Efficient Support for Fine-Grain Parallelism on Shared-Memory Machines

Vincent W. Freeh
David K. Lowenthal
Gregory R. Andrews

Efficient Support for Fine-Grain Parallelism on Shared-Memory Machines

Vincent W. Freeh David K. Lowenthal Gregory R. Andrews

TR 96-1

Abstract

A coarse-grain parallel program typically has one thread (task) per processor, whereas a fine-grain program has one thread for each independent unit of work. Although there are several advantages to fine-grain parallelism, conventional wisdom is that coarse-grain parallelism is more efficient. This paper illustrates the advantages of fine-grain parallelism and presents an efficient implementation for shared-memory machines. The approach has been implemented in a portable software package called Filaments, which employs a unique combination of techniques to achieve efficiency. The performance of the fine-grain programs discussed in this paper is always within 13% of a hand-coded coarse-grain program and is usually within 5 percent.

January 29, 1996

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

[1]This work was supported by NSF grants CCR-9415303 and CDA-8822652.

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. 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 because parallelism is expressed in terms of the problem size, not the number of processors that execute the program. Fine-grain programs are easier to write, because it is not necessary to statically cluster independent units of work into a fixed set of larger tasks; 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 loop parallelism either extracted by parallelizing compilers or expressed in parallel variants of languages such as Fortran. The fine-grain model simplifies code generation for such languages. Finally, when there are many more processes than processors, it is often easier to balance the total amount of work done by each processor.

This paper illustrates the usefulness of fine-grain parallelism by examining several applications and then presents an efficient implementation of fine-grain parallelism. The focus here is on shared-memory multiprocessors. Although most current work on supporting massive parallelism concentrates on machines with physically distributed memory, supporting small-scale parallelism on multiprocessors is still important. It will be more important as networks of multiprocessors become more common.

Our approach has been implemented in a portable software package called Filaments, which also supports fine-grain parallelism and a shared-memory programming model on both shared- and distributed-memory machines [FLA94]. It achieves efficiency in iterative computations by using stateless threads and implicitly coarsening granularity and in fork/join computations by pruning and automatically balancing the load. Filaments strives to make languages and compilers simpler by moving complexity into the run-time system. It has been used as a runtime library for parallel programs written in C and as the runtime system for a compiler for the functional language Sisal [Fre95].

The remainder of the paper is organized as follows. The next section gives an overview of fine-grain parallelism and the Filaments package. Section 3 describes how the package is implemented. Section 4 gives performance results for a variety of applications. Section 5 discusses related work. Finally, Section 6 contains concluding remarks.

2 Fine-Grain Parallelism and the Filaments Package

The most natural granularity to use when writing a parallel program depends on the application; however, fine-grain programs are usually the simplest. This section describes three applications in some detail: matrix multiplication, Jacobi iteration, and adaptive quadrature. We show how to program each application as a fine-grain shared-memory program using the Filaments package.

The Filaments package supports two kinds of very lightweight threads, which are called *filaments*. *Iterative* filaments execute repeatedly, with a global reduction operation (and hence a barrier synchronization) occurring after each execution of all filaments. The package also supports sequences of iterative filaments, which are used when loop bodies have multiple phases, each of which ends in a barrier. Iterative filaments are used in applications such as Jacobi iteration, LU decomposition, SOR, and multigrid. *Fork/join* filaments recursively fork new filaments and wait for them to return results. They are used in divide-and-conquer applications such as adaptive quadrature, quicksort, and recursive FFT. A *server thread* on each processor executes filaments.

A program that uses Filaments contains three additional components relative to a sequential program:

- declarations of variables that are to be located in shared memory,
- functions containing the code for each filament, and
- a section that initializes the package, creates the filaments, places them on processors, and times and controls their execution.

Upon creation, each iterative filament is placed in a *pool*, which is a group of filaments that ideally have similar data-reference patterns. A collection of pools is called a *pool set*. Pool sets additionally consist of a pointer to the function that each filament calls and a pointer to another function that is called after each execution of all filaments in the pool set; the latter function most often synchronizes the processors and checks for termination. Hence, a pool set logically represents a parallel code segment terminated by a barrier synchronization point.

Conversely, fork/join filaments are created dynamically and in parallel. When a processor forks a filament, it is placed on the processor's list; however, any processor may execute the filament. Fork/join applications do not have inherent locality. Hence, pools are not used because the data-reference patterns of fork/join filaments cannot be statically determined.

