atomic` block is protected by a single global lock.

4.5 MapReduce

MapReduce computations use an associative memory space to communicate between the map and reduce phases. Gossamer supports this by means of two associative memory types: `mr_space` and `mr_list`. The `mr_space` type declares an associative memory with key and value types and the `mr_list` type holds a list of values for a particular key. Keys and values can be any C primitive type or null-terminated string. For example,

```
mr_space myspace(char *, unsigned int);
```

declares an associative memory space called `myspace` with null-terminated strings as keys and `unsigned integers` as values. The location of the declaration determines where and when the type is initialized. Global `mr_space` declarations are initialized as part of the run-time initialization routine that is called before a user's `main()` function is executed; they are freed by the run-time just before the program exits. Local `mr_space` variables are stored on the stack of the server thread. They are implicitly initialized at the start of the function in which they are declared and freed just before the function returns.

For efficiency, keys are hashed to a per-processor linked list of key/value arrays; the list is used to chain key collisions on the same processor. Figure 4.3 shows a view of the `mr_space` data structure used by the run-time. If a key's array fills up, `realloc()` is used to allocate additional space.

The `mr_put()`, `mr_getkey()`, and `mr_getvalue()` operations are implemented using type-specific library routines. In particular, the Gossamer run-time contains instances of these operations for each possible `mr_space` type, and the translator selects the appropriate one to use by examining the first argument to `mr_put()`, `mr_getkey()`, and `mr_getvalue()`. This allows Gossamer to provide type-optimized versions of all MapReduce operations.

In the map phase, key/value pairs are put into an `mr_space` using the `mr_put()` operation. Since each hash entry contains a per processor list, the `mr_put()` operation is lock-free. In the reduce phase, keys are retrieved using the `mr_getkey()`

operation. The first call of `mr_getkey()` signals the end of the map phase for a particular `mr_space`, and any further use of `mr_put()` will result in a run-time error. The `mr_getkey()` operation is implemented by gathering all values for a given key from each processor and combining them into a single `mr_list` type; the key/value arrays are then removed from the corresponding `mr_space`. The `mr_getkey()` operation potentially modifies data shared across server threads, so it needs to lock the associative memory data structure. The `mr_getvalue()` operation is used to retrieve individual values from an `mr_list`; since each list is distinct, no locking is required as long as the entire list is processed by at most one filament⁵.

4.6 Static Program Analysis and Feedback

The translator performs flow-insensitive static analyses to verify that annotations meet the semantic requirements defined in Chapter 2, and to detect common mistakes that are often made in parallel programs. The analyses are necessarily conservative, and they do not perform pointer alias analysis. Additional run-time analyses are described in Section 4.7.

The static analyses done by the translator look for expressions that violate semantic rules of an annotation. Loops annotated with `parallel` need to follow a simple iterative format—variable assignment, variable condition, variable increment. Moreover, any variables used within the loop header must not be modified within the loop body. For example, in

```
parallel for (i = 0; i < n; i++)
    n = some_calculation(i, n);
```

the iteration variable `n` is modified within the `parallel` loop body. If `n` was declared locally, it would be private to each filament and the parent thread would never see any updates to its copy of `n`. If `n` was global in scope, assigning to `n` leads to potential race condition.

⁵Currently, the programmer is responsible for ensuring any `mr_list` returned from `mr_getvalue()` is used within a single filament.

Arrays passed to the `divide/replicate` annotation need to evaluate to a pointer type, and the divide dimension variables need to evaluate to a numeric type. Similarly, the argument of a `copy` annotation must be a pointer type, and the dimensions must evaluate to an numeric type. The `buffered` annotation's size option must evaluate to a numeric type.

One common programming mistake is to parallelize a loop that has a loop-carried dependency, which occurs when one iteration of a loop writes a value that is read by a later iteration. For example, consider the following annotation of matrix multiplication:

```
parallel for (i = 0; i < n; i++)
  parallel for (j = 0; j < n; j++)
    parallel for (k = 0; k < n; k++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

The shared variable `C` (including the indices `i` and `j`) appears on both the left-hand and right-hand sides of the assignment statement in the inner-most parallel loop; moreover, the of `C[i][j]` at iteration `k` depends on the value of `C[i][j]` at iteration `k-1`. The translator will display a warning message of a possible loop-carried dependency involving the shared variable `C` and indicate the source file and line number of the offending assignment.

Another common mistake is updating a shared variable without using an `atomic` statement inside a filament. Consider the following code fragment from the n-queens example in Section 3.2:

