Performance Experiments for the Filaments Package

David K. Lowenthal Dawson R. Engler

TR 93-26

Performance Experiments for the Filaments Package[1]

David K. Lowenthal   Dawson R. Engler

TR 93-26

Abstract

Ten representative benchmarks were run on two shared-memory multiprocessors using an efficient, fine-grain threads package called Filaments. This paper describes the implementation and performance of the applications and compares them to both coarse-grain and sequential counterparts. It also analyzes the results and explains why the fine-grain programs were faster or slower than the coarse-grain ones.

September 2, 1993

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported by the NSF grants under grant CCR-9108412 and CDA-8822652.

1 Introduction

The granularity of a parallel program refers to the amount of computation a process performs between synchronization and communication events. This definition allows granularity to be viewed as a spectrum. The coarsest-grain program is a sequential program, as one process does all the work (and there is no synchronization or communication). A coarse-grain parallel program creates one process per processor, and that process works on its part of the computation. For example, consider the problem of multiplying two n by n matrices. If one partitions the n^2 inner product computations among p processes, where p is the number of physical processors in the system, the result is a coarse-grain program. A fine-grain program creates processes (threads) that each consist of a small, independent unit of work. For matrix multiplication, each inner product can be computed in parallel by a logically distinct process. In fact, the decomposition could be even finer: a thread could be created to compute each multiplication on each inner product (n^3 threads). However, matrix multiplication is quite unnatural and inefficient to program in this manner, as locking would be necessary to add the multiplications to form an inner product.

The fine-grain program is easier to write, because it frees the programmer from the bookkeeping involved in clustering units of work into larger tasks. This clustering causes the programmer to get bogged down in algebraic details necessary to obtain a correct and efficient clustering. Some parallel programs, such as divide and conquer algorithms, do not in fact have any *a priori* fixed set of tasks. In this case the coarse-grain program must be completely different (and much more complicated) than the natural fine-grain solution. A fine-grain program is architecture independent in the sense that the same program could be executed on any number of processors, whereas a coarse-grain program is typically written for some fixed number of processors. Fine-grain parallelism can also be used to implement both explicit parallelism in imperative languages and implicit parallelism in functional or dataflow languages. Finally, if there are more processes than processors, there is the potential to balance the load automatically among the processors; in a coarse-grain program it is important that each process be assigned about the same amount of work, which is not necessarily a straightforward task.

Although fine-grain parallelism has attractive attributes, 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. Because of the ease of programming using the fine-grain model, we believe a small loss in efficiency is acceptable. Our goal is to keep this loss in efficiency under 15%.

The Filaments package [EAL93] is a software package that assists parallel programmers in writing efficient, fine-grain parallel programs for shared-memory multiprocessors. In particular, Filaments programs using a decomposition of a thread per point of a matrix can run competitively with hand-coded, coarse-grain programs. In this paper we describe the performance of the Filaments package on ten different applications: matrix multiplication, Jacobi iteration, convolution, Mandelbrot set calculation, Fast Fourier Transform, Gaussian elimination, multigrid, adaptive quadrature, calculating fibonacci numbers, and quicksort. For each application we compare the performance using Filaments to tuned coarse-grain programs and show that writing fine-grain programs with the Filaments package almost always leads to performance that is competitive with coarse-grain programs.

Filaments contains three types of threads: run-to-completion, barrier (iterative), and fork/join. Run-to-completion threads are run once; barrier threads are run some number of times, with barrier synchronization and termination checking performed after each iteration; fork/join threads create one or more threads, and then wait for them to complete. The implementation of Filaments is very efficient due to the lack of private stacks on a per filament basis. Filaments are run by servers, one per processor. When a filament needs to compute local results, it uses the server's stack. The lack of private stacks eliminates the need for costly context switches between threads, and also frees up cache space for data. There are other mechanisms for efficiency in Filaments, such as a lack of preemption, automatic load balancing, pruning for fork/join computations, control of thread placement for data locality, support for multiple barriers in a single thread function (continuations), and very efficient barrier synchronization. For further details on the Filaments package, see [EAL93] and [Eng93].

Should the reader be interested in the Filaments package or the programs used to obtain the results printed here, please contact the authors via email.

2 Experimental Assumptions

We used two machines in testing. The first was a 14 processor Sequent Symmetry 81 with a 16 MHz clock and a 64Kbyte mixed instruction and data cache. The other was a Silicon Graphics Iris 4D/340 multiprocessor, which has a 33 MHz clock, a 64 Kbyte instruction cache, 64Kbyte date cache, and a 256Kbyte secondary data cache.

We went to great lengths to try to ensure the tests were accurate and fair, and that all programs were efficient. Most importantly, the Filaments, coarse-grain, and sequential programs were implemented as similarly as possible. For example, they all declared the same variables in registers, used the same memory allocator, and used the same timing facilities. All tests were run in single-user mode. In order to avoid adding overhead, the sequential programs were written without any parallel constructs and the coarse-grain programs were written using vendor-supplied subroutine libraries.

In the times reported for experiments, we factored out the system time (which was typically almost zero). These times, as well as all times reported below, are the average of 3 test runs. They were normally very consistent, although occasionally tests were run again due to anomalies. In these cases, the times reported were the average times of the second group of test runs.

In the next section we explain and give the performance of our ten applications. We give times and speedups for Filaments and coarse-grain programs. *Reported speedups are relative to the sequential program time.* The last column includes the percentage time difference between Filaments and coarse-grain programs; a positive percentage indicates that the coarse-grain program ran faster, and a negative percentage indicates Filaments ran faster.

In choosing problem sizes, we generally strove to make the sequential program take about a minute. This allows for the potential for good speedup on the larger numbers of processors. It also allows the test runs to run in a reasonable amount of time (and we had limited single-user time).

3 Experiments

3.1 Matrix Multiplication

Consider the problem of computing the matrix product of two n by 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, functions containing thread code, and a main routine that initializes, times, 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 and the matrices, creates the filaments, and then starts the server processes. Pseudo-code follows:

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

matrix_mult(int i, j)
  real sum = 0.0  /* use local variable for cache hits */
  for k = 1 to n do
    sum = sum + a[i][k]*b[k][j]
  c[i][j] = sum
end

main()
  int server
  f_initialize(num_servers)
  create and initialize shared matrices
  start timer
  for i = 1 to n do {
    server = (i*num_servers)/n    /* server to use for row i */
    for j = 1 to n do
      f_rtc_thread(server, matrix_mult, i, j)
  }
  f_parallel(num_servers)
  stop timer
  print results
end
```

The call of `f_initialize` initializes the Filaments package to use *num_servers* server processes. The arrays are created dynamically so that the program does not need to be recompiled when the input size changes. 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 others are the arguments that will be passed to the thread when it is executed. 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; that call terminates when all threads have been executed. For this application we use run-to-completion filaments because the filaments are only run once.

The coarse-grain program also uses a strip assignment. This program also has to determine the starting and ending rows for each processor's strip. While this calculation is not incredibly difficult, any error will result in incorrect program execution. The Filaments program may not run as efficiently if some filaments are assigned to an incorrect processor, but will produce correct output. The important part of the coarse-grain program is:

