Filaments: Efficient Support for Fine-Grain Parallelism

Dawson R. Engler, Gregory R. Andrews, David K. Lowenthal
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

February, 1994

Abstract. It has long been thought that coarse-grain parallelism is much more efficient than fine-grain parallelism due to the overhead of process (thread) creation, context switching, and synchronization. On the other hand, there are several advantages to fine-grain parallelism: architecture independence, ease of programming, ease of use as a target for code generation, and load-balancing potential. This paper describes a portable threads package, Filaments, that supports efficient execution of fine-grain parallel programs on shared-memory multiprocessors. Filaments supports three kinds of threads—run-to-completion, barrier (iterative), and fork/join—which appear to be sufficient for scientific computations. Filaments employs a unique combination of techniques to achieve efficiency: stateless threads, very small thread descriptors, optimized barrier synchronization, scheduling that enhances data locality, and automatic pruning of fork/join threads. The gains in performance are such that on an application such as Jacobi iteration, the execution time for a fine-grain program with a worst-case granularity of a thread per point can be within 10% of that for a coarse-grain program with only one task per processor. Execution times for problems with more work per thread are usually indistinguishable from coarse-grain programs, and they can be faster when the amount of work per thread varies.

1. Introduction

The typical approach to writing a parallel program for a p-processor machine is to divide the application into p tasks and then to execute each task on a distinct processor. For example, to multiply two $n \times n$ matrices one can partition the n^2 inner product computations among p processes by assigning each process a strip or block of the result matrix. Assuming there are more inner products to compute than processors available, the result is a *coarse-grain program*. An alternative approach is to write a *fine-grain program* in which each process (thread) consists of a small, independent unit of work. Continuing the example, for matrix multiplication each inner product can be computed in parallel by a logically distinct process.

There are several advantages to fine-grain programs. First, they are architecture independent in the sense that parallelism is expressed in terms of the application and problem size, not the number of processors that might actually be used to execute the program. This also makes fine-grain programs easier to write, because it is not necessary to cluster independent units of work into a fixed set of larger tasks; indeed, adaptive programs such as divide and conquer algorithms do not have any *a priori* fixed set of tasks. Third, the implicit parallelism in functional or dataflow languages is inherently fine-grain, as is the inner-loop parallelism either extracted by parallelizing compilers or expressed in parallel variants of languages such as Fortran; this simplifies code generation for such languages. Finally, when there are many more processes than

This work was supported by NSF grants CCR-9108412 and CDA-8822652.

processors, it is often easier to balance the total amount of work done by each processor; in a coarse-grain program, it is important that each process be statically assigned about the same amount of work, but this is impossible if the computation is dynamic or if the amount of work per process varies dynamically (e.g., in the calculation of Mandelbrot sets).

Although fine-grain parallelism has attractive attributes as a programming model and a target for code generation, conventional wisdom is that a coarse-grain program will execute much more efficiently than a fine-grain one due to the overhead of process creation, context switching, and synchronization. For example, Lin and Snyder [Lin90] found a fine-grain implementation of Jacobi iteration using the Sequent Symmetry's parallel programming library to be 8 to 23 times slower than a coarse-grain one, depending on the problem size and number of processors. More recent work, such as Chores [Eage93], has been able to get efficient performance by automatically clustering fine-grain tasks into larger units, but executing fine-grain tasks independently is still 10 to 20 times slower, again for Jacobi iteration on a Sequent Symmetry.

This paper describes a threads package, Filaments, that employs a unique combination of techniques to implement fine-grain parallelism directly, efficiently, *and* portably on shared-memory multiprocessors. In particular, Filaments synthesizes and extends previous work such as WorkCrews [Vand88], Chores, the Uniform System [Thom88], and TAM [Cull93]. The most important technique is stateless threads; i.e., threads do not have a private stack. Thread descriptors are also small, so hundreds of thousands of threads can be supported without exhausting memory. Other techniques include control of thread placement for data locality, very efficient barrier synchronization for iterative computations, back-and-forth scheduling to enhance data locality, continuations for multiple barrier programs, pruning and automatic load-balancing for fork/join computations, and an efficient implementation of fork/join synchronization. The gains in performance are such that a fine-grain implementation of Jacobi iteration with a thread per point can be within 10 *percent* of that of a coarse-grain implementation with a process per processor. Execution times for programs with more work per thread are comparable to, and in some cases better than, those of coarse-grain programs.

Filaments contains a small set of primitives that we believe are sufficient to support parallel scientific computations. It is also written entirely in C, without any machine dependent context-switching code. The Filaments package has been used as a runtime library for parallel programs written in C (see Sections 3 and 4) and as the runtime system for a modified compiler for the functional language Sisal [Free93]. The relative effects of the various implementation techniques used by Filaments are described in detail in [Engl94].

The remainder of the paper is organized as follows. The next section describes the Filaments package and how it is used. Section 3 gives performance results for a variety of fine- and coarse-grain programs run on two contemporary multiprocessors: a 4-processor Silicon Graphics Iris and a 14-processor Sequent Symmetry. Section 4 gives an overview of the implementation of Filaments. Section 5 discusses related work. Finally, Section 6 presents conclusions and mentions current work on a distributed implementation of Filaments.

2. An Overview of the Filaments Package

The Filaments package supports three kinds of threads: run to completion, iterative, and fork/join. We have found these to be sufficient to support all parallel applications we have examined, which we believe are representative of most if not all such applications.

A run-to-completion thread executes once and then terminates; it is used in applications such as matrix multiplication. Iterative threads execute repeatedly, with barrier synchronization and termination detection occurring after each execution of all the threads; iterative threads are used in applications such as Jacobi iteration. Sequences of iterative threads are also supported; these are used in applications such as red/black SOR and multigrid in which each iteration has multiple phases, each of which ends with a barrier synchronization point. Fork/join threads recursively fork new threads and later wait for them to return results; they occur in divide-and-conquer applications such as adaptive quadrature.

A fine-grain program is executed by first initializing the Filaments package. Next, the application creates and initializes its shared data structures, and then creates threads. Each thread consists of a pointer to a user-supplied function and a set of user-supplied arguments. Once all (initial) threads have been created, the application calls a Filaments routine that creates one *server* process per processor. (Servers are created using a host system call such as `mfork`.) Each server runs on its own processor; they execute threads until all of them have terminated.