```
if (row == n) {
    solutions++;
    return;
}
```

Here, the shared variable `solutions` is assigned a new value inside the forked recursive function `putqueen()`, but it is not protected with an `atomic` statement. The translator will report a possibly unsafe update and indicate the source file and line number of the offending assignment.

4.7 Run-Time Program Analysis and Performance Profiling

Gossamer will add run-time checks into the code generated from annotations if the command line flag `--gs-runtime-checks` is passed to the translator. The more sophisticated user can turn on additional run-time performance profiling analyses by passing various profiling flags to the translator; see Appendix B for details. Since these checks are performed at run-time, they add overhead and thus are recommended only for debugging and testing purposes.

Gossamer's barrier implementation requires that number of filaments participating in a barrier be known a priori, otherwise filaments can deadlock. Thus, barriers used outside a `divide/replicate` filaments are not allowed, but cannot always be detected at compile time. Turning on run-time checks causes any `barrier` annotation used outside a `divide/replicate` filament to throw an exception and abort, displaying the source file and line number of the offending `barrier` annotation.

Similarly, using an `ordered` annotation inside a loop might result in deadlock. Once a filament executes an `ordered` block, it indicates to the run-time that it has completed its ordered operation and the run-time signals the next filament to run. If `ordered` is used inside a loop, it will never be signaled more than once to run, thus resulting in deadlock. If run-time checking is enabled, the translator inserts checks to verify that all `ordered` annotations are executed only once per filament; if not, the run-time will throw an exception and abort, displaying the source file and line number of the offending `ordered` annotation.

Dimension sizes used in `divide/replicate` and `copy` and buffer sizes used in `buffered` annotations must all evaluate to a positive number at run-time. Offending annotations will cause the run-time to throw an exception and abort, displaying the source file and line number of the offending annotation.

The translator can also instrument annotations for performance profiling. By using the command line flag, `--gs-profile-totals`, a summary of the annotations used in a program will be displayed when it exits. An example summary for the N-Queens program in Section 3.2 is shown below.

Gossamer Program Summary - Totals	
Total Filaments:	32
Total Forks:	32
Total Joins:	4
Total Parallel Loops:	0
Total Divide/Replicate Blocks:	0
Total Barrier Calls:	0
Total Atomic Statements (lock):	8
Total Atomic Statements (reduction):	0
Total Copied Arguments:	32
Total Average Size of Copied Argument:	196 bytes
Total Size of Copied Arguments:	6.125 KB

In addition to program totals, individual annotation counts can be enabled via the `--gs-profile-annotation-counting` command line flag. When an annotation is encountered at run-time, a count variable is incremented. The source file and line number are recorded along with the count. The output is sorted by annotation usage count. For example, annotation count profiling output for the quicksort program shown in Section 3.1 is shown below.

Annotation Counts			
Count	Annotation	Source File	Line Number
16	fork	gs_quicksort.gsc	56
16	fork	gs_quicksort.gsc	57
16	join	gs_quicksort.gsc	58

4.8 Combining Multiple Levels of Parallelism

Any significant application will contain multiple opportunities for concurrency, and the performance of a large application may depend on how the multiple levels of

concurrency are composed at run-time. Gossamer’s run-time supports both multiplexing and spatial distribution of recursive, task, iterative, and domain decomposition filaments over the total set of available processors at run-time. By default, filaments are multiplexed over all the available server threads without bias towards what type a filament is being enqueued. For a spatial distribution, users allocate some number of processors to each type of filament before the program starts using environmental variables. Each server thread will then only enqueue and execute the type of filament assigned to that server thread.

Bzip2 is an example of a program that has multiple levels of concurrency. In addition to compressing independent blocks of data in parallel, as shown in Section 3.3, Bzip2’s compression algorithm can exploit recursive parallelism. In the first step of the compression, Bzip2 performs a transformation, called the Burrows-Wheeler transform (BWT) [16]. BWT permutes the order of the characters so that the transformed string will have several places where a single character is repeated multiple times: in a row; this is done to optimize the remaining steps in the compression algorithm. The BWT transformation can be implemented using quicksort, which lends itself to recursive parallelism. Now there are two levels of concurrency: one at the outer level in the algorithm and one at the inner level of the first step in compression.

To experiment with this, we had the run-time reserve half the server threads for task parallelism to perform the block level concurrency and the other half for recursive parallelism to perform the BWT. The results were less than satisfactory since the quicksort step of the BWT accounts for only around ten percent of the total time it took to compress a block. This resulted in excessive idling for the server threads reserved for the BWT when it would be more efficient for all server threads just do the outer level parallelism. Because our experimental machine has only 8 cores, we were unable to exhaust all the concurrency on the outer level of the algorithm. However, if there were enough processing cores to saturate the outer level concurrency, then it would be practical to use the inner level concurrency on the remaining cores.

CHAPTER 5

PERFORMANCE

In this chapter we analyze Gossamer’s performance on ten applications: quicksort, n-queens, bzip2, matrix multiplication, sparse matrix multiplication, Jacobi iteration, run length encoding, word count, multigrid, and gravitational n-body. These tests show the performance of all 15 annotations available in Gossamer. We benchmark the set of applications using an 8-core Intel Xeon shared-memory workstation and an SGI Altix supercomputer using up to 32 cores. We also measure application-independent overhead and analyze Gossamer’s performance relative to Cilk++ and OpenMP in Section 6.6.

5.1 Testing Methodology and Hardware

For each application we developed a sequential program and a Gossamer program. The sequential program is used to compute speedups of the Gossamer programs. To ensure fairness and accuracy of each comparison, all application code is as similar as possible. The sequential programs were written without any parallel constructs, and the Gossamer programs were written using only Gossamer’s annotations without any explicit parallel constructs (e.g. explicit calls to the Pthreads library).

The benchmarks were run on an Intel Xeon E5405 (Harpertown) workstation, which has two quad-core processors running at 2.0 GHz with 32 kbyte of L1 instruction cache and 32 kbyte of L1 data cache per core. Each quad-core processor has 12 MB of L2 Cache and a 1333 MHz Front Side Bus frequency. The workstation has 8 Gbyte (8x1 Gbyte) of main memory. The operating system was Ubuntu Linux 9.10. The applications were compiled with the GNU C compiler version 4.4.1 using -O3 optimization. All timing tests were run in single-user mode.

The benchmarks were also run on a Silicon Graphics Altix 4700 multiprocessor

supercomputer, which has 256 dual-core Itanium2 processors running at 1.6 GHz with 16 kbyte of L1 cache, 1 Mbyte of L2 instruction cache, 256 kbyte of L2 data cache, and 24 Mbyte of L3 cache. The physical layout of the supercomputer consists of 128 nodes, each having two dual-core processors sharing 4 Gbyte (2 Gbyte per processor) of memory with total of 1024 Gbyte of main memory. Memory access is non-uniform and accessing memory outside the local node must travel through the interconnect, causing an order of magnitude slower access times. The operating system was SUSE Linux Enterprise Server 10. The applications were compiled with the GNU C compiler version 4.1.2 using `-O3` optimization. All timing tests were run through the supercomputer’s batch processing interface.

The execution times are for the algorithmic portions of each benchmark application. Reported results are the median of ten test runs (the average of the two middle times), as reported by `gettimeofday()`, rounded to the nearest hundredth of a second. Data sets were chosen such that all execution times were within the accuracy of `gettimeofday()` on the host operating system. Measurements of the total execution time yield very similar speedup numbers. This is because most programs do relatively little initialization—or the initialization is part of the timed portion of the algorithm. Applications with poor speedup numbers for overall execution time tend to have a large initialization step, e.g. reading a large file in its entirety before processing any input.

5.2 Application Performance on Intel Xeon E5405 (8-core)

The execution times for the benchmark applications on the Intel Xeon workstation are shown in Table 5.1; speedups for the benchmark applications are shown in Table 5.2. Gossamer demonstrates excellent speedups on most benchmarks.

Quicksort is a recursive application that uses the `fork` and `join` annotations. Each quicksort step partitions its array and then recursively sorts each partition concurrently. Quicksort does not scale well because array partitioning is a sequential bottleneck, and the function has to be forked several times before all processors have