```
real sum
int startrow, endrow

startrow = pid * N/W
endrow = startrow + N/W -1

for i = startrow to endrow do {
  for j = 1 to n do {
    for k = 1 to n do
      sum = sum + a[i][k]*b[k][j]
    c[i][j] = sum
  }
}
```

Below are the results on the Sequent and the Iris, with sizes 150 and 350, respectively.

Sequent: matrix size 150 Sequential time: 54.89 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	55.14	0.995	54.84	1.000	0.547
2	27.62	1.987	27.45	1.999	0.619
4	14.01	3.917	13.92	3.943	0.646
8	7.036	7.801	6.986	7.857	0.715
12	4.813	11.40	4.786	11.46	0.564

Iris: matrix size 350 Sequential time: 58.9 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	59.77	0.985	58.73	1.002	1.770
2	29.88	1.971	29.87	1.971	0.033
4	15.5	3.8	15.24	3.864	1.706

It is not surprising that the Filaments package is within 2% of the coarse-grain program in the worst case. There is a reasonable amount of work per thread (n multiplications and $n-1$ additions), which amortizes the Filaments overhead. The work per thread also increases with the problem size.

This along with the ability to assign threads to processors allows the Filaments package to perform very well.

3.2 Jacobi Iteration

Laplace's equation in two dimensions is the partial differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0.$$

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 barrier filaments, because we want to avoid thread creation on each iteration. 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 a dynamically allocated two-dimensional vector of matrices. Variables *old* and *new* are used to index into this array. The boundaries of the region are stored in the edges of *grid*.

```
shared real grid[2][n+2][n+2]   /* created dynamically */
shared real maxdiff = 0.0
shared int old = 0, new = 1, k = 0
```

Procedure `jacobi` contains the code executed by each thread:

```
jacobi(int i, j)
  real temp

  grid[new][i][j] = (grid[old][i-1][j] + other neighbors)/4
  temp = absval(grid[new][i][j] - grid[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 $maxdiff$ needs 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()
  k++; if (k > MAXITERS or maxdiff < EPSILON) then return DONE
  old = new; new = 1-new; 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 procedure is:

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

The `f_new_barrier` indicates that the routine `sequential_code` will be called at a barrier point. As in matrix multiplication, filaments are again assigned in strips to maximize locality.

The coarse-grain program looks much like the coarse-grain matrix multiplication program, except that barrier synchronization is necessary. Its pseudocode is:

```
int startrow, endrow
real temp, localdiff;
bool done

compute startrow, endrow

done = false

while not done do {
  for i = startrow to endrow do
```

```
    for j = 1 to n do {
      grid[new][i][j] = (grid[old][i+1][j]+other neighbors)/4.0
      temp = absval(grid[new][i][j] - grid[old][i][j])
      if temp > localdiff
        temp = localdiff
    }

  aquire lock
  if localdiff > maxdiff
    maxdiff = localdiff
  release lock

  barrier

  if id = 0 then
    if maxdiff < EPSILON then done = true
    else maxdiff = 0;

  barrier
}
```

Below are the results on the Sequent and the Iris, with sizes 150 and 300, respectively.

Sequent: jacobi size 150 Sequential time: 61.34 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	69.0	0.888	62.3	0.984	10.75
2	34.42	1.782	31.17	1.967	10.42
4	17.36	3.533	15.8	3.882	9.873
8	8.65	7.091	7.9	7.764	9.493
12	5.953	10.30	5.403	11.35	10.17

Iris: jacobi size 300 Sequential time: 16.35 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	24.46	0.668	16.71	0.978	46.37
2	12.14	1.346	8.263	1.978	46.92
4	5.983	2.732	4.116	3.972	45.35

There is very little work per thread, as each filament only performs 3 additions, a division, a subtraction, a comparison, and conditionally performs another comparison and a lock acquisition and release. Also, the work per thread does not increase with the problem size; it is constant. Even so, on the Sequent the Filaments program is consistently within 11% of the coarse-grain program.

On the Iris, the Filaments program is consistently around 45% slower than the coarse-grain program. About 15% is due to Filaments overhead for the same reasons as discussed above. Most of the other 30% time difference is due to the Iris compiler's ability to perform optimizations 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 = startrow to endrow do
  for j = 1 to n do {
    grid[new][i][j] = (grid[old][i+1][j]+...)/4.0
    ...
  }
```

Compare this to the important 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, the 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 loaded once and 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 main computational chunk of this program is three times longer for the Filaments program than for the coarse-grain program. This is a problem with fine-grain programs in general — it is especially pronounced in Jacobi because the thread code does not contain a loop. (If the thread code 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's C compiler did not attempt to optimize the coarse-grain program.

In defense of fine-grain programs, for problems in which threads execute in a predetermined order (i.e. no threads migrate), a good compiler might be able to eliminate common subexpressions. The constants can always be saved across thread executions. Also, because loops are eliminated, 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 have the advantage of being more amenable to compiler optimizations.

To make sure the Iris compiler's lack of optimization of the Filaments program was responsible for the large overhead on Jacobi, we manually rewrote the Filaments program, to eliminate common subexpressions. The resulting Jacobi program on the Iris was indeed around 15% slower than the coarse-grain Jacobi program, which is from the overhead from Filaments. (Of course, no user would ever want to manually eliminate common subexpressions, and usually would not do so. It just shows that thread packages have no inherent limitation in efficiently executing fine-grain programs, as has been previously claimed by many, for example [LS90]).

3.3 Convolution

Convolutions are frequently used in engineering and other areas to solve problems such as polynomial multiplication [Baa88]. The convolution of two n length vectors, A and B, is the vector C such that the i^{th} component of C is given by $C_i = \sum_{j=0}^{n-1} A_j B_{i-j}$, where the indices on the right-hand side are taken modulo n. The algorithm is very simple; following is the code for procedure `conv`, which each thread executes. A, B, and C are shared vectors that are created dynamically.