Threads in Filaments cannot be preempted. This is necessary in order to make them stateless, and we have not found it be be a limitation, because fairness is not really needed in a parallel program (all work must be done before the program can complete). If a thread needs to lock a shared data structure, it must spin until it acquires the lock. However, this cannot cause deadlock, because the lock must be held by another thread, which is executing and will not be preempted and hence will eventually release the lock.

This section gives an overview of the primitives provided by the Filaments package and shows how they are used. We do so by considering three simple, but representative applications: matrix multiplication, Jacobi iteration, and adaptive quadrature. Calls to primitives in the Filaments package begin with `f_`. In addition to the primitives illustrated, there are a few additional ones for memory allocation and profiling.

2.1. Run to Completion Threads: Matrix Multiplication

Consider the problem of computing the matrix product of two $n \times n$ matrices `a` and `b`. The natural unit of parallelism in this problem is one inner product, and there are n^2 inner products.* Each inner product can be computed by a thread that executes once and terminates.

A program that uses the Filaments package typically has three parts: declarations of variables that are to be located in shared memory, declarations of functions containing thread code, and a main routine that initializes and controls the computation. For the matrix multiplication problem, the shared variables are the source and result matrices. The thread code computes an inner product. The main routine initializes the Filaments package, creates and initializes the shared matrices, creates the threads, starts the server processes, and finally prints the results when the server processes complete. Pseudo-code follows:

```
shared real a[n][n], b[n][n], c[n][n]

/* compute the inner product of a[i,*] and b[*,j]
real inner_prod(int i, j)
  real sum = 0  /* use local variable for cache hits */
  for k = 1 to n do
    sum = sum + a[i][k]*b[k][j]
  c[i][j] = sum
```

*There is actually even a finer grain of parallelism for this problem: do all multiplies within an inner product in parallel, and then add them in parallel using a combining tree. However, this granularity of parallelism is quite difficult to program, because of all the synchronization steps and the bookkeeping for the combining tree. Vector machines directly support the first step (the multiplies), but they too require multiple machine instructions for the combining tree.

```
end

main()
  int server
  f_initialize(num_servers)     /* initialize Filaments */
  create and initialize the shared matrices
  for i = 0 to n-1 do {
    server = (i*num_servers)/n  /* server for row i */
    for j = 1 to n do
      f_rtc_thread(server, inner_prod, i, j)
  }
  f_parallel(num_servers)  /* returns when servers done */
  print results
end
```

The call of `f_initialize` initializes the Filaments package to use `num_servers` server processes. The call of `f_rtc_thread` creates one thread; the first argument specifies the server that will execute the thread, the second is a pointer to the thread's code, and the other arguments are passed to the thread when it is executed. Above, threads are assigned to servers by "strips;" i.e., each server computes all inner products in a contiguous set of rows of result matrix `c`. This provides data locality and hence good cache performance.* The call of `f_parallel` starts the server processes; the call returns after all threads have been executed.

2.2. Jacobi Iteration: Iterative Threads

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 numerically by 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; the new value for a point is the average of the values of its four neighbors from the previous iteration. The computation terminates when all new values are within some value `EPSILON` of all old values, or when some maximum number of iterations have occurred.

Because Jacobi iteration uses two grids, all new values can be computed in parallel. If the grid has n^2 points, this leads to the fine-grain program shown below. For this application we use iterative threads. In particular, each iteration of the computation first computes new values for all grid points. Then we (sequentially) swap the old and new values and iterate again until convergence occurs.

The grid is stored in two dynamically allocated vectors of vectors, which are pointed to by variables `new` and `old`. The boundaries of the region are stored in the edges of the grids to simplify processing of boundary points.

```
shared real **new, **old  /* created dynamically */
shared real maxdiff = 0.0 /* max difference/iteration */
shared int k = 0          /* iteration count */
```

*Because data locality is inherently algorithm-dependent, the decision about which server is to execute each thread has to be made outside the Filaments package, either by a compiler or as the result of programmer-provided annotations in a high-level language.

Procedure jacobi contains the code executed by each thread:

```
jacobi(int i, j)
  real temp
  new[i][j] = (old[i-1][j] + other neighbors)/4
  temp = absval(new[i][j] - old[i][j])
  if temp > maxdiff then {
    acquire lock
      if temp > maxdiff then maxdiff = temp
    release lock
  }
end
```

After computing the new value of grid point (i,j), jacobi computes the difference between the old and new values of that point. If the difference is larger than the maximum difference seen on this iteration of the entire computation, then global variable maxdiff might need to be updated. Above we compare temp and maxdiff twice, once before acquiring the lock and once while holding it. This speeds up the computation because in practice very few threads will have to acquire and release the lock.

After all grid points are updated, the following procedure is called to check for convergence and to swap grids:

```
sequential_code()
  real **temp
  k++
  if (k > MAXITERS or maxdiff < EPSILON) then return DONE
  temp = old; old = new; new = temp  /* swap grids */
  maxdiff = 0.0
  return NOTDONE
end
```

This code is executed sequentially by server 0 at the end of every update phase, i.e., after every thread reaches the barrier synchronization point.

The main computation again initializes the servers, allocates and initializes shared variables, creates the threads, and starts the servers. Before creating the threads, main calls the primitive f_new_barrier both to indicate that iterative threads will be used and to specify the sequential code for the corresponding barrier.

```
main()
  f_initialize(num_servers)
  create and initialize grids
  f_new_barrier(sequential_code)
  for i = 0 to n-1 do {
    server = (i*num_servers)/n  /* server to use for row i */
    for j = 1 to n
      f_iterative_thread(server, jacobi, i, j)
  }
  f_parallel(num_servers)
end
```

Threads are again assigned to servers by strips of rows to get good data locality.

Iterative threads are created just once, but they are executed multiple times. In particular, each server executes all threads assigned to it. When all servers are done (i.e., they reach a barrier), server 0 executes the sequential code. If it returns `DONE`, all servers terminate; otherwise they execute their threads again.

The Filaments package also supports applications that have multiple barrier synchronization points in each iteration of the main computation, e.g., red/black SOR and multigrid. (This is the reason for the call to `f_new_barrier` above.) We also provide continuations to allow the threads implementing one phase to pass locally-computed values to the threads implementing the next phase.

2.3. Adaptive Quadrature: Fork/Join Threads

Adaptive quadrature is an algorithm to compute the area under a curve defined by a continuous function `f(...)`. It works by dividing an interval in half, approximating the areas of both halves, and then subdividing further if the approximation is not good enough. The easiest way to program this is to use a divide-and-conquer algorithm. Because subintervals are independent, a new thread can be created to compute each subinterval. Hence this application uses fork/join threads.