Application	Parameters	Execution Time (seconds)				
		Sequential	1-CPU	2-CPU	4-CPU	8-CPU
Quicksort	n = 100,000,000	17.86	17.41	9.04	6.45	5.65
N-Queens	n = 14	9.29	10.41	5.01	2.42	1.31
Bzip2	file = 256.0 MB	46.06	46.37	23.88	12.58	6.41
Matrix Multiplication	n = 4096	198.51	193.06	96.18	47.42	23.74
Sparse Matrix Multiplication	n = 2048 fill = 15%	211.87	211.36	106.45	60.88	59.35
Jacobi Iteration	grid = 1024x1024 iterations = 65536	277.99	278.04	124.89	67.99	38.94
Run Length Encoding	file = 4.0 GB	6.23	6.48	3.23	1.62	0.84
Word Count	files = 1024 file size = 2.0 MB	124.06	125.08	61.32	35.48	19.05
Multigrid	grid = 1024 iterations = 128 cycles = 128	8.51	8.50	4.19	2.23	1.42
N-Body	bodies = 32768 iterations = 16384 tree builds = 16	193.10	193.05	97.07	49.15	25.36

Table 5.1: Execution Times on Intel Xeon

Application	Parameters	Speed Up (relative to sequential)			
		1-CPU	2-CPU	4-CPU	8-CPU
Quicksort	n = 100,000,000	1.026	1.98	2.77	3.16
N-Queens	n = 14	0.89	1.86	3.85	7.11
Bzip2	file = 256.0 MB	0.99	1.93	3.66	7.19
Matrix Multiplication	n = 4096	1.03	2.06	4.19	8.36
Sparse Matrix Multiplication	n = 2048 fill = 15%	1.00	1.99	3.48	3.57
Jacobi Iteration	grid = 1024x1024 iterations = 65536	1.00	2.23	4.09	7.14
Run Length Encoding	file = 4.0 GB	0.96	1.93	3.85	7.44
Word Count	files = 1024 file size = 2.0 MB	0.99	2.02	3.50	6.51
Multigrid	grid = 1024 iterations = 128 cycles = 128	1.00	2.03	3.82	6.00
N-Body	bodies = 32768 iterations = 16384 tree builds = 16	1.00	1.99	3.93	7.61

Table 5.2: Application Speedups on Intel Xeon

work. It has been shown that quicksort has a theoretical limit of about 10.5x speedup for 16 or more processors [40].

N-Queens is another recursive application that uses the `fork`, `join`, `copy`, and `atomic` annotations. It places a queen chess piece on the chess board (represented by a two dimensional array) and uses `fork` and `join` to concurrently check the rest of the board. The `copy` annotation is used to ensure that each filament receives its own copy of the chess board. The global variable storing the number of solutions is protected by an `atomic` annotation to ensure that updates to that variable are executed one at a time. Pruning of recursive forks and relatively low contention for the global variable holding the number of solutions allows n-queens to get excellent speedups: 1.86, 3.85, and 7.11 on 2, 4, and 8 processors, respectively.

Bzip2 compresses independent blocks of data using `fork` and `join` for task parallelism. The compressed output is sequentialized using `ordered`. It scales well as long as the compression time of each block is approximately the same, because this keeps the pipeline of concurrently compressing blocks full. Using an input file of randomly generated text, Gossamer’s bzip2 implementation achieves speedups relative to the sequential program of 1.93, 3.66, and 7.19 on 2, 4, and 8 processors, respectively.

Matrix multiplication is an example of an embarrassing iterative parallel algorithm that uses the `parallel` annotation to concurrently execute the outer-most loop of n iterations. It achieves excellent speedups: 2.06, 4.19, and 8.36 for 2, 4, and 8 processors, respectively. The super-linear speedups are due in part to cache effects, because moving from 4 to 8 cores doubles the amount of shared L2 cache that is available. Although it is possible also to parallelize the middle loop in the algorithm, the results are not as favorable due to the overhead of creating n^2 filaments instead of n . For $n = 1024$, parallelizing both loops results in an 8 processor speedup of only 4.52.

Sparse matrix multiplication is an example of matrix multiplication where most of the entries are zero, and thus do not have to be explicitly stored in the matrix. Substantial memory reductions can be realized by storing only the non-zero entries, but at the cost of added synchronization needed to protect updates to the complex

data structure used to store the sparse entries. Gossamer’s sparse matrix multiplication achieves speedups of 1.99, 3.48, 3.57 for 2, 4, and 8 processors, respectively. The performance drop-off on 8 processors is due to the `atomic` update of the result matrix `C`; the `atomic` operation uses a global lock and becomes a bottleneck when the lock has to be passed back and forth between the two processor caches. A modified version of sparse matrix multiplication which uses a lock per row achieves speedups of 2.04, 4.09, 7.61 for 2, 4, and 8 processors, respectively. The `atomic` annotation would have to be modified in order to achieve this locking technique and is left for future work.

Jacobi iteration is an example of domain decomposition where two grid arrays are divided among the processing cores using the `divide/replicate` annotation. It scales well relative to the sequential program, with speedups of 2.23, 4.09, and 7.14 on 2, 4, and 8 processors, respectively.

Run length encoding also uses domain decomposition, and it uses a `where` clause to constrain where the data is divided. In order to obtain the best possible compression, the divide must not occur at boundaries where the elements are equal to one another. The compressed output is buffered using the `buffered` annotation. The application reads a 4 GB file of randomly generated data into an array and then compresses each subarray independently. The compression algorithm gets excellent speedups of 1.93, 3.85, and 7.44 on 2, 4, and 8 processors, respectively. However, run length encoding shows relatively poor speedups for overall execution time—1.35 on 4 cores and 1.42 on 8 cores—because the entire 4.0 GB file is read before parallel execution begins.

Word count calculates the number of occurrences of each word in a set of data files. It uses `fork` and `join` for concurrency and synchronization and creates a filament for each input file. Word count also achieves good speedups relative to the sequential program: 2.02, 3.50, and 6.51 on 2, 4, and 8 processors, respectively. The small amount of performance drop-off on 8 processors is due to I/O bottleneck of the system disk¹.

¹Even though the input files were stored on a Linux software RAID-0 disk array composed of

Multigrid approximates a partial differential equation using domain decomposition. It varies the granularity of the grid in order to increase the rate of convergence and uses a four-level V cycle. Multigrid converges much faster than the Jacobi iteration shown above. For a grid size of 1024x1024, Jacobi iteration takes over 700,000 iterations to converge, whereas multigrid only takes 16,384 iterations to converge. The lower number of iterations comes at the cost of additional barrier synchronization, which is required each time the grid changes granularity. This additional synchronization limits the scalability of multigrid. It scales well up to 4 processors before showing signs of synchronization overhead with speedups of 2.03, 3.82, and 6.00 on 2, 4, and 8 processors, respectively.

The N-Body application uses the Barnes-Hut method to track the motion of a group of astronomical bodies that are subject to gravitational interaction. It uses domain decomposition to divide up the forces, masses, positions, and velocities of the astronomical bodies and calculate the motion of the bodies over some number of time intervals. It shows excellent speedups: 1.99, 3.93, and 7.61 on 2, 4, and 8 processors, respectively. By carefully choosing the right number of sequential quadtree builds required by the Barnes-Hut algorithm, one can achieve a good balance between application performance and the accuracy of the astronomical body movements.

5.3 Application Performance on SGI Altix 4700 (32-core)

The execution times for the benchmark applications on the SGI Altix 4700 are shown in Table 5.3; speedups for the benchmark applications are shown in Table 5.4. Again, Gossamer demonstrates excellent speedups on some benchmarks, while others yield poor speedups and explained below.

Quicksort show poor speedups on the NUMA-memory supercomputer. As before, array partitioning is a sequential bottleneck, and the function has to be forked several times before all processors have work. Moreover, moving past 4 processors causes sections of the array being sorted to have to be moved back and forth between

two 10,000 RPM SATA disks, by 8 threads the bandwidth needed for 8 simultaneous I/O streams was starting to limit overall performance.

Application	Parameters	Execution Time (seconds)						
		Sequential	1-CPU	2-CPU	4-CPU	8-CPU	16-CPU	32-CPU
Quicksort	n = 100,000,000	31.76	31.74	17.69	13.10	16.86	15.26	15.68
N-Queens	n = 14	29.11	28.98	14.71	8.06	6.09	4.34	4.39
Bzip2	file = 256.0 MB	57.82	58.68	30.64	15.60	8.24	4.48	2.67
Matrix Mult.	n = 4096	150.29	149.29	76.08	37.36	29.38	14.03	8.50
Sparse Matrix Multiplication	n = 2048 fill = 15%	196.80	197.63	108.08	67.19	74.19	93.69	101.41
Jacobi Iteration	grid = 1024 iterations = 512	59.21	59.12	29.02	17.44	15.17	13.33	11.44
Run Length Enc.	file = 4.0 GB	22.23	24.90	12.55	6.23	4.93	2.30	1.39
Word Count	files = 1024, file size = 2.0 MB	127.24	127.46	62.16	47.71	43.62	20.28	34.93
Multigrid	grid = 1024 iterations = 128 cycles = 128	29.38	30.01	14.382	7.55	5.05	4.32	6.67
N-Body	bodies = 32768 iterations = 16384 tree builds = 16	149.41	144.04	73.30	38.22	21.42	14.72	16.00

Table 5.3: Execution Times on SGI Altix 4700

Application	Parameters	Speed Up (relative to sequential)					
		1-CPU	2-CPU	4-CPU	8-CPU	16-CPU	32-CPU
Quicksort	n = 100,000,000	1.00	1.80	2.42	1.88	2.08	2.02
N-Queens	n = 14	1.00	1.98	3.61	4.78	6.71	6.64
Bzip2	file = 256.0 MB	0.99	1.89	3.71	7.02	12.92	21.68
Matrix Multiplication	n = 2048	1.01	1.98	4.02	5.12	10.71	17.68
Sparse Matrix Multiplication	n = 2048 fill = 15%	0.99	1.821	2.93	2.65	2.10	1.94
Jacobi Iteration	grid = 1024, iterations = 512	1.00	2.04	3.40	3.90	4.44	5.18