```
conv(int i, n)
  for k = i to n*n do  /* note loop starts at i, not 1, so load imbalance  */
    C[i] = C[i]+A[k]*B[k-i]
end
```

The main procedure contains the following:

```
main()
  initialization code
  for i = 0 to n*n do
    f_rtc_thread(random() mod nservers, conv, i, n)
end main
```

This application is load imbalanced, because each thread runs a different number of iterations of a loop. Thus, the filaments program needs to balance the load to get good performance. One method for balancing the load in such applications is to use a random assignment policy. Generally, the drawback to this scheme is a decrease in cache hits, because processors do not work on a local group of points. However, in this application there is very little locality, so random assignment of run-to-completion threads, one per point, is used to provide load balancing. As there is a lot of work, a random assignment will approximate a load balanced distribution.

The coarse-grain program divides up the work cyclically, as the computation of point i is assigned to processor i *mod* P, where P is the number of processors in the system. A cyclic assignment for this application turns out to be the best assignment to achieve a load-balanced program other than random assignment, There is not a natural notion of random assignment for a coarse-grain program. We tried a strip assignment; the results were predictably terrible, as the last processor got most of the work, causing severe load imbalance. The coarse-grain program is:

```
for i = pid to n*n by W do
  for k = i to n*n do
    C[i] = C[i] + A[k] * B[k-i]
```

Below are the results of running convolution on the Sequent and the Iris, with vector sizes 40 and 100 respectively.

Sequent: conv size 40 Sequential time: 25.75 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	25.41	1.035	25.32	1.039	0.355
2	13.89	1.894	13.78	1.910	0.798
4	7.15	3.681	7.17	3.670	-0.27
8	3.736	7.044	3.596	7.319	3.893
12	2.486	10.58	2.443	10.77	1.760

Iris: conv size 100 Sequential time: 39.58 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	39.93	0.991	39.14	1.011	2.018
2	19.9	1.988	19.97	1.981	-0.35
4	10.08	3.926	10.22	3.702	-1.36

The Filaments programs obtained very good speedups, and these times were nearly identical to the times of the coarse-grain program. The coarse-grain program benefits from the knowledge that every point i will require more work than point $i-1$. Thus, a cyclic assignment of work will result in a perfectly load balanced program. In general, a cyclic assignment of work will perform very well in a coarse-grain program, although there are load-imbalanced programs for in cyclic assignment will perform poorly (see next section).

3.4 Mandelbrot Set Calculation

The Mandelbrot set [Dew85] is the set of complex numbers z such that $(z^2+z)^n$ is bounded as n approaches infinity. To compute the set, one selects a region and a resolution. The region specifies the range of values to test, and the resolution specifies the distance between complex numbers to be tested for membership (the resolution can intuitively be thought of as a mesh spacing). After each iteration, each point is checked to see if it has exceeded a selected bound (our bound is 100). If it has, computation of that point ceases; otherwise, another iteration is performed. This continues until some specified number of iterations are performed. At this point, all numbers that have not exceeded the bound are in the Mandelbrot set. Because each point requires a different amount of computation, this application is load-imbalanced.

Procedure `mandelbrot` contains the computation for each point on the grid. All elements of `pixel_done` are assumed to be initialized to zero. The `pixel_x` and `pixel_y` arrays hold the current value of each point. Identifier `pixel_val` holds the number of iterations a point took to converge (if it converges), and `BOUND` is the bound.

```
mandelbrot(i, j)
```

```
real x1, x2, x, y

for iter = 1 to MAXITERS do {
  if (!pixel_done[i][j]) then {
    x=pixel_x[i][j];
    y=pixel_y[i][j];
    x1=x*x-y*y+xval[i];
    y1=2*x*y+yval[j];
  }
  if (magnitude of this complex number > BOUND) then {
    pixel_done[i][j] = 1
    pixel_val[i][j] = iter
  }
  else {
   pixel_x[i][j]=x1;
   pixel_y[i][j]=y1;
  }
}
end
```

The main procedure is virtually identical to that of matrix multiplication. A cyclic assignment policy using run-to-completion threads was used in the Filaments program, as this typically results in good performance for Mandelbrot (see below). The important part of the coarse-grain program was as follows (W is the number of workers):

```
compute startrow, endrow

for i = startrow to endrow by W do
  for j = 1 to N do
    for iter = 1 to MAXITERS do
      (same code as for Filaments version)
```

Again, the coarse-grain program used a cyclic assignment — each worker works on every W^{th} row. As with the Filaments program, a cyclic assignment balances the load well for this application. We tried a strip decomposition in the coarse-grain program, and it ran *very* slowly, as some processors contained a whole strip of rows in which most points diverged very quickly.

Below are the results on the Sequent and the Iris, with sizes 150 and 300, respectively.

Sequent: mbrot size 150 Sequential time: 52.94 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	53.07	0.997	53.19	0.995	-0.22
2	26.59	1.990	26.63	1.987	-0.15
4	13.33	3.971	13.36	3.962	-0.22
8	6.75	7.842	6.766	7.824	-0.23
12	4.493	11.78	4.516	11.72	-0.51

Iris: mbrot size 300 Sequential time: 27.55 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	27.84	0.989	27.4	1.005	1.605
2	13.89	1.98	13.8	1.996	0.652
4	6.940	3.939	6.936	3.972	0.057

Mandelbrot is similar in style to convolution: both applications are load-imbalanced with no locality inherent in the application. The only difference is that in Mandelbrot, there is no pattern to how much work each point will require. However, since the amount of work each point performs is completely random, the cyclic assignment worked very well on the coarse-grain program. The Filaments program used the same assignment policy. A random assignment policy will work just as well, but the cost of using the random function caused the Filaments program to run a little slower than without it. (Adjoint convolution only incurs n calls to random, whereas Mandelbrot would have incurred n^2.) Again, it is important to note that Filaments programs can trivially use random assignment to load balance programs that have a lot of work and a small amount of locality. This saves the programmer from worrying about which assignment policy will perform well on a specific load-imbalanced program. Also, for some applications a random assignment policy will outperform any other static policy. It is very difficult to program a random assignment policy with a coarse-grain program. Thus, without writing complex code, some coarse-grain implementations will exhibit load-imbalance.

3.5 Fast Fourier Transform

A Fourier Transform is used to approximate a function with another curve comprised of sines and cosines. The naive way to compute a Fourier Transform takes $O(n^2)$ complex additions and multiplications. Many people simultaneously discovered that the terms use common intermediate values. Cooley and Tuckey are usually given credit for the discovery of the Fast Fourier Transform (FFT) [CT85]. Their FFT is called the Radix-2 FFT, and the number of complex arithmetic operations is $O(n \log n)$. This algorithm runs for $\log n$ steps. At each step new intermediate values are computed for each of the n elements in the vector. Each new value is computed based on two values: its old value and one other value. This other value, which changes on each step, is determined by a *butterfly* pattern. With such a pattern, element i will first use a value that is a distance $n/2$ away. On successive steps, element i uses values that are $n/4$, $n/8$, etc. away. There are $log(n)$ steps in a such a butterfly-style algorithm.a

We run a two-dimensional FFT, which is widely used in image processing. A two-dimensional FFT is performed by first performing a sequential FFT on each row, and then a sequential FFT on each column. This program is easily parallelized in a shared-memory setting, assuming the code for a sequential FFT is available (eg. see [CT85]). The Filaments program simply creates one Filament per row of the matrix, and this filament performs a sequential FFT on that row. Then one filament per column is created, and another sequential FFT is performed. Pseudocode follows:

```
shared complex **x

/* rowfft and colfft are just sequential FFT's, the only difference is that
   colfft is set up to work on columns to avoid transpose  */

main()

  for i := 0 to N-1 do
    f_rtc_thread(i*nservers/N, rowfft, x[i])

  for i := 0 to N-1 do
    f_rtc_thread(i*nservers/N, colfft, x, i)

end
```

Below are the results on the Sequent and the Iris, with sizes 32768 and 65536, respectively.

Sequent: fft size 256 Sequential time: 55.24 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	55.22	1.000	55.23	1.000	-0.01
2	27.66	1.997	27.65	1.997	0.036
4	13.85	3.988	13.84	3.991	0.072
8	6.956	7.941	6.94	7.959	0.230
12	4.8	11.50	4.793	11.52	0.146

Iris: fft size 1024 Sequential time: 57.38 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	58.52	0.980	58.26	0.984	0.446
2	30.13	1.904	29.99	1.913	0.466
4	15.46	3.711	16.3	3.520	-5.43

One can see that the Filaments programs run nearly identically to the coarse-grain programs. A one-dimensional FFT requires a *lot* of work, and each filament is performing several such FFT's. Thus, the result is not surprising. We should mention that the two-dimensional FFT is not nearly as straightforward on a distributed memory machine, because of the complex communication patterns.

3.6 Gaussian Elimination

Gaussian elimination is a classic method for solving the linear system $Ax = b$. For an n by n matrix, n iterations are performed. On the j^{th} iteration, column j is pivoted and normalized. The pivot value is chosen to be the largest element (in absolute value) in the column in order to minimize round-off error. Then, all rows below row j are updated in an elimination phase.

The Filaments program uses one filament per column. Because the values in the pivot row are accessed in the elimination phase, a barrier synchronization is required after the pivot phase. In addition, the pivoting phase requires that the elimination phase be complete, which results in another barrier after the elimination phase. Therefore, we need to not only use barrier threads, but also allow for two barriers (as opposed to one in Jacobi iteration). We use the Filaments mechanism for multiple barriers: multiple calls to `f_new_barrier()`, with thread creations in between these calls. The first group of threads are run, then the first barrier is reached (and the associated sequential code is run), then the second group of threads are run, and then the second sequential code is run, and so on.

Gaussian elimination exhibits load-imbalance because after a column is pivoted, it is never accessed again. In fact, on each iteration, one row and one column are accessed for the last time. On iteration j, only an $n - j + 1$ by $n - j + 1$ submatrix is accessed. The Filaments program achieves load balancing by assigning the columns cyclically to processors. Pseudocode follows:

```
  shared int k;  /* just the outer loop counter */
  shared int N;
  shared int A[N][N];  /* matrix operated on, created dynamically */
  shared int pivrow;  /* row in which largest value is found */
```

Procedure `dopivot` is invoked by each thread; however, only one thread actually computes the pivot value.

```
dopivot(int j, n)
  if (j == k-1)  then {  /* i am the pivoter */
    pivot(k-1, &pivrow)  /* procedure pivot returns the desired row */
    swap(A[pivrow][k-1], A[k-1][k-1])  /* largest value now in pivot position */
    for i = k to n-1 do  /* eliminate my column */
       A[i][j] = A[i][j] - A[k-1][j] * A[i][k-1]/A[k-1][k-1]
  }
```

Procedure `eliminate` is invoked by each thread.

```
eliminate (int j, N)
  int i;

  if j >= k then {  /* if column j is to the right of  the current pivoting column */

    swap(A[pivrow][j], A[k-1][j]) /* help complete pivot */
```

```
    for i=k to N-1 do  {  /* only eliminate points to right of pivot */
      A[i][j] = A[i][j] - A[k-1][j] * A[i][k-1]/A[k-1][k-1];

  }
end
```

The sequential code run after `eliminate` is:

```
sequential_code()
  if ++k < N then return NOTDONE;  /* just managing the outer loop counter */
  else return DONE;
end
```

The main procedure is somewhat different here, as we have multiple barriers. It looks as follows:

```
main()
  f_new_barrier(null_code)  /* null_code just returns F_NOTDONE (simple barrier) */
  for j = 0 to n-1 do
    f_bar_thread(j%nservers, dopivot, j, n)  /* these threads are associated with null_code */

  f_new_barrier(sequential_code)
  for j = 0 to n do
    f_bar_thread(j%nservers, eliminate, j, n)  /* these threads are associated with sequential_code */
end
```

Run-to-completion threads could be used here since we know how many iterations will be performed, but our measurements showed that the creation overhead was slightly greater than the overhead of checking for loop termination. As stated above, we create one filament per *column*, because that is the most natural way to code Gaussian elimination. If one wanted to code this problem with a filament per point, then the pivoting code would have to contain several conditionals. To better understand this, consider that if there were a filament per point, there would be several actions a thread could take on a given iteration: a thread could be idle, a thread could be responsible for the actual pivot, a thread could be responsible for helping to complete the pivot, or a thread could be responsible for eliminating a point. Typically when one thinks of the advantages of programming with a thread per point, one thinks of removing loops. But in this case, there is a blowup in conditionals, so the full advantage is not gained. There is even a disadvantage, because of increased code complexity.

We do not create one filament per row because that forces the pivoting routine to be sequential. Using one thread per column allows all processors to participate in the swap.

The coarse-grain program also uses a cyclic assignment. One added complexity in the coarse-grain program arises because of the possibility of the number of workers, W, not dividing the problem size, N. In this case, some workers are assigned N/W columns, and some are assigned $(N/W)+1$. All processors loop over the number of columns they are assigned, with a barrier synchronization in the middle of the loop. After iteration N/W, some processors exit this loop, and some perform one more iteration. Because *all* processors must participate in a barrier synchronization, the processor that exit the loop must perform one extra barrier (otherwise deadlock will occur). This additional complexity is an unfortunate attribute of the coarse-grain programming model. The important part of the coarse-grain program is:

```
startcol := myID  /* myID is which process i am */

for k = 1 to N-1 do {
  for j = startcol to N-1 by W do {
    if j = k-1 then {
      pivot(k-1, &pivrow)
      swap(A[pivrow][k-1], A[k-1][k-1])
      for i = k to N-1 do
        A[i][j] = A[i][j] - A[k-1][j] * A[i][k-1]/A[k-1][k-1]
    }

    barrier

    if j >= k then {
      swap(A[pivrow][j], A[k-1][j])  /* help complete pivot */
      for i = k to N-1 do
        A[i][j] = A[i][j] - A[k-1][j] * A[i][k-1]/A[k-1][k-1]
    }