The computational routine for adaptive quadrature is:

```
quad(real a, b, fa, fb, area)
  real *left, *right, fm, m, aleft, aright
  compute midpoint m and areas aleft and aright
  if (close enough)
    then return (aleft+aright)
    else {
      /* recurse, forking two new threads */
      left = f_fork(quad, a, m, fa, fm, aleft)
      right = f_fork(quad, m, b, fm, fb, aright)
      /* wait for children to terminate */
      f_join()
      return (*left+*right)
    }
end
```

The algorithm evaluates `f()` just once at each point and evaluates the area of each subinterval just once. Function values and areas are passed to new threads.

The main routine is:

```
main()
  real left, right, *answer, fleft, fright, init_area
  f_initialize(num_servers, MAXTHREADS)
  input left and right
  compute fleft, fright, and init_area
  answer = f_fork(quad,left,right,fleft,fright,init_area)
  f_parallel(num_servers)
end
```

The call of `f_parallel` terminates when all threads have terminated; this serves as an implicit call of join in the main routine.

There is an option in Filaments to stop forking new threads when a server already has enough work.* In particular, the argument `MAXTHREADS` in the call of `f_initialize` above specifies a limit on how many threads can be active at a time. When this pruning threshold is reached, a server turns a fork (thread creation) into a recursive call, which is more efficient. The value of the pruning threshold could be hidden within the Filaments package, but it is not at present to facilitate experimentation with different values.

3. Applications and Performance

The Filaments package has been tested on 10 applications: matrix multiplication, Jacobi iteration, adaptive quadrature, adjoint convolution, Mandelbrot set calculation, fast Fourier transform (FFT), Gaussian elimination, multigrid, Fibonacci numbers, and quicksort. These cover a range of applications, programming styles, and computation/communication ratios.

We used two machines in testing: a 14 processor Sequent Symmetry 81, which has a 16 MHz clock and a 64Kbyte unified instruction and data cache; and a Silicon Graphics Iris 4D/340 multiprocessor, which has a 33 MHz clock, 64 Kbyte instruction cache, 64Kbyte data cache, and 256Kbyte secondary data cache. We used the vendor supplied `cc` compiler on all tests, with the highest possible level of optimization.

For each application, we developed a sequential program, a coarse-grain program, and a fine-grain (Filaments) program. We tried hard to ensure the tests were accurate and fair. First, the programs were implemented as similarly as possible: all declared the same variables in registers and used the same memory allocator and timing facilities. Second, all tests were run in single-user mode. Third, the sequential programs were written without any parallel constructs, and the coarse-grain programs were written using vendor-supplied subroutine libraries. Only the Filaments programs incurred overhead due to the Filaments package. (Filaments itself uses vendor-supplied subroutine libraries to implement its servers.)

The next three subsections give details on the performance of matrix multiplication, Jacobi iteration, and adaptive quadrature. Section 3.4 summarizes the performance of the other seven applications (for complete results, see [Lowe93]).

The performance tables give execution times and speedups. The execution times are the averages of three test runs, as reported by `getrusage`, rounded to the nearest hundredth of a second. These times were normally very consistent, although occasionally the tests were run again when anomalies occurred. Server and filament Creation time is included, but system time is not (it was typically negligible). The reported speedups are the ratio of the *sequential program* times to the other times, rounded to the nearest hundredth. In choosing problem sizes, we strove to make each sequential program take about a minute. This allows the potential for good speedup, and it enabled us to run a large number of tests in a reasonable amount of “wall clock” time.

3.1. Matrix Multiplication

The following tables give the performance of multiplying two 150×150 matrices on the Sequent and two 350×350 matrices on the Iris. Each process in the coarse-grain program computes inner products for a strip of the result matrix. Each filament in the fine-grain program is executed by

*The importance of pruning is discussed at the end of Section 4.5.

the corresponding processor, as shown by the pseudo-code in Section 2.1.

Sequent: matrix size 150 Sequential time: 52.28 sec.

CPUs	Filament Time	F Speedup	CG Time	CG Speedup
1	53.34	0.98	53.25	0.98
2	26.93	1.94	26.75	1.95
4	13.86	3.77	13.57	3.85
8	7.18	7.28	6.80	7.69
12	5.06	10.33	4.66	11.22

Iris: matrix size 350 Sequential time: 58.90 sec.

CPUs	Filament Time	F Speedup	CG Time	CG Speedup
1	59.77	0.99	58.73	1.00
2	29.88	1.97	29.87	1.97
4	15.50	3.80	15.24	3.86

The Filaments program is only slightly slower than the coarse-grain program, because there is a reasonable amount of work per thread (n multiplications and n–1 additions), and this amortizes the Filaments overhead. The work per thread also increases with the problem size.

3.2. Jacobi Iteration

For Jacobi iteration, the Filaments program uses one thread per point and the coarse-grain program uses one thread per strip of the result matrix. The performance of Jacobi iteration on the Sequent, with a 150 by 150 matrix, was as follows:

Sequent: Jacobi size 150 Sequential time: 61.34 sec.

CPUs	Filament Time	F Speedup	CG Time	CG Speedup
1	69.00	0.89	62.30	0.98
2	34.42	1.78	31.17	1.97
4	17.36	3.53	15.80	3.88
8	8.65	7.09	7.90	7.76
12	5.95	10.30	5.40	11.35

The Filaments program is consistently 10-11% slower than the coarse-grain one. This is due to Filaments overhead and is a consequence of there being very little work per thread. Also, the work per thread in this application is constant, independent of problem size.

The performance of Jacobi iteration on the Irs, with a 300 by 300 matrix, was:

Iris: Jacobi size 300 Sequential time: 58.09 sec.

CPUs	Filament Time	F Speedup	CG Time	CG Speedup
1	81.14	0.72	58.30	1.00
2	39.95	1.45	29.10	2.00
4	19.75	2.94	14.52	4.00

On the Iris, the Filaments programs are close to 40% slower than the coarse-grain ones. About 10% of this is due to Filaments overhead. The other 30% is due to optimizations that the C compiler on the Iris can perform for the coarse-grain program that it cannot perform for the fine-grain program. To see why, consider the important part of the coarse-grain program:

```
for i = 1 to n do
  for j = 1 to n do {
    grid[new][i][j] = (grid[old][i+1][j]+...)/4.0
    ...
  }
```

Compare this to the corresponding part of the fine-grain program:

```
jacobi(i,j)
  grid[new][i][j] = (grid[old][i+1][j]+...)/4.0
  ...
end jacobi
```

In the coarse-grain program, a compiler can eliminate common subexpressions—such as `grid[new][i]`—over several iterations of the inner loop. Also, constants such as `4.0` can be placed in a register. However, neither of these optimizations can be used in the fine-grain program, because the compiler has no idea in what order the threads will execute. In fact, the generated code for the body of `jacobi` above is three times longer than the corresponding code for the coarse-grain program.