Run Length Encoding	file = 4.0 GB	0.89	1.77	3.57	4.51	9.68	16.02
Word Count	files = 1024, file size = 2.0 MB	1.00	2.05	2.67	2.92	6.27	3.64
Multigrid	grid = 1024 iterations = 128 cycles = 128	0.98	2.04	3.89	5.82	6.79	4.40
N-Body	bodies = 32768 iterations = 16384 tree builds = 16	1.04	2.04	3.91	6.97	10.15	9.34

Table 5.4: Application Speedups on SGI Altix 4700

nodes, since the entire array is allocated in the main filament before sorting takes place. This makes the memory bandwidth the dominant bottleneck at 16 and 32 processors.

N-Queens suffers from memory access issues with the chess board similar to those for quicksort. It is still able to achieve some speedup improvements for 8 and 16 processors, but starts declining by 32 processors. This is due to the copy of the chess board made when a filament is created. The memory is allocated on the node in which the filament is created on, and with round-robin scheduling of recursive filaments, a filament may or may not run on the same node on which they were created. If it does not run on the same node, the chess board will have to be transferred over the NUMA backplane to the node on which the filament is being run incurring a longer memory access penalty.

Bzip2 shows good speedups: 1.89, 3.71, 7.02, 12.92, and 21.68 on 2, 4, 8, 16, and 32 processors, respectively. Each input block is only 900 kbytes and easily fits into the processor cache; and each block has to be moved only once from the memory bank of the node running the main filament to the processor running the filament that compresses the block.

Matrix multiplication shows good speedups on up to 32 processors. The application allocates all matrices in the main filament. This requires all processors not on the node of the main filament to have to perform remote memory accesses for each multiplication. An optimization would be for each filament to allocate the rows of its **A** and **C** matrices on its own node, and for each processor node to have a copy of the **B** matrix. This would ensure that each filament would not have any memory accesses outside its processor node.

Sparse matrix multiplication strains to achieve any speedup past 4 processors. The node to node memory transfers of the global lock compound the bottleneck even further. Speedups can be improved by applying the one lock per row technique mentioned earlier, but the sparse data structure still hinders overall performance.

Jacobi iteration shows positive speedups on up to 32 processors, but struggles to scale linearly due to remote memory accesses to the border rows, which lie on

different nodes.

Run length encoding shows good speedups: 1.77, 3.57, 4.51, 9.68, and 16.02 on 2, 4, 8, 16, and 32 processors, respectively. Performance is somewhat restricted by having the entire 4 Gbyte input file read into memory allocated in the main filament. This requires each filament not on the main filament's node to perform expensive node to node memory transfers. Unlike the N-Queens program, RLE memory transfers are much larger, and they have greater impact on overall program performance.

Word count shows poor speedups mainly due to accessing 1,024 input files over a shared network filesystem that is run simultaneously with hundreds of other experiments on the supercomputer. Limits in rotational disk speeds, along with added network latency caused erratic application performance and poor overall speedups.

Multigrid shows good speedups on up to 4 processors, but fails to scale linearly past 8 processors. Remote memory accesses of border rows and the additional barrier synchronization both contribute to its scalability issues.

The n-body simulation shows some speedup on up to 16 processors but starts to tail off at 32 processors. The n-body calculation accesses four shared floating point arrays and a shared quadtree that are allocated in the main filament. The quadtree is built and maintained in the main filament, but all filaments access the data structure during the force calculation step causing remote node to node memory transfers each time the quadtree is rebuilt.

5.4 Application-Independent Gossamer Overheads

Gossamer programs have two types of overheads: initialization overheads that occur before the user's program starts executing, and run-time overheads that occur because of the annotations contained in a program. As the ratio of computation to communication increases in a program, the overheads imposed by Gossamer become relatively less significant. Measuring application-independent overheads allows programmers to obtain an idea of how much computational work is required for the

Operation	1-CPU	2-CPU	4-CPU	8-CPU
Gossamer Initialization	2,603,052	2,804,688	3,014,055	3,647,604

Table 5.5: Run-Time Initialization Overheads (CPU Cycles)

Operation	Time (μ s)	Ops/sec
Filament Creation	0.868	1,152,074
Pthread Creation	39.6	25,253

Table 5.6: Filament Creation Overheads (Intel Xeon workstation)

Gossamer overheads to be relatively negligible.

Gossamer initialization overheads involve creating server threads using the Pthreads library and setting up the data structures used by the run-time. Table 5.5 shows the number of processor cycles required for run-time initialization on up to 8 processors. The results are the median of ten test runs. The numbers of cycles are used since the startup times are smaller than the accuracy of `gettimeofday()`.

Filament creation is a general run-time overhead that occurs regardless of application. Table 5.6 shows both the time it takes to create a single filament and the number of filaments that can be created per second; Pthreads is also shown for comparison. The Intel Xeon workstation running at 2.0 GHz is capable of creating approximately 1.1 million filaments per second. Filament creation is over an order of magnitude better than Pthreads.

5.5 Summary

We have measured the performance of 10 different applications using an 8-core shared-memory workstation and 32 cores of a NUMA-memory supercomputer. In general, all applications show good speedups on the shared-memory workstation, with the exception of quicksort. Speedups on the NUMA-memory supercomputer yield less positive results, because most of the benchmark applications are memory intensive, and the cost of remote memory access limits overall performance.

CHAPTER 6

RELATED WORK

There is a wealth of related research on thread packages and parallel programming languages. The most common way to write parallel programs is to use a sequential programming language with a threads library. Other approaches include using a compiler that automatically parallelizes sequential code, using a new parallel language that provides implicit parallelism through high-level abstractions, using a specialized tool to exploit vector processing units of general purpose graphic processing units (GPGPU), and using a sequential language with parallel annotations to specify concurrency and synchronization. These approaches are described below, then Gossamer’s performance is compared to that of Cilk++ and OpenMP.

6.1 Thread Packages and Libraries

General purpose thread packages, such as POSIX threads (Pthreads) [34], provide the programmer with a low-level, natural thread abstraction, and many of the usual concurrent programming primitives. Pthreads is a fairly heavyweight threads package, creating a separate run-time stack for each thread and supporting preemption of threads to provide fairness. However, having stacks and preemption requires context switching code, which is inherently machine dependent and costly in terms of overall performance. This makes it difficult to efficiently support fine-grain parallelism. For example, using Pthreads to implement a fine-grain matrix multiplication in which a new thread is created for each innermost loop iteration would spend more execution time in thread creation and context switching overheads than it would in the actual calculation itself. In addition, the per thread stacks can waste valuable cache space, which could otherwise be used to hold actual data. To address these issues, Gossamer only has one stack per processor and does not context switch or

preempt threads.

Intel Threading Building Blocks (TBB) [35] is C++ template library that provides explicit parallelism by specifying task patterns on shared memory architectures. The programmer defines application level parallelism, and the library handles machine issues such as thread, memory, and cache management. Compared to annotation based languages like Gossamer or Cilk, TBB requires a fair amount of explicit parallel code to be written by the programmer.

Filaments [42] is a threads packages that supports fine-grain parallelism. It has several things in common with Gossamer: threads have no private stacks, threads do not context switch and are not preempt-able, run-time system uses per processor run queues for work scheduling, support for fork/join and iterative threads, and provide an interface for the C programming language. However, Filaments does not allow both fork/join and iterative threads to be used in the same program, and it does not support nested parallelism. Another key difference is that Filaments is just a static C library, providing no translator or annotations to aid the programmer in application development.

Apple's Grand Central Dispatch (GCD) [3] comprises new language features, run-time libraries, and operating system support for concurrent code execution on multicore hardware. GCD defines block objects, which are a C-level syntactic and run-time feature that allow you to compose function expressions that can be passed as arguments, optionally stored, and used by multiple threads. Once blocks are defined for a program, all concurrency and synchronization must be programmed explicitly using the low-level run-time library API.

6.2 Parallelizing Compilers