  }

  if (this process worked on 1 fewer column than another processor)
    barrier /* see explanation above in text */

  barrier

  }
}
```

Below are the results on the Sequent and the Iris, with sizes 150 and 300, respectively.

Sequent: gauss size 150   Sequential time: 32.21 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	32.42	0.993	32.19	1.000	0.714
2	16.86	1.910	17.23	1.869	-2.19
4	8.956	3.596	9.313	3.458	-3.98
8	4.963	6.490	5.146	6.259	-3.68
12	3.65	8.824	3.866	8.331	-5.91

Iris: gauss size 300   Sequential time: 18.29 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	19.18	0.953	20.4	0.896	-6.36
2	12.69	1.441	17.11	1.068	-34.8
4	10.14	1.803	15.52	1.178	-53.0

We programmed Gaussian elimination in what we felt was the most natural way: by columns. As mentioned above, because each processor works on a set of non-contiguous columns, there will be many cache invalidations due to the row-major storage of C. On the j^{th} iteration, each element of the $n-j+1$ by $n-j+1$ submatrix will be written, causing an invalidation on every other processor. For this reason, the speedup in both the Filaments and coarse-grain programs is very poor. (When we implemented Gaussian elimination without pivoting, we were able to have each processor work on a series of rows and achieved near-perfect speedup.) It should be noted that another way to write this program would be first to transpose the matrix and then to assign rows to processors. We chose not to do this because it deviated significantly from the basic Gaussian-elimination program.

The Filaments programs run significantly faster than the coarse-grain programs on both machines. A profile of the code suggests that the vendor-supplied facility, called $m\_sync()$ on both the Sequent and the Iris, is highly inefficient when processors arrive at a barrier at varied times. In Jacobi iteration, the barriers do not affect program performance very much because all processors do the same amount of work and thus reach the barrier at about the same time. In contrast, Gaussian elimination has severe load imbalance in the pivoting code because one processor lags far behind the other processors on each iteration.

3.7 Multigrid

In section 3.2, we described Jacobi iteration as a method for solving partial differential equations. Jacobi iteration is a very straightforward algorithm but is also very slow. Each iteration involves an averaging at each grid point (plus other operations), and hence the effect of points on one boundary cannot reach points on the opposite boundary until at least n iterations. This can be completely unacceptable for large problem sizes.

In this section we describe multigrid methods for solving partial differential equations. Multigrid is a very complex algorithm: the code size is around 10 times larger than that of Jacobi iteration. However, the effect of points on one boundary can reach points on the opposite boundary in $log(n)$ steps, a significant improvement over Jacobi iteration.

Multigrid methods are based on what is known as a *coarse grid correction*, which works as follows. First, use a relaxation technique for a few iterations. Because of the inefficiency of such techniques, the number of iterations should be kept to a minimum (usually two or three). Then, restrict the result to a coarser grid, which has $\frac{1}{4}$ the area of the original grid. This coarsening is provided by a restriction operator R (one example of R might be to just copy certain points on the fine grid directly to the coarse grid, although typically a more sophisticated operator is used). Third, solve the problem completely on the coarse grid (for example by relaxation or direct methods). Fourth, interpolate this solution back to the fine grid using an interpolation, or prolongation, operator P (one example of this operator might be just to copy the points back to the fine grid and then average them with neighboring points). Finally, apply the relaxation technique on the fine grid for a couple of iterations. After the relaxation technique is applied to the fine grid, the values of boundary points will be affect far beyond the 5 points they would be normally felt with just 5 iterations of relaxation.

The multigrid method is just a slight extension of the above. Instead of solving exactly on the coarse grid, again relax for a few iterations, and restrict to an even coarser grid. This process is continued until a coarse enough grid is obtained so that iterating a relaxation scheme to convergence takes virtually no time. Then the interpolation step is done all the way back to the original grid, with another relaxation step in between each interpolation.

We can improve the efficiency of multigrid by using an algorithm known as the full multigrid method (FMG). A large amount of time in multigrid is spent iterating on the finest grids. Instead of starting from the finest grid, first fin the exact solution on the coarsest grid (of course, it's only a guess initially). Second, interpolate to the next finer grid. Then, find the exact solution on this grid, by applying the regular multigrid method (relax, restrict, interpolate). Only then do we proceed up to the next finer grid. In this manner as little time as possible is spent on the finest grids. Typically only one or two iterations is needed on each grid. An example of the progression of grids for a 3-grid FMG is given in Figure 1.

Our algorithm is an adaptation and parallelization of the FMG as explained in [PT91]. Only certain phases (relaxation, restriction, interpolation) are parallelized. Parallelizing some phases (such as solving directly) actually resulted in a decrease in performance. Each phase was parallelized separately, and assigned filaments to processors in strips. Below we discuss the interpolation and restriction phases. The relaxation phase is very similar to that shown in the section on Jacobi iteration, and will be omitted.

The interpolation phase requires the use of continuations and multiple barriers. Hence, the interpolation procedure is split into three procedures, `interp1`, `interp2`, and `interp3`. This interpolation, known as bilinear interpolation, uses the prolongation operator:

$$\begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{bmatrix}$$

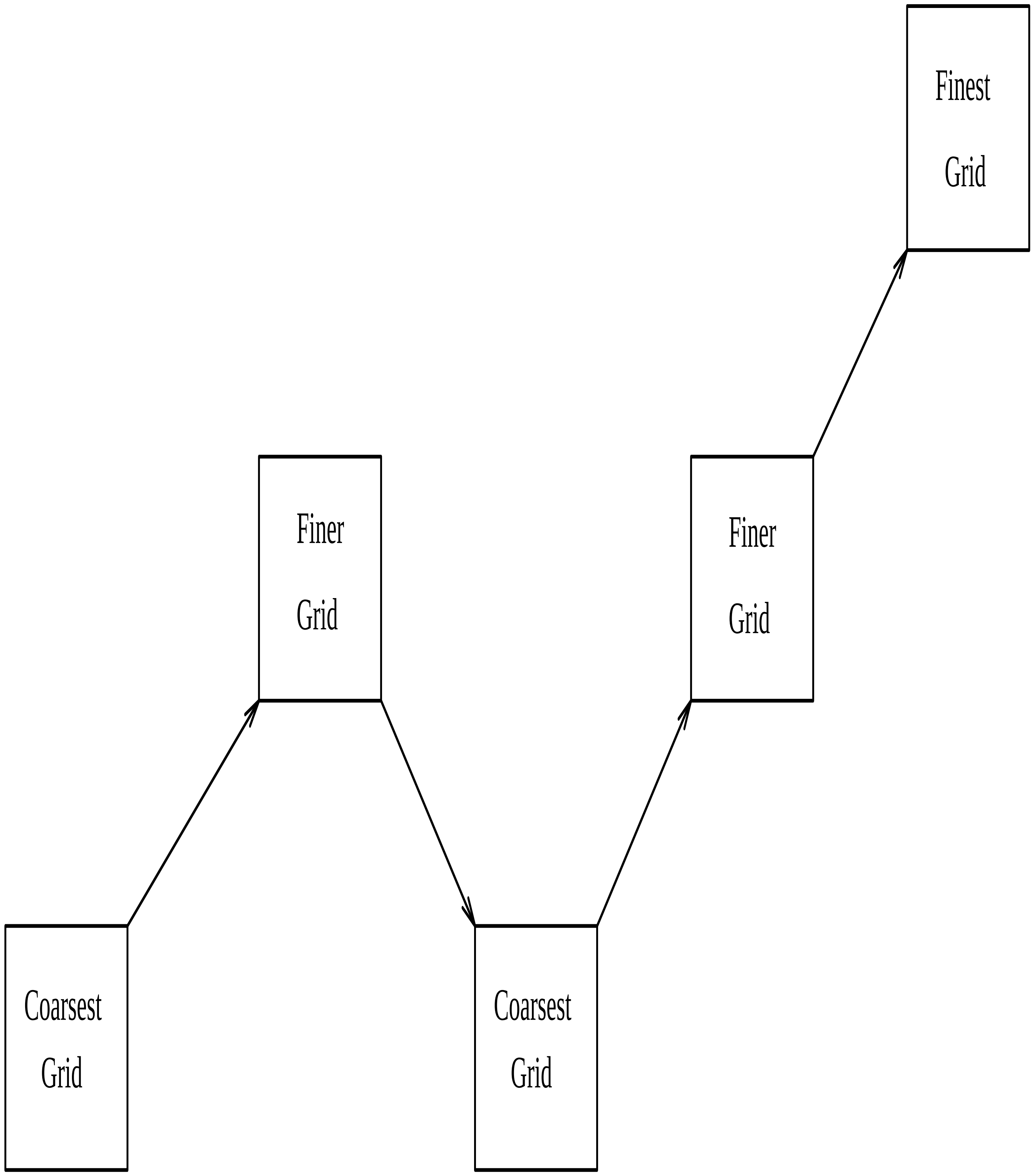

Figure 1: Example of Full Multigrid Method, 3 grids

This allows for a point on the coarse grid to have a direct effect on the corresponding fine grid point and a lesser effect on other neighbors.

Pseudocode follows:

```
interp1 (int pid, nf; double **uf, **uc)  /* uf and uc are fine and coarse grids */
  int jc, jf, nc = nf/2 + 1  /* nf, nc dimensions of fine and coarse grids */

  if pid <= nc then  /* only some processors will do the copy */
    for jc = 1 to nc do {
      jf = 2 * jc - 1   /* figuring out mapping between fine and coarse grid */
      uf[2*pid-1][jf] = uc[pid][jc]  /* do the copy */
    }
end

interp2 (int pid, nf; double **uf, **uc)
  int jf;
```

```
  /* first average the points over the columns that we copied in in
     procedure interp1 */

  if pid mod 2 = 0 then
    for jf = 1 to nf by 2 do
      uf[pid][jf] = 0.5 * (uf[pid+1][jf] + uf[pid-1][jf])
end

interp3 (int pid, nf; double **uf, **uc)
  int jf;

  /* next average over all the rows */

    for jf = 2 to nf-1 by 2 do
      uf[pid][jf] = 0.5 * (uf[pid][jf+1] + uf[pid][jf-1])
end
```

The code to set up the parallelization is a little different than we have seen so far. We use continuations for parallelizing the interpolation function. Continuations are used when several functions, each separated by a barrier, use the same number of threads, with the same arguments. (In Gaussian elimination, we could not use continuations because we did not create the same number of threads for each function.) Multiple barriers can be used to implement anything continuations can implement, but continuations are an optimization of multiple barriers since the same argument block is used for each function (and thus need be created only once, as opposed to several times with multiple barriers). Below, `null_code` just returns `F_NOTDONE` (so is a simple barrier). Procedure `f_new_continuation` specifies the routines to link together. The effect is that all threads run `interp1`, then `null_code` is run, then all threads run `interp2`, etc. Because we need more than three parameters, we have to use variable argument filaments. The string `iipp` indicates there will be four parameters, 2 integers followed by 2 pointers. The function to execute is not specified because `f_new_continuation` already did so. Identifier *grid* is a vector of matrices, ranging from the finest grid to the coarsest grid.

```
  f_new_barrier(null_code)
  f_new_continuation(interp1, interp2, interp3)
  for i = 1 to fineGridSize do {  /* size of grid we are interpolating onto */
    server = i*nservers/fineGridSize
    f_bar_threadV(server, NULL, "iipp", i, fineGridSize, grid[j], grid[j-1])
  }
```

The code for restriction is, in a sense, the opposite of that for the interpolation. For a restriction operator we use:

$$\begin{bmatrix} 0 & 1/8 & 0 \\ 1/8 & 1/2 & 1/8 \\ 0 & 1/8 & 0 \end{bmatrix}$$

So a coarse grid point is computed by taking $\frac{1}{2}$ of the corresponding fine grid point and adding to that $\frac{1}{8}$ of the north, east, west, and south neighbors. This choice of R allows the effect of several points on the fine grid to affect points that are far away, although it does ignore half of the neighbors. Boundary points that do not have neighbors are just copied directly onto the coarse grid. One thread is created for each row of the coarse grid; after computing the mapping from the fine to coarse grid, each thread computes each point in its row.

```
restrict(int nc, pid; double **uc, **uf)  /* nc is dimension of coarse grid */
  int ic, jc, if, jf

  /* copy a couple of boundary points */
  uc[pid][1] = uf[2*pid-1][1]
  uc[pid][nc] = uf[2*pid-1][2*nc-1]

  if pid != 1 and pid != nc then
    for jc = 2 to nc-1 do {
      jf = jc * 2 - 1
      if = 2 * pid - 1
      uc[pid][jc] = 0.5*uf[if][jf] + 0.125*(uf[if+1][jf] + uf[if-1][jf] +
                    uf[if][jf+1] + uf[if][jf-1])
    }

  else
    for jc = 1 to nc do
      uc[pid][jc] = uf[2*pid-1][2*jc-1]
```

The threads are created in the same manner as in the interpolation, except that we use run-to-completion threads in this case, because they simply do a restriction and then terminate.

Below we give a very high level outline of the entire multigrid method.