This is a problem with fine-grain programs in general, but it is especially pronounced in Jacobi, because the thread code does not contain a loop. (If the thread contained a loop, then common subexpressions could be eliminated from it.) The Filaments program on the Sequent performed nearly as well as the coarse-grain program, because the Sequent has very few registers, and hence the C compiler was not able to optimize the coarse-grain program to nearly the same extent as on the Iris.

To check this assumption about the differences in times above being due to the differences in compiler optimizations, we manually rewrote the Filaments program on the Iris to eliminate common subexpressions. The resulting program was around 10% slower than the coarse-grain one. (Of course, no user would ever want to eliminate common subexpressions manually. It just shows that thread packages are not inherently a lot less efficient when executing fine-grain programs, as has been previously claimed, e.g., in [Lin90, Eage93].)

For fine-grain programs in which threads are statically assigned to servers, a good compiler might be able to eliminate common subexpressions. In any event, constants can always be saved across thread executions. Also, because at least some loops are eliminated in fine-grain programs, more registers are freed up, which a smart compiler can take advantage of; in fact, for matrix multiplication on the Sequent, we had to annotate heavily used variables with "register" declarations to get the coarse-grain program to run faster than the fine-grain program! In general, however, coarse-grain programs are amenable to more compiler optimizations.

3.3. Adaptive Quadrature

For recursive applications the natural parallel program is inherently fine grain, with sequentially ordered recursive calls being replaced by concurrently executed recursive calls using fork/join threads. The amount of work per thread becomes less and less as the depth of recursion increases.

For this application, we programmed two coarse-grain algorithms. In both, each server keeps its part of the total area in a privately indexed global vector; when the computation terminates, the final result is obtained by having one server sum the contents of this vector.

The first coarse-grain program statically divides the interval into equal size segments, one per server (processor). Each server process then executes a sequential adaptive quadrature algorithm on its segment.

The second coarse-grain program uses the dynamic “bag of tasks” paradigm [Carr86, Andr91]. In particular, there is one central bag of tasks, and each task in the bag specifies one subinterval. Initially, the bag contains the entire interval over which to integrate. Each server repeatedly gets a task from the bag, uses one level of the adaptive quadrature algorithm to approximate the area under an interval, and either accepts the result or keeps one subtask and puts the other back into the bag. The “bag of tasks” program also does pruning when there is a sufficient number of tasks already in the bag (i.e., a server uses sequential recursion on the task it keeps).

Adaptive quadrature can lead to load imbalance if the processors (servers) receive unequal amounts of work, which could happen if certain areas of the curve are rough and others are smooth. In order to stress the various programs, we used the function `f(x)=exp(x)*sin(x)`. This function requires more work at the right part of the interval because the oscillations are much sharper there.

The following tables give the results for adaptive quadrature on the Sequent and Iris, using intervals on the `x` axis of 1 to 27 and 1 to 35, respectively. The sequential programs use recursion but no parallelism.

Sequent: quad size 27 Sequential time: 56.86 sec.

CPUs	Fil. Time	F Speedup	Static Time	St Speedup	Bag Time	Bag Speedup
1	56.91	1.00	57.40	0.99	56.77	1.00
2	28.52	1.99	56.61	1.00	28.40	2.00
4	14.41	3.95	50.69	1.12	16.01	3.55
8	7.20	7.90	39.26	1.45	15.86	3.59
12	4.74	11.99	25.52	2.23	14.92	3.73

Iris: quad size 35 Sequential time: 54.19

CPUs	Fil. Time	F Speedup	Static Time	St Speedup	Bag Time	Bag Speedup
1	54.68	0.99	55.41	0.98	54.42	1.00
2	27.29	1.99	54.45	1.00	27.39	1.98
4	13.81	3.92	50.97	1.06	14.52	3.73

The Filaments program got near perfect speedup. This is due to the efficient fork/join mechanism in the Filaments package, which balances load when necessary and prunes computations when there is already enough work present. For larger numbers of processors, the pruning threshold had to be tuned to obtain good performance. With problems such as Fibonacci (see next section), where each thread does very little work, pruning is vital to get reasonable performance.

The static coarse-grain program gets almost no speedup. This is because the server working on the right-most interval has to do much more work than the server working on the left-most interval; hence, execution time is dominated by the worst case in the absence of load balancing.

The bag-of-tasks program gets reasonable speedup for up to four servers, but after that there is almost no improvement in performance. This is because access to the bag becomes a bottleneck—even with pruning.

For comparison, we also programmed a third coarse-grain program that divides the interval into a large number of small, equal-sized segments and apportions the segments among the servers. Each server then computes a simple trapezoidal approximation of the area of each of its interval and adds the results. To get the same degree of accuracy of the final result, we had to use an extremely large number of intervals. This made the third program even slower than the other two coarse-grain programs.

3.4. Other Applications

We also tested seven other applications: convolution, Mandelbrot set calculation, fast Fourier transform (FFT), Gaussian elimination, multigrid, Fibonacci numbers, and quicksort. The results are summarized below; they are discussed in detail in [Lowe93].

Convolution is a method frequently used in engineering and related fields to solve problems such as polynomial multiplication [Baas88]. The Mandelbrot set is, of course, a special set of complex numbers that occurs in fractals [Dewd85]. Both adjoint convolution and Mandelbrot can result in load-imbalanced programs, and neither has data locality. The Filaments programs for both create one run-to-completion thread per point. These threads do a variable amount of work, so to balance load, we simply schedule each thread on a random server. Since there is a lot of work to be done, random scheduling approximates a load-balanced distribution of work. The corresponding coarse-grain programs work on a fixed partition of the set of points, which is determined at the beginning of the program. On the other hand, the coarse-grain programs do not incur overhead from running multiple threads. The results are essentially a tradeoff, with the fine-grain programs running from 4% faster to 2% slower than the coarse-grain programs. Speedup was excellent on convolution and almost perfect on Mandelbrot.