A quite different way to parallelize a program is to use a parallelizing compiler. Such a compiler automatically extracts parallelism from sequential programs and produces correctly synchronized parallel programs. To do so, the compiler performs a dependence analysis on the program being compiled [6, 60, 45]. This tells

the compiler which parts of the program are independent and good candidates for parallel execution and which parts are dependent on each other and require synchronization or sequential execution. This frees the programmer from having to learn how to write a parallel program. Such compilers include Intel C compiler [10], Paradigm [29], Polaris [11] Rice Fortran D [1], SUIF [59], and Vienna Fortran [8];

Although the quality of parallelizing compilers has improved in the past several decades, fully automatic parallelization of programs written in a sequential programming language remains a notable challenge for compiler writers. Automatic parallelization often need complex program analysis and knowledge of unknown run-time factors (e.g. input data sizes and ranges) during compilation. This limits the range of applications a parallelizing compiler can handle. Additionally, most compilers for automatic parallelization target Fortran programs, because Fortran makes stronger guarantees about pointer aliasing than languages such as C.

6.3 Parallel Languages

At the other extreme from parallelizing compilers are parallel programming languages such as Chapel [20], Erlang [38], Fortress [56], X10 [54], and ZPL [17]. These languages are designed to provide implicit parallelism through their high-level abstractions. This makes parallel programs easier to write, debug, and maintain, since all the parallelism is implied through the programming language model. Unfortunately, this can be impractical for existing applications since it would require the entire application to be rewritten using the new language.

Chapel [20] is a multithreaded parallel programming language developed by Cray Inc. It supports a multithreaded execution model via high-level abstractions for data parallelism, task parallelism, nested parallelism, and features for controlling locality. The language consists of all of the features that you would traditionally find in a sequential programming language such as types, variables, expressions, statements, functions, and so forth. Chapel has support for iterators or functions that generate a stream of values during their execution rather, which is useful for

programming loops. Chapel’s task parallel features support the ability to create a number of tasks running concurrently. Tasks coordinate with one another through the use of synchronization variables that support a **full/empty** state in addition to their normal value. By default, reads and writes on these synchronization variables block until the variable is full (for read) or empty (for write), which provides a more elegant means of coordinating than traditional locks and semaphores. Chapel’s data parallel features are built around a rich set of array types including multidimensional, strided, sparse, and associative arrays. Parallel loops can be used to iterate over an array’s indices or elements, and scalar functions can be mapped to array elements in parallel.

Erlang [38] is a general-purpose concurrent programming language and run-time system. All concurrency in Erlang is explicit and follows the Actor model [33]; the “actors” are Erlang processes. Erlang provides language-level features for creating, managing, and communicating with processes. Erlang processes have no shared state between them and communicate via asynchronous message passing. Every process has a mailbox, which is a queue of messages that have been sent by other processes but have not yet been processed. The receive primitive is used to retrieve messages that match specified patterns from mailboxes. Erlang can also be run in a distributed environment and has various error detection primitives to support automatic fail-over to other nodes. Erlang supports soft real-time scheduling and garbage collection, and it allows program code to be changed in a running system on-the-fly without needing to go offline.

Fortress [56] is a multithreaded programming language designed for high-performance computing (HPC) with implicit parallelism, transactions, flexible mathematical syntax, and static type-checking (but with type inference). It is intended to be a successor to Fortran, but the language is not designed to be similar to Fortran. Fortress syntax is designed to resemble mathematical syntax as much as possible, so that anyone solving a problem with math in its specification can write a program that can be more easily related to its original specification. Fortress is also designed to be both highly parallel and to have rich functionality contained

within libraries. For example, the `for` loop is a parallel operation, which will not always iterate in a strictly linear manner depending on the underlying software and hardware. The looping constructions, together with the library, turns iteration inside out; instead of the loop specifying how the data is accessed, the data structures specify how the loop is run, and aggregate data structures are designed to break into large parts that can be effectively scheduled for parallel execution. Transactions are provided so that programmers are not faced with the task of determining lock orders. Fortress's limitations revolve around its mathematical syntax. If the problem being solved isn't easily expressed in the form of Fortress's mathematical notion, writing a Fortress program may be too cumbersome and time consuming to be practical.

X10 [54] is a second generation, type-safe, explicitly concurrent, object-oriented language that extends sequential Java with a handful of constructs for concurrency and distribution. It introduces a clustered address space to deal with data distribution. It is targeted to future low-end and high-end systems with nodes that are built out of multicore SMP chips with non-uniform memory (NUMA) hierarchies, and interconnected in scalable cluster configurations. To be able to create lightweight threads locally or remotely, X10 introduces the notion of asynchronous activities. The language constructs `async`, `future`, `foreach`, `ateach`, `finish`, `clocks` and atomic blocks are designed to coordinate asynchronous activities in an X10 program. It provides support for recursive fork/join parallelism and a very simple form of atomic blocks in lieu of locks for mutual exclusion. Tasks in X10 have the customary parent and child relationship with regards to threads. To prevent deadlocks between tasks, a child task cannot wait for a parent to finish, but a parent can wait for a child using the `finish` command. X10 is most suitable for clusters of machines and requires the programmer to explicitly specify large portions of parallel codes.

ZPL[17] is an array programming language designed for engineering and scientific applications. ZPL uses the array abstraction to implement a data parallel programming model. ZPL programs exclusively use implicit parallelism, and parallelism arises from the semantics of the array operations. Parallel arrays are distributed

across the processors. Scalar variables and indexed arrays are replicated across the processors. Replicated data is kept consistent and can be viewed by the programmer as a single object. ZPL is translated into a conventional abstract syntax tree representation on which program analysis and program optimizations are performed. ANSI C code is generated and is then compiled using a standard C compiler and linked with run-time libraries.

6.4 Stream and Vector Processing Packages

General purpose graphics processing units (GPGPUs) are high-performance many-core processors capable of very high computation and data throughput. Applications targeting GPGPUs can often achieve speedups of orders of magnitude versus optimized CPU-based implementations. GPGPUs are essentially stream processors and can operate in parallel by running a single computation on many data records in a stream at once.

Programming languages and tools supporting GPGPUs include Intel Ct [27], ATI Stream SDK [2], Nvidia CUDA [50], OpenMP to GPGPU [39], and RapidMind [53]. While these approaches can produce highly efficient programs, they are only effective at tackling problems that can be solved using stream processing. This limits the range of computations this approach can handle.

Intel Ct [27] is another C++ template library that includes the compiler, threading run-time, and memory manager for vectorized code generation that makes use of both conventional CPUs and general purpose graphic processing units (GPGPU). Ct code is dynamically compiled, so the run-time tries to aggregate as many smaller tasks or data parallel work quanta so that it can minimize threading overhead and control the granularity according to run-time conditions. With Ct, the programmer uses vectors of data called TVECs, or throughput vectors. All operators on TVECs are implicitly parallel. The Ct API specifies parallel tasks that are dependent on each other via collections of Ct operations on TVECs. Ct also introduces two abstractions for task parallelism. The basic tasking unit in the Ct run-time is

a future, an idea that goes back to MultiLisp [30]. A future is basically a parallel-ready procedure (a function and its arguments) that may or may not be executed immediately. This is similar to Gossamer’s concept of a filament. Ct also incorporates a structured task parallel abstraction called hierarchical synchronous tasks (HSTs). With HSTs, Ct can support various other forms of structured parallelism, including pipeline and fork-join patterns. Ct results in good speedups, but requires a data set and computation that fits the TVEC model.