```
while not converged {
  get initial solution on coarsest grid
  while not yet at finest grid {
    interpolate to next finer grid (call it G)
    /* now going all the way to coarsest grid, and then back
       down to the current grid */
    while not at coarsest grid {
      relax on current grid
      restrict to next coarser grid
    }
    solve on the coarsest grid
    while not back at current grid (G) {
      interpolate to next finer grid
      relax on that grid
    }
```

```
  }
}
```

The coarse-grain program is omitted here, because the relaxation and interpolation, and restriction code is similar to that of Jacobi iteration, and the restriction code is similar to that of matrix multiplication.

Below are the results on the Sequent and the Iris, with sizes 129 and 513, respectively. (For programming convenience, the multigrid program in [PT91] is programmed to use a finest grid whose size is one more than a power of two.)

Sequent: multigrid size 129 Sequential time: 46.50 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	42.09	1.104	45.68	1.017	-8.52
2	22.67	2.051	23.88	1.947	-5.33
4	13.01	3.574	12.77	3.641	1.879
8	8.093	5.745	7.323	6.349	10.51
12	6.513	7.139	5.69	8.172	14.46

Iris: multigrid size 513 Sequential time: 42.23 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
	44.35	0.952	42.3	0.998	4.846
2	23.77	1.776	22.67	1.862	4.852
4	14.82	2.849	14.01	3.014	5.781

Multigrid is a very difficult algorithm on which to get a lot of speedup, because it is quite complex. As shown above, neither the coarse-grain nor Filaments programs got good speedup, but the Filaments program was generally competitive. The exception to this is the twelve processor test on the Sequent, where the Filaments program was around 15% slower than the coarse-grain program. We cannot figure out why this particular case ran so poorly. It does, however, motivate us to try to find more complex applications in the future, as they tend to give more insight into the (potential) inefficiencies of fine-grain parallelism.

3.8 Adaptive Quadrature

Consider the problem of approximating the integral

$$\int_a^b f(x)dx.$$

One method to solve this problem is adaptive quadrature. Divide an interval in half, approximate the areas of both halves and of the whole interval, and then compare the sum of the two halves to the area of the whole interval. If the difference of these two values is not within a specified tolerance, recursively compute the area of both intervals and add them.

The best way to program this is to use a divide-and-conquer approach. Because subintervals are independent, a new thread can be created to compute each subinterval. Hence this application uses fork/join threads.

Adaptive quadrature clearly can exhibit load imbalance if certain areas of the curve will require a lot of work and others will not, and if the processors receive an unequal amount of work. Adaptive quadrature is used for exactly those curves that need a very small mesh size for parts of it and a very large mesh size for others. In turn, this leads to load-imbalance in a parallel program and presents a challenge for the Filaments package. In order to ensure that load imbalance will indeed occur, below we compute the area under the function $f(x) = exp(x) * sin(x)$. This function will require more work at the right part of the interval since the oscillations will become sharper.

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 under f() from a to m and m to b
  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)
    f_join()  /* wait for children to terminate */
    return *left+*right
  }
end
```

The algorithm evaluates $f()$ just once at each point and evaluates the area of each interval just once. Previously computed values and areas are passed to new threads.

The main routine for adaptive quadrature is:

```
main()
  real left, right, *answer, fleft, fright, init_area
  f_initialize(num_servers)
  f_set_prune_threshold(MAXTHREADS)
  input left and right, then 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 whenever doing so will result in poor performance. In particular, the call above of `f_set_prune_threshold` 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. We are currently working on a dynamic mechanism to adjust the pruning threshold, which would obviate the user's need to set it.

Our coarse-grain program is very different from the fine-grain one. There are at least two ways to write the coarse-grain program, but there is no direct analog to the fine-grain program. One possible coarse-grain program is to divide up the intervals statically and assign intervals to processors. Each processor than performs adaptive quadrature on its intervals. But this method can (and likely will) exhibit load-imbalance, as early finishing processors cannot help compute the intervals that take more time. We take a different approach. Our program creates one process per processor and uses a shared job queue. Access to this queue is protected by a lock. Processes remove work when they are idle. If a process creates two subproblems are it keeps one and insert the other into the queue. To minimize contention, if there is no work on the queue, processes wait a certain amount of time before trying again to remove a job. We keep the queue size bounded to prevent the creation of too much parallelism; this is directly analogous to pruning. Each process keeps its total area in a privately indexed global vector, and then the total area is obtained when the algorithm terminates by summing this vector. Pseudocode follows for the `work` procedure:

```
work(int pid)
  while true {
    remove job from queue
    while (no job to remove) {
      increment counter to indicate this process is idle
      if counter = numWorkers then
        terminate program and sum total area
      backoff and remove job from queue
    }
    compute area of whole interval and sum of both halves
    if (estimates are close enough) then
      sum[pid] +:= sum of both halves
    else
      if (too many jobs outstanding) then
        sum[pid] +:= area()  /* area is a sequential adaptive quadrature fn. */
      else {
        insert left half of interval on queue
        keep right half for myself
      }
  }
end worker
```

We have left many details out of the pseudocode to make it comprehensible. One can see, however, that this way of programming is *much* more difficult than that of the simple recursive, fine-grain program.

Below are the results on the Sequent and the Iris, with sizes 27 and 30, respectively.

Sequent: quad size 27 Sequential time: 56.86 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	56.91	0.999	56.77	–	–
2	28.52	1.993	28.40	–	–
4	14.41	3.945	16.01	–	–
8	7.196	7.901	–	–	–
12	4.743	11.98	–	–	–

Iris: quad size 30 Sequential time: 8.593 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	8.586	0.987	8.55	0.991	0.421
2	4.326	1.960	4.25	1.995	1.788
4	2.18	3.889	2.343	3.619	-7.47

The Filaments programs got near perfect speedup on small numbers of processors, which is due to the efficient fork/join mechanism in the Filaments package. It balances the load when necessary and prunes computations when there is already enough work present. For larger numbers of processors, we had to tune the pruning threshold to obtain good performance, but the Filaments package makes this easy to do. With problems such as Fibonacci (see below), where each thread does very little work, pruning is vital to get reasonable performance.

The coarse-grain program performed fairly well on smaller numbers of processors, but dropped off somewhat on larger numbers. This is likely due to contention for the shared queue. Also, it is very important to note that we had to adjust the pruning threshold to get any kind of good performance. This required a lot of tuning — much more than the tuning the Filaments program required. The coarse-grain program seems to be much more volatile with respect to the tuning threshold than the Filaments program.

3.9 Computing Fibonacci Numbers

Given any number n, consider the problem of recursively calculating the n^{th} Fibonacci number. The first two Fibonacci numbers are 1 and 1, and thereafter each Fibonacci number is the sum of the previous two. This problem can be posed as a divide and conquer problem. To compute the n^{th} Fibonacci number, recursively and in parallel compute both the $n-1^{st}$ and $n-2^{nd}$ Fibonacci numbers, then add the results. Thus, computing Fibonacci numbers is a fork/join application, like adaptive quadrature.

Here is the pseudocode for the `fib` procedure:

```
fib(int num)
  int *res1, *res2
```

```
  if num = 1 then return 1;
  else {  /* recurse, forking two new threads */
    res1 = f_fork(fib, num-1)
    res2 = f_fork(fib, num-2)
    f_join()  /* wait for children to terminate */
    return *res1+*res2
  }
end
```

The code for the main program is nearly identical to that for adaptive quadrature.

The coarse-grain program is also nearly identical in style to the coarse-grain program for adaptive quadrature, and will be omitted.

Below are the results on the Sequent and Iris, computing the 32^{nd} and 35^{th} Fibonacci numbers, respectively.

Sequent: fib size 32 Sequential time: 34.38 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	34.4	0.999	34.39	0.999	0.002
2	17.25	1.993	17.25	1.993	0
4	9.026	3.808	8.58	4.000	5.198
8	4.35	7.903	5.08	6.76	-14.37
12	2.933	11.72	3.58	9.60	-18.07

Iris: fib size 35 Sequential time: 13.99 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	14.16	0.992	14.15	0.992	0.070
2	7.06	1.990	7.09	1.981	-0.42
4	3.58	3.924	3.58	3.924	0.0

Unlike adaptive quadrature, the program to compute Fibonacci numbers creates filaments that do virtually no actual computation: each filament performs only an add. This problem presents the worst case fork/join problem with respect to amount of work per thread. However, with the Filament package optimizations (namely pruning), near perfect speedup is achieved. After just a few forks, all processors have work, and no more forking is done. The coarse-grain program is competitive on the Sequent until the number of processors get large. The reason for this is likely contention for the shared job queue. For Fibonacci, the fine-grain program is clearly superior.

3.10 Quicksort

Quicksort is another fork/join application, but it has a lot of work per filament. Each thread will do a partition, which consists of two loops, before making a recursive call. Pseudocode follows:

```
quicksort(int start, finish)
  int pivot, temp
```

```
int left, right

if finish-start) < THRESH  /* avoid quicksorting small arrays */
  selectionsort(start, finish) /* selectionsort not shown */

else {

  left = start
  right = finish
  pivot = A[(start+finish)/2]

  while (left < right)  {
    while (A[left] < pivot) left++
    while (A[right] > pivot) right--
    if left <= right {
      swap(A[left], A+[right])
      left++; right--
    }
}

/* fork two children */

if (start < right) f_fork("v",quicksort,start, right, NULL)
if (left < finish) f_fork("v",quicksort,left, finish, NULL)

f_join()

}
```

Again, the main procedure is similar to the other fork/join applications, and the coarse-grain program is similar to the that of adaptive quadrature and quicksort, so both will be omitted.

Below are the results on the Sequent and the Iris, with sizes 100,000 and 500,000, respectively.

Sequent: qsort size 100000   Sequential time: 17.83 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	17.9	0.996	17.98	0.991	-0.44
2	9.176	1.943	9.249	1.927	-0.79
4	4.853	3.674	6.589	2.706	-35.7
8	2.946	6.052	4.279	4.166	-45.2
12	2.486	7.172	4.196	4.249	-68.7

Iris: qsort size 500000   Sequential time: 9.36 sec.

Processors	Filament Time	Filament Speedup	CG Time	CG Speedup	% Slower
1	10.75	0.871	9.586	0.977	12.14
2	5.383	1.739	5.26	1.780	2.338
4	2.75	3.405	2.816	3.325	-2.4

Quicksort does not run very well on the Sequent. This application has a lot of work per filament, which generally makes for high performance. However, if some processors are idle, it doesn't matter how much work each filament performs. Because of quicksort's time-consuming `partition` routine, it takes time for all processors to acquire work. For example, the initial node must partition the array, which involves two linear loops inside another loop (as shown above). Only at this point does the initial node fork 2 threads. If the input size is on the order of hundreds of thousands of numbers, this (sequential) partitioning can take a considerable amount of time. The time taken to distribute work is too much, and speedup is poor. We believe the Iris time is relatively better than the Sequent because of its faster processor, which allows the `partition` routine to be completed faster and processors to acquire work more quickly.

The Filaments programs again outperform the coarse-grain programs. Again, the coarse-grain program has to deal with contention for the shared bag when the number of processors is large. Also, we also had to spend a significant amount of time adjusting the pruning threshold on the coarse-grain program.

4 Conclusion

We have shown a wide range of applications to be efficiently implementable with the Filaments package. Most were easily written and immediately showed good performance. The programs that were hard to implement efficiently were Jacobi iteration, quicksort, and multigrid.

We explained that the inefficiency of Jacobi iteration was due to the compiler performing optimizations for the coarse-grain program that it could not for the fine-grain program. In addition, quicksort does not run efficiently because a large part is inherently sequential. Neither of these inefficiences is due to inefficiency of the Filaments package.

The poor performance of multigrid on the Sequent is still an open problem. Furthermore, we had to program multigrid with a thread per row, because the version that used a thread per point ran too slowly. Multigrid has multiple phases, each of which is often run on a very small problem size (e.g. relaxation on a five by five grid). This tends to make fine-grain computing inefficient, as a reasonable amount of work is needed to amortize the (relatively small) overhead of the Filaments package.

Due to the relative ease of fine-graim programming, we consider the small disadvantage in performance of fine-grain programs to be completely acceptable. The current state of parallel computing is poor because applications programming is just too difficult. We feel that fine-grain programming is one step in the right direction to making parallel computing easier.

We currently are porting Filaments to a distributed memory multiprocessor, where we plan to integrate the package into a distributed shared memory system. In the future we also plan to use Filaments as a target for a parallel programming language to be developed and as a machine-independent intermediate form for several existing languages.

References

[Baa88] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 1988.

[CT85] J.M. Cooley and J.W. Tuckey. An algorithm for the machine calculation of complex fourier series. *Math. Comp.*, pages 297–302, 1985.

[Dew85] A. K. Dewdney. Computer recreations. *Scientific American*, pages 16–24, August 1985.

[EAL93] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Efficient support for fine-grained parallelism. Technical Report 93-13, Dept. of Computer Science, University of Arizona, April 1993.

[Eng93] Dawson R. Engler. Filaments: the design and implementation of efficient efficient fine-grain parallelism. Technical report (in preparation), Dept. of Computer Science, University of Arizona, August 1993.

[LS90] Calvin Lin and Lawrence Synder. A comparison of programming models for shared memory multiprocessors. *ICPP*, 10(1):163–170, January 1990.

[PT91] William H. Press and Saul A. Teukolsky. Multigrid methods for boundary value problems 1. *Computers in Physics*, pages 514–519, September/October 1991.