A Fourier transform is a used to characterize a function using just sines and cosines; FFT is a fast way to compute a Fourier transform. We ran a two-dimensional radix FFT, which is often used in image processing. A two-dimensional FFT is computed by first computing a sequential FFT on each row and then computing a sequential FFT on each column. Our program uses run-to-completion threads, first creating and executing one thread for each row and then creating and executing one thread for each column. In the coarse grain program, we simply divided the work into equal-size parts. The coarse- and fine-grain programs again had nearly identical performance, with the fine-grain program normally just a few tenths of a percent slower but on occasion faster. Speedup was also nearly perfect for both programs.

Gaussian elimination is a classic method for solving the linear system $Ax = b$. For an $n \times n$ matrix, n iterations are performed, one per column. Each iteration has two phases: one to compute a pivot value, and the other to do elimination. Our Filament program uses multiple barrier threads, with one pair of threads per column. The first thread in a pair computes the pivot; the second implements the elimination phase. Gaussian elimination exhibits load imbalance, because after a column is pivoted, it is never accessed again. Thus, the Filaments program does static load balancing by assigning the `i`th column to processor `i mod p`. The coarse-grain program also uses a cyclic assignment for load balancing, with each processor being assigned a subset of the columns. However, this mildly complicates the coarse-grain program, because processors are not necessarily assigned the same number of columns. It also slows down the coarse-grain program when processors have different numbers of columns, because an extra barrier is then required for the processors having an extra column. Consequently, the fine-grain program was from 2-6% faster than the coarse-grain one on the Sequent and from 6-53% faster on the Iris. The percentage improvement increased as the number of processors increased. On the other hand, the speedup for this application was somewhat less than for most of the other applications, and it got worse as the number of processors increased. This is due to the decreasing amount of work as the computation proceeds.

Multigrid is a popular method for approximating solutions to partial differential equations [Pres91]. Multigrid methods are based on what is known as a coarse grid correction, which works as follows. First, a relaxation technique is used for a few iterations, then the results are restricted to a coarser grid, which has 1/4 the number of points of the original grid. Third, the problem is solved on the coarser grid, using either a relaxation or a direct method. Fourth, the solution on the coarse grid is interpolated back to the finer grid. Finally, a relaxation method is applied to the finer grid. The full multigrid method uses a series of coarse-grid corrections, in a V or W shaped pattern between finer and coarser grids.

Our Filaments implementation of multigrid uses both run-to-completion and barrier threads. We separately parallelized each of the relaxation, restriction, and interpolation phases. In each phase, filaments were assigned to processors by strips of rows. The interpolation phase required the use of multiple barriers together with continuations to pass local values across barrier points. The coarse-grid program also parallelized the relaxation, restriction, and interpolation phases; the code for the first two phases was similar to that for Jacobi iteration, and the code for the interpolation phase was similar to that for matrix multiplication. The Filaments program on the Sequent was from 9% faster to 14% slower than the coarse-grain program, with the relative performance of the Filaments program decreasing as the number of processors increased. The Filaments program was consistently about 5% slower on the Iris. Speedup was decent but not great for both programs.

The other two applications were Fibonacci and quicksort. Both are fork/join applications. However, they are quite different from each other—and from adaptive quadrature—in the amount of work that is done before a parallel recursive call. Fibonacci does virtually no work, whereas quicksort executes two loops. On the other hand, the Filaments program for Fibonacci gets near perfect speedup, whereas the Filaments program for quicksort does not. The excellent performance on Fibonacci is due to the pruning optimization in Filaments; The poorer performance on quicksort is due to the lengthy, sequential partitioning portion in each thread; moreover, this sequential part grows with the problem size. For both applications, the coarse-grain programs use the bag-of-tasks paradigm. The execution time for the Filaments program was from 5% faster to 18% slower on the Sequent; the Filaments and coarse-grain programs had virtually identical performance on the Iris.

4. Implementation

As mentioned, Filaments supports three types of threads: run-to-completion (RTC), iterative (barrier), and fork/join. Filament threads do not have private stacks, state information, or a pointer to the "next" thread (as discussed below). Each server process has one local queue for each type of thread. All thread synchronization must be done through the Filaments package and is limited to barriers and join. As also mentioned, threads cannot be preempted. This helps avoid needing private stacks and machine-dependent context-switching code, but it is not a limitation in parallel applications.

The Filaments package is highly portable; it is written entirely in C, and needs only some means of creating servers, allocating shared memory, and locking. Servers can be created using a system call such as `mfork()`, and shared memory can be allocated using a system call such as `mmap()`. Locks can be provided by the underlying system, or they can be written using one of the many software algorithms. The generic thread specification is only 16 bytes, thread creation costs on average 12 instructions, and the critical path between execution of threads is on average 6 instructions. (Both memory consumption and thread overhead increase for fork/join and variable argument threads.)

This section describes thread descriptors, the internal queue structures, details particular to each type of thread, and the effects of our desire for portability.

4.1. Thread Descriptors

A thread is specified by a pointer to the user-supplied code the thread is to execute and an argument vector of either fixed or variable length. A thread descriptor contains these values; the descriptor for a fork/join thread also has a parent pointer, child counter, and pointer to a lock used for load balancing.

A *basic* thread has three arguments. The first two are of type unsigned long; the last is a void pointer. These arguments typically consist of two array indices and a pointer to shared data. Three arguments are sufficient for most applications (e.g., Weicheck found that, on average, procedures require 2.1 parameters [Weic84]). However, we also support *variable-argument* threads. When such a thread is created, the user provides a format string specifying the argument types as well as the arguments themselves. Supported argument types are integer, pointer, and double. Both basic and variable-argument threads may be intermingled in the same queue, and no explicit checks must be done to determine which type of thread is running.

4.2. Queue Structures

Local ready queues are used for each of the three types of threads (RTC, iterative, and fork/join). This both reduces contention [Ande88] and allows scheduling to be locality driven [Mark92]. Three different queues are used, because each is managed somewhat differently, as discussed in Sections 4.3-4.5. This also allows using multiple types of threads in an application.

Access to the RTC and iterative queues is non-locking in order to avoid locking overhead. It is up to an application program to achieve data locality and load balancing by placing threads on an appropriate queue when they are created, as described in Section 3. Access to fork/join queues is locking, because fork/join applications are inherently dynamic, and hence the load on each server has to be balanced dynamically. In RTC and iterative applications, a server terminates when it exhausts its supply of work. In fork/join applications, when a server's queue is empty, it scans other queues and "steals" threads from the server having the most work; the application terminates when all queues are empty.