ATI Stream SDK [2] includes the stream optimized version Brook+ of the Brook language [14], a variant of ANSI C developed by Stanford University. The SDK contains the AMD Core Math Library (ACML) and AMD Performance Library (APL) with optimizations for the AMD FireStream GPGPU. ATI Stream SDK has good performance, but it is limited to GPGPU computations. Additionally, most of setup and bookkeeping required to do the GPGPU computations requires manual effort from the programmer.

NVIDIA’s Compute Unified Device Architecture (CUDA) [50] is the compute engine built in NVIDIA GPUs. Programmers use C for CUDA, compiled through a PathScale Open64 C compiler to code algorithms for execution on the GPU. CUDA gives developers access to the native instruction set and memory of the parallel computational elements in CUDA GPUs. CUDA does not support recursive functions, so all recursive functions must be converted to loops. Although CUDA automates thread management, programmers must analyze their problem and determine how best to divide the data into smaller chunks for distribution among thread processors. This requires programmers to write explicit code to decompose the data into a layout suited for CUDA. The bus bandwidth and latency between the CPU and the GPU may also be a bottleneck. Memory has to be allocated in both main memory and GPU memory. If an application is repeatedly moving memory back and forth between main memory and the GPU, performance can suffer.

OpenMP to GPGPU [39] presents a compiler framework for automatic source-to-source translation of standard OpenMP applications into CUDA-based GPGPU applications. The framework consists of two phases. First, the OpenMP stream

optimizer transforms traditional CPU-oriented OpenMP programs into OpenMP programs optimized for GPGPUs, using three high-level optimization techniques: parallel loop-swap and matrix transpose techniques for regular programs, and loop collapsing for irregular programs. Second, the framework converts the output of the first phase into CUDA-based GPGPU programs and exploits CUDA-specific features.

The RapidMind Multicore Platform [53] is a framework for expressing data parallel computations from within C++. It is a commercialized version of Sh [43], a metaprogramming language for programmable GPGPUs developed at the University of Waterloo. Although its programming model is primarily data parallel, it is generic enough to express task parallel operations. The platform currently targets multicore x86 processors, GPUs (via OpenGL), and the Cell processor. There are three basic types you need to be aware of when programming with RapidMind. First, **Values** are small pieces of data, similar to the primitive types such as float and integer in C++. Second, **Arrays** represent arrays of **Values**, just like C arrays or C++ vectors. Third, programs encapsulate computation in the same way that a C++ function does. After identifying the parallel components of an application, the developer replaces numerical types representing floating point numbers and integers with the equivalent RapidMind platform types. While the user's application is running, sequences of numerical operations invoked by the application are profiled, and then dynamically compiled to a program object by the RapidMind platform. The RapidMind platform run-time manages parallel execution of program objects on the target hardware platform. Applications that fit directly into RapidMind's data parallel model can see significant performance gains. However, porting existing C++ applications requires a nontrivial amount of modification in order to fit the RapidMind model and language constructs.

6.5 Annotation Based Languages

Gossamer is an annotation-based approach, so the most closely related work is Cilk++ [18], OpenMP [51], and Unified Parallel C (UPC) [57]. All four define a set of parallel annotations to specify concurrency and synchronization in a program. What sets Gossamer apart is that it can handle a broad range of parallel computations with its relatively small set of annotations, and it does so efficiently.

Cilk++ adds three keywords to the C++ programming language to express parallelism and synchronization: `cilk_spawn` and `cilk_sync` for task and recursive parallelism, and `cilk_for` for loop parallelism. Cilk++ also provides library support for generic mutex locks and C++ templates called *hyperobjects* that allow Cilk strands to coordinate when updating a shared variable or performing a reduction operation. An output stream *hyperobject* can be used to achieve the same behavior as Gossamer’s `buffered` annotation. Cilk++ cannot (at all directly) support all computational dwarfs, because it lacks annotations equivalent to Gossamer’s `divide/replicate`, `barrier`, `copy`, and MapReduce.

OpenMP supports shared-memory programming in C, C++ and Fortran on many platforms and architectures. OpenMP consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP supports task, recursive, iterative, and domain decomposition parallelism. It also supports reductions on scalar variables. OpenMP’s domain decomposition does not do any bookkeeping like Gossamer’s `divide` annotation. It simply replicates code over threads and requires the programmer to manually divide any data used in the computation. OpenMP also lacks annotations equivalent to Gossamer’s `buffered` and MapReduce, and its `ordered` construct is only allowed inside parallel for loops. Timing tests on recursive programs like quicksort and n-queens show that Gossamer is orders of magnitude faster because OpenMP does not automatically prune parallel recursive calls.

Unified Parallel C (UPC) is an extension of the C programming language designed for high performance computing on large-scale parallel machines. In the UPC

Application	Parameters	Execution Time (seconds)				
		Sequential (GCC)	Gossamer	Cilk++	Sequential (ICC)	OpenMP
Quicksort	n = 100,000,000	17.86	5.65	3.83	17.96	19.15
N-Queens	n = 14	9.30	1.31	26.81	15.25	25.91
N-Queens Alt.	n = 14	9.79	2.17	3.02	12.62	5.20
Bzip2	file = 1.0 GB	169.92	26.35	26.96	-	-
Matrix Mult.	n = 4096	198.51	23.74	22.61	100.62	20.10
Matrix Mult.	n = 8192	1590.39	291.91	191.14	877.17	160.33

Table 6.1: Gossamer, Cilk++, and OpenMP: Execution Times

execution model, the amount of parallelism is fixed before the program starts. The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. All thread interaction is explicitly managed by the programmer through locks, barriers, and memory fences provided by the language. UPC has no annotations for forking arbitrary tasks as threads, programming recursive tasks, buffering output, or ordering threads. UPC is appropriate for problems with regular, predictable communication patterns over large sets of data like those often found in scientific computing. For unpredictable computations, this can lead to load imbalances among the threads and result in poor performance.

6.6 Performance of Gossamer versus Cilk++ and OpenMP

This section describes the performance between Gossamer, Cilk++, and OpenMP running on an 8-core Intel Xeon multiprocessor system for a variety applications. Sequential and Gossamer programs were compiled with the GNU C compiler version 4.4.1 using option -O3. Cilk++ programs were compiled with the Intel Cilk++ Software Development Kit version 1.10 (build 8503) using option -O3 and use the Cilk++ *Miser* memory allocator. OpenMP programs were compiled with the Intel C/C++ compiler version 11.1 using option -O3. Tables 6.1 and 6.2 show execution times and speedups, respectively, of quicksort, n-queens, bzip2, and matrix multiplication for each annotation-based approach.

Application	Parameters	Speed Up (relative to sequential)		
		Gossamer	Cilk++	OpenMP
Quicksort	n = 100,000,000	3.16	4.66	0.93
N-Queens	n = 14	7.11	0.35	0.36
N-Queens Alternative	n = 14	4.51	3.24	1.88
Bzip2	file = 1.0 GB	6.45	6.30	-
Matrix Multiplication	n = 4096	8.36	8.78	5.01
Matrix Multiplication	n = 8192	5.45	8.32	5.47

Table 6.2: Gossamer, Cilk++, and OpenMP: Speedups

Cilk has the best performance on Quicksort because its work-stealing scheduler helps cover some of the load imbalance that results from the sequential partition step. Gossamer does reasonably well. OpenMP does very poorly because it does not prune parallel recursive calls and hence creates far too many tasks for this problem.

Gossamer achieves excellent speedups for N-Queens when it is programmed as in Figure 3.2, but Cilk++ and OpenMP do very poorly. Code profiling shows that both Cilk++ and OpenMP spend over 50% of their execution time inside of `malloc()` calls that are needed to make copies of the chess board for each thread.

The Intel Cilk++ implementation provides an alternative N-Queens implementation that uses a single dimension array to store the chess board, which reduces the memory requirements. Cilk++ does much better with this alternative, and OpenMP does somewhat better. Gossamer does somewhat worse with this representation of the chess board, but it still has the best performance.

Both Gossamer and Cilk++ have excellent speedup for Bzip2. Cilk++ uses a *hyperobject* to order the compressed output, and it also buffers the output in a string buffer before flushing the output stream. This adds extra memory operations to every compressed block and accounts for the slowdown compared to Gossamer. OpenMP cannot directly implement the parallel version of Bzip because its ordering primitive can only be used within parallel for loops, and it does not have other synchronization primitives that could be used.

Gossamer and Cilk have superlinear speedup for matrix multiplication with `n` equal to 4096. This is because the available L2 cache doubles when going from 4 to 8 processors. Intel's C compiler generates better sequential code than the GNU C compiler for this application, therefore its overall speedup is less. Gossamer is