The queues for RTC and iterative threads are stored in arrays rather than linked lists. This allows enqueue and dequeue operations to be implemented by simply incrementing or decrementing a pointer. It also saves space, because a thread descriptor does not need to contain a link to the next thread.

4.3. RTC Threads

In traditional thread packages, stack manipulation, memory allocation, and thread setup consume a significant portion of machine resources and comprise the bulk of thread overhead. We do not need stacks for RTC threads, because they are stateless almost by definition: they do not need any previous local history, and they terminate when they are done. All they need is a shared memory from which to get global values and into which to put results. Consequently, all we need to represent and activate an RTC thread is a pointer to code and an argument vector. To create an

RTC thread, we simply allocate a new descriptor from the queue of the intended server (the `where` parameter) and fill in the user code and argument fields:

```
new_thread(where, code, arg1, arg2, arg3) {
  filament *f = new_filament(where);
  f->code = code;
  f->arg1 = arg1; f->arg2 = arg2; f->arg3 = arg3;
}
```

To execute an RTC thread, we simply make a function call; this uses the calling server's stack.

4.4. Iterative Threads

Iterative threads are a generalization of RTC threads: they are executed more than one time, with barrier synchronization between every execution phase. A barrier is described by a queue of iterative threads and user-supplied sequential code, which is run by a single server between iterations. Multiple barriers may be queued one after another; they are executed in circular order until the user code signals termination. To create a barrier, the user first supplies sequential code and then creates some number of threads.

Iterative threads are executed as follows:

```
run_iterative() {
   do {
      while(iterative queue not empty) {
         filament *f = get_thread(iterative queue)
         run_thread(f)
         }
      synchronize()
      if( pid == 0) {
         terminate = barrier.sequential_code()
         /* get next iterative queue */
         iterative = iterative.next;
      }
      synchronize()
   } while(terminate == false)
}
```

The call to `get_thread` above returns a pointer to the next thread to execute. How this is implemented can have a major impact on how well caching performs. The obvious manner is to execute iterative threads in sequence, i.e., from the beginning to the end of the queue. However, if there are a large number of threads, by the time we have reached the end of the queue, the initial thread descriptors and data have been flushed. An alternative is to execute threads in a back-and-forth order, from front to rear, then rear to front, and so on. On the Iris, this gives as much as a 10% improvement; it is also easy to implement when queues are stored in arrays.

Each iterative thread is stateless. We provide continuations and multiple queues of iterative threads to support applications that require multiple barrier synchronization points with "state" passed across the barriers. Each thread in the first queue is executed once, then each thread in the second queue is executed once, and so on up to the last queue; the process is then repeated. User-supplied sequential code, if any, is executed at each barrier point, e.g., to check for termination. Data is passed between threads by means of global variables or arguments.

4.5. Fork/Join

Fork and join are used to create threads dynamically and later wait for them to terminate and return results. Implementing fork/join typically requires that a thread be capable of suspension (sleep) and subsequent wake-up. In short, threads must be able to save state. The overhead of sleeping in traditional thread packages involves a minimum of two context-switches and possibly the manipulation of queues. Another major cost is memory: Because state must be saved on the thread stack, every thread must have a stack; because stack growth is usually difficult to predict, each of these stacks may be quite large. If a number of threads are active at a time, this adversely impacts cache performance and could even lead to virtual memory swapping.

Threads in Filaments are stateless, so the problem to solve is how to save state in fork/join applications without using separate stacks for each thread. The solution is to use recursion.

A simple example should make this clear. The semantics of a join operation are that a parent thread waits for all of its children to complete, either by sleeping (as in traditional thread packages) or by having a signaling mechanism in place that indicates when all of its children have completed. The signaling mechanism we use is a child counter that is incremented on child creation and decremented on child completion. When this counter is zero, the parent is guaranteed that all of its children have completed. Thus, to implement join without actually suspending the parent, we need only have the parent spin in a loop, waiting for its child counter to become zero. But while waiting, the parent can execute other threads! In other words, the parent *becomes* the server.

```
void join() {
   while( my_child_counter > 0 )
      get another thread and call it
}
```

This exchanges function calls for context-switches and recursion for large amounts of wasted stack space. It works correctly because parent/child relationships form a tree; i.e., children are always created after their parents and terminate before their parents.

Forking a thread consists of creating and initializing a fork/join descriptor, assigning user code and arguments, setting a pointer to the parent thread, and atomically incrementing the parent's child counter. A parent joins children as shown above by waiting for its child counter to become zero and then resuming execution. A child signals its parent by atomically decrementing the parent's child counter and then exiting.

An important point to note is that on machines where locks are expensive, incrementing and decrementing the parent's counter can actually be the main source of thread overhead. A number of simple optimizations can be done so that only children that are executed by remote servers need to decrement the parent's counter atomically. In the case of programs like Fibonacci, this can give a factor of 2 or more improvement in performance. The specifics of this are discussed in [Engl94].

Fork/join programs tend to employ a divide-and-conquer strategy. Consequently, initial threads generate larger tasks, which get broken down into smaller and smaller ones [Vand88]. Fork/join programs also need to do load balancing. In Filaments, newly forked threads are put on the tail of a server's local queue, a server removes work from the front of its local queue, and other servers "steal" from the front of another server's queue. This works well because the largest units of work tend to be at the front of server queues.

A very effective optimization is pruning [Vand88]. The concept is simple: If we have enough work, there is no need to create more. The implementation is equally simple: Check to see if the number of threads on the server queue is greater than some threshold; if it is call the user code directly, otherwise create a thread in the standard manner. Naively this takes the form:

```
fork(code, arg1, arg2, arg3) {
   if ( my_ runnable_threads > pruning threshold )
       code(arg1,arg2,arg3);  /* call code directly */
   else
       create and enqueue thread
}
```

The above code has a conditional check, which adds some overhead to all forks. In addition, for variable-argument threads, `fork` has to allocate an argument block and interpret the arguments in order to call the user's thread code. To make fork/join much more efficient, we can automatically duplicate user code; in the copy, all fork calls are replaced by recursive subroutine calls. Now the first time we decide to prune, we can simply call the copy of the original procedure. This removes all Filaments overhead from subsequent "forks" in the user code, because they are now simply recursive calls. This is a very important optimization when the either the amount of work is extremely small (as in Fibonacci), or when variable-argument threads are needed. For programs like quicksort, however, this optimization does not have much overall effect, because there is a reasonable amount of work per fork or call.

4.6. The Effects of Portability

Our desire for portability affects simplicity, efficiency, and expressiveness. First, by supporting variable argument threads, we have to use a preprocessor to avoid an explosion in code size. In particular, procedure calls in C cannot be constructed "on the fly," so we need to preprocess the application code to determine what kind of arguments are being used and then to generate the appropriate procedure declarations and calls. (The alternative would be to generate code for all possible combinations, and then to select the right one at run time.)

Second, the stack of one server process may not be accessible to another. For example, on the Sequent only global data can be shared after `mfork` is called; a machine like the Convex does not even provide any sort of shared memory fork. What this amounts to is that threads cannot communicate using variables on server stacks, but must do so using explicitly allocated shared memory. This difficulty arises in fork/join problems when threads return results. The difference in performance in Fibonacci between returning results using server stacks and using explicitly allocated shared memory is a factor of two.

Third, the Filaments package cannot support any application that requires that a thread be able to block, e.g., a multithreaded event handler or a file server.

An easy way to get around the last two problems is simply to install Filaments on top of a widely supported thread package (e.g., uSystem [Buhr88]) and to use its threads as the Filaments servers. Unfortunately, the only ways we see to solve the code size problem are to revert back to contexts or to resort to runtime code generation.

5. Related Work

There is a wealth of related research on threads packages, some of which support fine-grain parallelism. The first step towards efficient parallelism was lightweight thread packages such as Threads [Doep87], Presto [Bers88], μSystem [Buhr88], C Threads [Coop90], μC++ [Buhr92], and Sun Lightweight Processes [Stei92]. We will call these standard packages to distinguish them from threads packages that do not have a stack for each thread. The goal of standard packages is to provide the user with a natural thread abstraction, and many of the usual concurrent programming primitives; different packages provide different primitives. All of the above packages create a stack for each thread and support preemption to provide fairness. However, having stacks requires context switching code, which is inherently machine dependent. Most of the above packages also use a central ready queue.

Standard threads packages, while providing the user with programming convenience, make it difficult to support fine-grain parallelism efficiently. Consider, for example, a Jacobi iteration program written with a thread per point. In a standard threads package, each thread is context-switched in, performs a handful of instructions, and then context switches into another thread. With large matrices, large numbers of threads are created. Because the work a thread does is very small, the context switching time will constitute a very large overhead; supporting preemption also adds context switching overhead. In addition, the presence of stacks can waste valuable cache space, which could otherwise be used to hold actual data. The use of central ready queues also hurts locality and hence efficient cache usage. Finally, the generality of standard packages increases their size and overhead. Filaments is efficient because it has only one stack per server (processor), does not context switch, does not preempt threads, uses local ready queues, and provides the minimal set of primitives needed for parallel applications.

Several researchers have proposed ways to reduce the inefficiencies of standard thread packages. Anderson et al. [Ande88] discusses the gain from using local ready queues, and [Ande91] shows how to do user-level scheduling [Ande91]. Schoenberg and Hummel [Humm91] explain how to avoid allocating a stack per thread and context switching in nested parallel `for` loops, which are a form of run-to-completion threads. Markatos et al. [Mark92] presents a thorough study of the tradeoffs between load balancing and locality in shared memory machines. Keppel [Kepp93] describes a portable threads package that supports efficient barrier synchronization and non-preemptive threads.

Threads packages that support finer-grain parallelism include the Uniform System, WorkCrews, TAM, and Chores. The Uniform System [Thom88], built for the BBN Butterfly, has several things in common with Filaments: There are no private stacks per thread, no context switches, and threads are not preemptable. The Uniform System's synchronous mode supports a simple form of barrier threads, and their finalization code is equivalent to our sequential code. However, barrier threads in Filaments are more powerful, and they need be created only once. The Uniform System does not support fork/join threads. Another difference is that the Uniform System uses a central ready queue. In contrast, Filaments uses local ready queues; this minimizes lock overhead and enhances data locality. The Uniform System also employs task generators (a related collection of tasks)—and hence has essentially a coarse-grain programming model—whereas Filaments directly supports a fine-grain model. Because of these differences, the Uniform System cannot efficiently support thread-per-point decompositions for problems like Jacobi, as Lin [Lin90] and others have noted.

WorkCrews [Vand88] supports fork/join parallelism on small-scale, shared-memory multiprocessors. The package is implemented in Modula2+. WorkCrews introduced the concepts of pruning and of ordering queues to favor larger threads. Filaments has borrowed these ideas in its implementation of fork/join threads.

TAM [Cull93] is a compiler-controlled threaded abstract machine. It evolved from graph-based execution models for dataflow languages and provides a bridge between such models and the control flow models typically employed by standard multiprocessors. TAM is oriented more for distributed- than shared-memory machines, because threads send messages, which enable other threads, possibly on remote machines. Threads execute from beginning to end without blocking, but they may be preempted by message handlers (inlets in TAM). Given its dataflow heritage, TAM is oriented toward fork/join parallelism. This can be used to support iterative parallelism by turning loop bodies into **cobegin** statements, but at the expense of forking and joining threads on each iteration of the outer loop. In contrast, Filaments directly supports this kind of parallelism by means of barrier threads. In any event, the essential differences between TAM and Filaments are their different heritages (dataflow versus imperative) and consequently their different means for specifying and implementing fine-grain parallelism. TAM defines an abstract machine of self-scheduling parallel threads, which is used as an intermediate language that is mapped to existing processors, whereas Filaments defines a portable systems-call library, which is used to specify parallelism in a traditional, imperative way.

Chores [Eage93] is also similar both to Filaments and to the Uniform System. Chores runs on top of Presto on a Sequent Symmetry. It uses a central ready queue, but servers take jobs in chunks. This amortizes the lock overhead of the central ready queue. Like Filaments, Chores has no private stacks per thread, no context switches, and no preemption. Chores is more flexible than Filaments because user threads are run on top of system threads that have a stack, which permits blocking if necessary. However, Filaments directly supports efficient fine-grain parallelism, whereas Chores requires preprocessor support and the use of task generators in order to cluster fine-grain tasks into coarse-grain units. For example, on Jacobi iteration, Chores is nearly 20 *times* slower on a Sequent with a thread per point than a thread per row.