```

double **A, **B, **C;
int n;

void mm(int i, int new_size) {
    int j, k;

    for (; i < new_size; i++)
        for (k = 0; k < n; k++)
            for (j = 0; j < n; j++)
                C[i*n + j] += A[i*n + k] * B[k*n + j];
}

void main() {
    int i;
    int stepi = 0, stepd = n / NCPUS, stepn = stepd;
    ...
    for (i = 0; i < NCPUS - 1; i++) {
        fork mm(stepi, stepn);
        stepi += stepd;
        stepn += stepd;
    }
    // leftover chunk in case of uneven division
    fork mm(stepi, nn);
    join;
    ...
}

```

Figure 6.1: Matrix Multiplication Using Only One Filament Per Server Thread

slower than Cilk or OpenMP for the larger matrix because of the overhead of forking 8192 filaments. When matrix multiplication is programmed in Gossamer to use only one thread per server, the speedup is 8.67. Currently, to program a coarse version of matrix multiplication with Gossamer, the programmer has to manually divide up the iteration space in the outer loop. It also requires that the programmer know how many server threads are available at run-time, which can be either done manually or by using the macro `NCPU`. The program used in this benchmark is shown in Figure 6.1.

CHAPTER 7

CONCLUSION AND FUTURE WORK

As multicore architectures move from having tens of processing cores to having hundreds or thousands of processing cores, there will be an increasing need for better concurrent programming tools. Most current parallel programming tools require a lot of effort from the programmer to create threads, synchronize them, and do all the bookkeeping that is needed for correct and efficient parallel programs.

This dissertation presented the Gossamer package, which provides fifteen simple annotations that one uses to parallelize sequential programs and a translator and run-time system that together produce efficient, scalable programs. The annotations allow programmers to focus on program design and algorithms, leaving the bookkeeping of thread management and run-time support to Gossamer. The examples have shown that many common sequential codes can see speedups by adding just a few annotations to a program, and that classic parallel algorithms can be written with fewer lines of code than before.

The key techniques that we have used to achieve efficiency are small stateless threads, efficient synchronization primitives, efficient scheduling, and automatic pruning of `fork/join` computations. The Gossamer package is also portable, because it uses Pthreads to provide server threads and is written in C except for fewer than one hundred lines of assembly code.

There are many open research topics related to Gossamer. For example, heterogeneous hardware is a growing trend in the industry [41, 28]. Section 6.4 describes current GPGPU approaches to parallel programming. By expanding Gossamer to support GPGPUs, programs containing stream-based computations can see greater speedups while still hiding the complex GPGPU programming using its annotations. Additional research can be done on efficiently scheduling and executing threads on heterogeneous processing cores.

As the number of cores increases, non-uniform memory access (NUMA) may become more common. Results in Section 5.3 shown that this would require a more complex memory model. To tune for NUMA, the run-time must know what data will be accessed by a filament and to schedule that filament on a processor that is close to the data. This is a challenging topic due to the complex program control flow, data structures, and modularity that are common in many non-scientific programs.

Dynamic load balancing can reduce run-time overhead. The overhead of Gossamer is low compared to similar programs using operating system level threads, such as Pthreads; however, creating, scheduling, and executing excess filaments can lead to poor program performance. For example, the overhead of `parallel` annotations can be further improved by only creating enough filaments to get all server threads active. This is similar to the pruning technique used for recursive `fork` calls. Initially, the loop iterations are divided among the server threads with one chunk of iterations for each server. Each time a loop iteration finishes, the run-time checks to see if any other server threads are idle. If so, it further divides its remaining iterations with the idle server threads. Special care would have to be taken to make sure enough work is produced to keep all server threads busy; otherwise server threads could go unnecessarily idle, negating any benefit from reducing creation overhead.

A difficult aspect of parallel programming is verification and testing. The non-deterministic behavior of concurrent programs can wreak havoc when trying to verify results. Adding to the Gossamer framework the ability to replay programs deterministically [13, 15, 44], integrating model-checking algorithms [19, 25], and using probabilistic models [9, 58] could all help test and verify Gossamer programs.

Another difficult aspect of concurrent programming is performance tuning. Improving Gossamer’s static and run-time analyses could greatly aid in this solving this problem, and while many static analysis problems are NP complete or harder (some are known to be undecidable) [48, 37], a combination of heuristics and run-time tools may prove fruitful in this area.

Finally, *ease of use* is subjective and difficult to measure. Proper human-based user studies should be done in order to put a quantitative value on how easy it is to

use Gossamer relative to other concurrent programming approaches. For example, students doing an assignment in a parallel programming class could be split up into two groups: one programming using Gossamer and the other using Pthreads. Metrics like the time it took to complete their assignments and the number of lines of code written could be used to determine how much easier it may have been to use Gossamer versus using a traditional threads library.

APPENDIX A

GOSSAMER ANNOTATIONS

This appendix contains a brief synopsis of the Gossamer annotations. The annotations are highlighted in blue and are boldfaced. ANSI-C constructs are in black. Optional constructs are enclosed in < > symbols. For example,

```
< return-var = > function ( arg );
```

indicates that the return variable assignment is optional (provided the function returns a value). Statements using the notation

```
stmt || block-stmt
```

can either be a single valid ANSI-C statement or a series of ANSI-C statements enclosed in braces.

A.1 Task Parallelism

```
< return-var = > fork function ( copy arg[n][...], ... );
```

```
join( < n > ) || join;
```

A.2 Loop Parallelism

```
parallel for ( init loop var; test loop var; inc/dec loop var )  
  stmt || block-stmt
```

A.3 Domain Decomposition

divide arrvar[size<^>][...], ... **replicate**

stmt || block-stmt

divide arrvar[size<^>][...]

where where-stmt, ...

replicate

stmt || block-stmt

barrier;

A.4 Synchronization

atomic < using mutex||spinlock >

stmt || block-stmt

buffered

stmt || block-stmt

buffered (< ordered >, < buffer-size >)

stmt || block-stmt

ordered

stmt || block-stmt

A.5 MapReduce

```
mr_space var ( key-type, val-type );  
mr_list var;  
  
mr_put ( mr_space, key, value );  
mr_getkey ( mr_space, &key, &values );  
mr_getvalue ( mr_space, values, &value );
```

APPENDIX B

GOSSAMER MANUAL

This appendix describes how to install Gossamer, and how to compile, run, debug, and profile Gossamer programs.

B.1 Installing Gossamer

The Gossamer source code can be obtained from <http://www.cs.arizona.edu/people/robackja/Gossamer/>. To install Gossamer, first unpack the source tar file and change directory into the source folder. For example:

```
$ tar jxvf gossamer.tar.bz2
$ cd gossamer
```

Next, type `make` to build the Gossamer binaries, then type `make install` to install the binaries. By default, Gossamer will be installed in `$HOME/local/gossamer`. The install directory can be specified during the `make` process. For example:

```
$ make PREFIX=/opt/gossamer
$ make PREFIX=/opt/gossamer install
```

The same `PREFIX` location must be specified during the build and install phases. Finally, add `$PREFIX/bin` to your environment's `$PATH` variable to easily access the Gossamer translator. For example, using `bash` shell:

```
$ export PATH="/opt/gossamer/bin:${PATH}"
```

B.2 Compiling and Running Gossamer Programs

Gossamer programs are compiled (or translated) into binaries using the Gossamer translator. The translator is invoked using the UNIX command `gsc`. The `gsc`

command assumes that files with the `.gsc` extension are Gossamer programs and acts accordingly.

The Gossamer translator, `gsc`, accepts many of the same arguments as the GNU C Compiler, `gcc`. For example, the source code for the quicksort example from Section 3.1 can be found under the name `qsort.gsc` in the `examples` directory of the Gossamer installation. The `qsort` example can be translated and compiled using level 2 GCC optimizations by typing the following command:

```
$ gsc -O2 qsort.gsc -o qsort
```

The translator produces the `qsort` executable. To run the `qsort` program, type:

```
$ qsort 100000 qsort.in qsort.out
```

This executes `qsort` on all available processors, sorts 100,000 random integers from the file `qsort.in`, and puts the result in `qsort.out`.

The `gsc` translator accepts the following arguments:

- `-c` Stop after producing `.o` files (binary files to be linked).
- `-E0` Stop after producing `.gsi` files (`.gsc` files preprocessed once).
- `-E1` Stop after producing `.gc` files (annotations translated into C code using the run-time library).
- `-E` Stop after producing `.i` files (fully preprocessed files).
- `-help` Print help and quit.
- `-S` Stop after producing `.S` files (assembly files).
- `-save-temps` Save all the intermediate files.
- `-v` The same as `-v2`.
- `-v1` Verbosity level 1: Show what commands are being executed.
- `-v2` Verbosity level 2: Show what commands are being executed.
- `-version` Print version and quit.

The `gsc` translator passes the following arguments directly to `gcc`:

- `-A<argument>` Pass to the preprocessor.
- `-D<argument>` Pass the `-D<argument>` to the preprocessor.
- `-g` Pass to `gcc` (produce symbol table information).
- `-I<directory>` Add the directory to the head of the list of directories to be searched for header files.
- `-include<argument>` Pass to the preprocessor.
- `-l<argument>` Pass to `gcc`, to specify a library to link.
- `-L<argument>` Pass to `gcc`, to specify a directory to look for libraries in.
- `-m*<argument>` machine-specific options, passed to `gcc`. (If you specify this argument and are not using the right kind of machine, you may get unspecified behavior.) The `-m` argument is passed to the preprocessor and to `gcc`.
- `-M` Produce dependency rules (see the option for `-M` to `gcc`).
- `-MM` Produce dependency rules (see the option for `-MM` to `gcc`).
- `-n` Do not actually do any compilation. Useful with `-v`.
- `-o <file>` Specify where to write the output.
- `-O<argument>` Pass to `gcc` (optimize, e.g., `-O2`).
- `-static` Pass to `gcc`, to specify that static linking should be used.
- `-W<argument>` Pass to `gcc`.
- `-Wall` or `-W`] Pass to `gcc`.
- `-Wl,<argument>` Pass to `gcc`, which will pass it to the linker.
- `-Wp,<argument>` Pass to the preprocessor.

Both `gcc` and the `gsc` translator accept the following argument:

`-x <lang>` Specify the input language for the rest of the files. Normally, the compiler determines the input language from the file suffix. For example, files named `*.gsc` are assumed to be Gossamer files. However, you can explicitly specify the language as follows:

<code>-x</code> argument	equivalent file suffix	file type
<code>-x gsc</code>	<code>.gsc</code>	Gossamer
<code>-x gsi</code>	<code>.gsi</code>	Preprocessed by translator
<code>-x gc</code>	<code>.gc</code>	Translated (to be preprocessed again)
<code>-x c</code>	<code>.c</code>	C
<code>-x objective-c</code>	<code>.m</code>	Objective C
<code>-x c++</code>	<code>.C, .cc, .cxx, .cpp</code>	C++
<code>-x c-header</code>	<code>.h</code>	C header
<code>-x cpp-output</code>	<code>.i</code>	Completely preprocessed C
<code>-x c++cpp-output</code>	<code>.ii</code>	Completely preprocessed C++
<code>-x assembler</code>	<code>.s</code>	Assembly
<code>-x assembler-cpp</code>	<code>.S</code>	Assembly (to be preprocessed)
<code>-x none</code>		Revert to normal suffix rules

B.3 Debugging and Profiling Gossamer Programs

Gossamer adds run-time checks into the code generated from annotations when a program is compiled with the option `--gs-runtime-checks`. For example:

```
$ gsc --gs-runtime-checks -O2 runtime-test.gsc -o runtime-test
```

If the program were to violate a run-time annotation check, it would be aborted and the source file and line number would be displayed, as in the following example:

```
Gossamer Abort Trap
'barrier' annotation used outside a divide/replicate filament
File: runtime-failtest.gsc
Line: 56
```

The translator can also instrument programs to profile annotations. Using the command line flag, `--gs-profile-totals`, a summary of the annotation usage in a

program is displayed when a program exits. For example, assume `qsort` is compiled as follows:

```
$ gsc --gs-profile-totals -O2 qsort.gsc -o qsort
```

When `qsort` is executed, profiling data is gathered. It is displayed to the console after the program exits, as shown below:

```
-----
| Gossamer Program Summary - Totals                                     |
|-----|
|                               Total Filaments: 16                     |
|                               Total Forks: 16                         |
|                               Total Joins: 16                         |
|                               Total Parallel Loops: 0                 |
|                               Total Divide/Replicate Blocks: 0        |
|                               Total Barrier Calls: 0                  |
|                               Total Atomic Statements (lock): 8        |
|                               Total Atomic Statements (reduction): 0   |
|                               Total Copied Arguments: 0               |
| Total Average Size of Copied Argument: 0 bytes                       |
|                               Total Size of Copied Arguments: 0 bytes  |
|-----|
```

Individual annotation counts can be gathered via the `--gs-profile-annotation-counting` command line flag. Suppose the `qsort` example above is compiled as follows:

```
$ gsc --gs-profile-annotation-counting -O2 qsort.gsc -o qsort
```

During execution, annotations would be counted. After execution, the following profile would be printed:

Annotation Counts			
Count	Annotation	Source File	Line Number
16	fork	gs_quicksort.gsc	56
16	fork	gs_quicksort.gsc	57
16	join	gs_quicksort.gsc	58

All debugging and profiling options can be used together or individually. Since these options are performed at run-time, they add overhead and thus are recommended only for debugging and testing purposes.

REFERENCES

- [1] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High performance Fortran compilation techniques for parallelizing scientific codes. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–23, Washington, DC, USA, 1998. IEEE Computer Society.
- [2] AMD Corporation. ATI Stream SDK v2.0, 2009.
- [3] Apple Inc. Grand Central Dispatch (GCD) reference, 2009.
- [4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.
- [5] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiawicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.
- [6] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.
- [7] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.
- [8] Siegfried Benkner. VFC: The Vienna Fortran Compiler. *Sci. Program.*, 7(1):67–81, 1999.
- [9] Andrea Bianco and Luca De Alfaro. Model checking of probabilistic and non-deterministic systems. pages 499–513. Springer-Verlag, 1995.
- [10] Aart Bik, Milind Girkar, Paul Grey, and X. Tian. Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems, 2001.

- [11] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.
- [12] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *In Languages and Compilers for Parallel Computing*, pages 141–154. Springer-Verlag, 1994.
- [13] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *In First USENIX Workshop on Hot Topics in Parallelism (HotPar’09)*, March 2009.
- [14] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.
- [15] Jacob Burnim, George Necula, and Koushik Sen. Separating functional and parallel correctness using nondeterministic sequential specifications. In *In Second USENIX Workshop on Hot Topics in Parallelism (HotPar’10)*, June 2010.
- [16] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.
- [17] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, and Calvin Lin. The case for high-level parallel programming in ZPL. *IEEE Comput. Sci. Eng.*, 5(3):76–86, 1998.
- [18] Cilk Arts Inc. Cilk++ technical specifications and data sheet. Data Sheet, Cilk Arts Inc., September 2009.
- [19] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.
- [20] Cray Inc. Chapel language specification, February 2008.
- [21] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured programming*. Academic Press Ltd., London, UK, UK, 1972.
- [22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

- [23] Stuart Dreyfus. Richard bellman on the birth of dynamic programming. *Oper. Res.*, 50(1):48–51, 2002.
- [24] Daniel J. Ellard. S-Q Course Book, July 1997.
- [25] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.
- [26] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.
- [27] Anwar Ghuloum, Eric Sprangle, Jesse Fang, Gansha Wu, and Xin Zhou. Ct: A flexible parallel programming model for tera-scale architectures, October 2007.
- [28] Khronos Group. OpenCL v1.0, 2010.
- [29] Manish Gupta and Prithviraj Banerjee. Paradigm: a compiler for automatic data distribution on multicomputers. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 87–96, New York, NY, USA, 1993. ACM.
- [30] Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.
- [31] Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview, 2006.
- [32] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.
- [33] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
- [34] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. pub-IEEE-STD, pub-IEEE-STD:adr, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.
- [35] Intel Corporation. Intel thread building blocks, 2009.

- [36] Donald E. Knuth. Dynamic huffman coding. *J. Algorithms*, 6(2):163–180, 1985.
- [37] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
- [38] Jim Larson. Erlang for concurrent programming. *Commun. ACM*, 52(3):48–56, 2009.
- [39] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 101–110, New York, NY, USA, 2009. ACM.
- [40] Stephen Lewin. Picturing Parallelism: The Cilk Performance Analyzer, October 2009.
- [41] Owen Liu. Amd technology: power, performance and the future. In *CHINA HPC '07: Proceedings of the 2007 Asian technology information program's (ATIP's) 3rd workshop on High performance computing in China*, pages 89–94, New York, NY, USA, 2007. ACM.
- [42] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Efficient support for fine-grain parallelism on shared-memory machines. *Concurrency: Practice and Experience*, 10(3):157–173, 1998. <http://citeseer.ist.psu.edu/lowenthal99efficient.html>.
- [43] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 787–795, New York, NY, USA, 2004. ACM.
- [44] Michael D. McCool. Structured parallel programming with deterministic patterns. In *In Second USENIX Workshop on Hot Topics in Parallelism (Hot-Par'10)*, June 2010.
- [45] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.
- [46] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.
- [47] G.E. Moore. Progress in digital integrated electronics. *Electron Devices Meeting, 1975 International*, 21:11–13, 1975.

- [48] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 219–230, New York, NY, USA, 1981. ACM.
- [49] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'reilly and Associates, 101 Morris Street, Sebastopol, CA 95472, USA, 1998.
- [50] Nvidia Corporation. Nvidia Compute Unified Device Architecture (CUDA), 2009.
- [51] OpenMP Architecture Review Board. OpenMP specifications, v3.0, May 2008.
- [52] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.
- [53] RapidMind Inc. RapidMind Multicore Development Platform, 2009.
- [54] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 271–271, New York, NY, USA, 2007. ACM.
- [55] Julian Seward. Bzip2, 2008.
- [56] Sun Microsystems Inc. The Fortress language specification, v1.0. Technical report, Sun Microsystems Inc., March 2008.
- [57] UPC Consortium. UPC language specifications, v1.2. Technical report, UPC Consortium, 2005.
- [58] Eli Upfal. A probabilistic relation between desirable and feasible, models of parallel computation. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 258–265, New York, NY, USA, 1984. ACM.
- [59] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.
- [60] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990.

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