One attribute of Chores is the ability to combine functional and data parallelism in a single application. An example in the Chores paper is a binary tree for computing matrix expressions; each leaf node is a matrix and the interior nodes are matrix multiplies. In Chores, functional parallelism is used to traverse the expression tree, and data parallelism is used to compute the matrix products. This application can be programmed very simply in Filaments using just fork/join filaments; no locking or other kind of explicit synchronization is required. The Chores paper describes two tests using a 128 leaf tree with 25 by 25 matrices and an 8 leaf tree with 100 by 100 matrices. On a 16-processor Sequent, they got sequential and parallel times of 28.2 and 2.2 seconds for the first tree and 99.1 and 6.5 seconds for the second test. We ran similar tests using Filaments on an 8-processor Sequent, and got sequential and parallel times of 13.3 and 2.1 seconds for the first tree and 42.1 and 7.6 seconds for the second test. The speedups with Filaments were not as great as with Chores, but the time per processor was *much* less. Moreover, the application was very easy to program.

6. Conclusion

In this paper we have shown that fine-grain parallelism can be implemented directly, efficiently, and portably on shared-memory multiprocessors. The key techniques that we have used to achieve efficiency are small, stateless threads, control of data locality, efficient barrier synchronization, efficient scheduling, and automatic pruning of fork/join computations. The Filaments package supports three kinds of threads, which are sufficient to implement every parallel computation we have examined so far. The package is also written entirely in C, so it is portable.

The results reported here apply to shared-memory multiprocessors. We are working on an extension of Filaments for distributed-memory multiprocessors. The challenge is to overlay computation with communication. If we are able to get good performance—preliminary results are encouraging [Free94]—then fine-grain parallelism will indeed provide an architecture-independent infrastructure for parallel computing. We are also working on mappings from a variety of languages (imperative, dataflow, and data parallel) to Filaments and on experiments with thread placement/load balancing strategies and tradeoffs.

Acknowledgements

The starting point for the design of Filaments was MultiSR, the multiprocessor version of the SR run-time system, which was designed and implemented by Dave Bakken. Peter Bigot provided helpful ideas on how to make fine-grain parallelism fast, based on his work modifying MultiSR to produce an implementation of Janus (a concurrent constraint language). Vincent Freeh assisted with the design of the Filaments package itself, in particular with how to make fork/join threads stateless.

References

[Ande88] Anderson, T. E., Lazowska, E. D., and Levy, H. M. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Transactions on Computers 38*, 12 (Dec. 1989), 1631-1644.

[Ande91] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: effective kernel support for the user-level management of parallelism. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, 95-109.

[Andr91] Andrews, G. R. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings Publishing Co., Redwood City, CA, 1991.

[Baas88] Baase, Sara. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 1988.

[Bers88] Bershad, B. N., Lazowska, E. D., and Levy, H. M. PRESTO: a system for object-oriented parallel programming. *Software—Practice and Experience 18*, 8 (August 1988), 713-732.

[Buhr88] Buhr, Peter A. and Stroobosscher, R. A. The uSystem: providing light-weight concurrency on shared memory multiprocessor computers running UNIX. *Software—Practice and Experience 20*, 9 (Sept. 1990), 929-964.

[Buhr92] Buhr, Peter A., Ditchfield, Glen, Stroobosscher, R. A., and Younger, B. M. uC++: concurrency in the object oriented language C++. *Software—Practice and Experience 22*, 2 (Feb. 1992), 137-172.

[Carr86] Carriero, N., Gelernter, D. and Leichter, J. Distributed data structures in Linda. *Thirteenth ACM Symp. on Princ. of Programming Languages*, January, 1986, 236-242.

[Coop90] Cooper, Eric C. and Draves, Richard P. C Threads. Internal note of the Mach research project, September 11, 1990.

[Cull93] Culler, David E., et al. TAM—a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing 18*, 347-370 (1993).

[Dewd85] Dewdney, A. K. Computer recreations. *Scientific American* (August, 1985), 16-24.

[Doep87] Doeppner, Thomas W. Threads: a system for the support of concurrent programming. TR CS-87-11, Dept. of Computer Science, Brown University, June 1987.

[Eage93] Eager, Derek L., and Zahorjan, John. Chores: enhanced run-time support for shared-memory parallel computing. *ACM Trans. on Computer Systems 11*, 1 (Feb. 1993), 1-32.

[Engl94] Engler, Dawson R. The implementation of efficient thread-based parallelism. In preparation.

[Free93] Freeh, Vincent W. A comparison of implicit and explicit parallel programming. TR 93-30, Dept. of Computer Science, The Univ. of Arizona, October 1993.

[Free94] Freeh, Vincent W., and Lowenthal, David K. Distributed Filaments: efficient fine-grain parallelism on a cluster of workstations. In preparation.

[Humm91] Hummel, S. F., and Schonberg, E. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development 35*, 5 (September/November 1991), 743-765.

[Kepp93] Keppel, David. Tools and techniques for building fast portable threads packages. TR 93-05-06, Dept. of Computer Science, University of Washington, May 1993.

[Lin90] Lin, Calvin, and Snyder, Lawrence. A comparison of programming models for shared-memory multiprocessors. *Proceedings of the International Conference on Parallel Processing*, St. Charles, Ill., 1990, p. II: 163-170

[Lowe93] Lowenthal, David K., and Engler, Dawson R. Performance experiments for the Filaments package. TR 93-26, Dept. of Computer Science, The University of Arizona, September 1993.

[Mark92] Markatos, E. P. and T. J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. *Proc. 1992 International Conference on Parallel Processing*, August 1992, p. I:258-267.

[Pres91] Press, William H., and Teukolsky, Saul A. Multigrid methods for boundary value problems. *Computers in Physics*, Sept./October 1991, 514-519.

[Stei92] Stein, D. and Shah, D. Implementing lightweight threads. USENIX 1992, June 8-12.

[Thom88] Thomas, Robert H. and Crowther, Will. The Uniform System: an approach to runtime support for large scale shared memory parallel processors. *Proceedings of the 1988 Conference on Parallel Processing*, p. 245-254, August 1988.

[Vand88] Vandevoorde, M. and Roberts, E. WorkCrews: an abstraction for controlling parallelism. *Int. Journal of Parallel Programming 17*, 4 (Aug. 1988), 347-366.

[Weic84] Weicker, R. P. Dhrystone: a synthetic systems programming benchmark. *Comm. ACM 27*, 10 (October 1984), 1013-1030.