2.1 Matrix Multiplication: Simple Iterative Filaments

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; there are n^2 inner products.[1] The shared variables for this application are the three `n` $\times$ `n` matrices: `a`, `b`, and `c`. Each inner product is computed by a filament that executes the following function:

```
void inner_prod(int i, int j) {
  int k;
  double sum = 0.0;
  for (k = 0; k < n; k++)
    sum += a[i][k]*b[k][j];
  c[i][j] = sum;
}
```

The key roles of the initialization section are to initialize the Filaments package, create and initialize the matrices, and create the filaments themselves. This section is executed on only one server thread.

[1] 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.

```
void main() {
  /* declarations of local variables */
  /* create and initialize the shared matrices */
  fInit();
  ps = fCreatePoolSet(inner_prod, NULL);
  pool = fCreatePools(ps, num_fil);
  for (i = 0; i < n; i++) {
    pool_id = whichPool();  /* calculate pool to use for filaments in row i */
    for (j = 1; j < n; j++)
      fCreateFilament(ps, pool[pool_id], i, j);
  }
  fStart();
}
```

The call to `fInit` initializes the Filaments package. The call of `fCreatePoolSet` creates a pool set, which contains filaments that execute the function `inner_prod`. The last argument is a pointer to a function for the *sequential code*. This is the reduction/termination checking routine, which is executed after all filaments in the pool set have completed; it is not needed in this application (hence the `NULL` pointer). The `fCreatePools` call creates an array of pools, one per processor, where each pool contains space for `num_fil` filaments. The `fCreateFilament` routine creates a single filament. Each filament is defined by `i` and `j`, which are passed as arguments to `inner_prod`. The value of `pool_id` indicates the pool into which the filament is to be placed. For this application, the programmer-defined function `whichPool` should, assign strips of filaments to the same pool; i.e., each server thread should execute filaments that compute the results for a strip of the result matrix. This will provide data locality and hence good cache performance.[2]

The final Filaments package call, `fStart`, starts the parallelism. All filaments are executed to completion before the call returns.

2.2 Jacobi Iteration: Iterative Filaments with Barriers

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. Jacobi iteration is one such method; it works as follows. 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 tolerance, `EPSILON`, of their respective old values. Because there are two grids, the n^2 updates are all independent computations; hence, all new values can be computed in parallel.

For this application, the key shared variables are the two `n` $\times$ `n` arrays, `new` and `old`, and the *reduction* variable `maxdiff`. The `new` and `old` variables are dynamically allocated two-dimensional vectors. (The boundaries of the region are stored in the edges of the arrays.) A reduction variable

[2] Because data locality is inherently algorithm-dependent, the decision about which server thread is to execute each filament has to be made outside the Filaments package, either by language annotations, compiler analysis, or run-time support. See [LA96] for details.

is a special kind of variable with one copy per processor. Here, maxdiff is of type double and is shared among all filaments on a processor. The local copy of a reduction variable can be accessed directly. In addition, such variables are used in calls to fReduce, which atomically combines all processor's private copies into a single copy using an associative/commutative operator such as add or minimum. A call of fReduce also results in a barrier synchronization.

The code executed by each filament computes an average and difference:

```
void jacobi(int i, int j) {
  double temp;
  new[i][j] = (old[i-1][j] + old[i+1][j] + old[i][j-1] + old[i][j+1]) * 0.25;
  temp = absval(new[i][j] - old[i][j]);
  if (temp > Local(maxdiff))
    Local(maxdiff) = temp;
}
```

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 maxdiff needs to be updated. The Local macro accesses the server thread's local copy of maxdiff.

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

```
int convergenceCheck() {
  fReduce(maxdiff, MAX);
  if (maxdiff < EPSILON)
    return F_DONE;
  swap(old, new);
  maxdiff = 0.0;
  return F_CONTINUE;
}
```

This code is executed sequentially by one processor at the end of every update phase, i.e., after every filament in the pool sets has been executed once. The call to fReduce combines all copies of maxdiff (using the MAX operator) into a single consistent copy. If convergenceCheck returns F_DONE, then the computation terminates; otherwise, the server thread on each processor starts another iteration by once again executing each filament in its pool of filaments.

The initialization section again initializes the Filaments package, creates and initializes the arrays, creates the pools and filaments, and then starts the server threads on each node.

```
void main() {
  /* declarations of local variables */
  /* create and initialize the shared variables */
  fInit();
  ps = fCreatePoolSet(jacobi, convergenceCheck);
  pool = fCreatePools(ps, num_fil);
  for (i = 1; i < n-1; i++) {
    pool_id = whichPool();     /* determine which pool to use for this row */
    for (j = 1; j < n-1; j++)
      fCreateFilament(ps, pool[pool_id], i, j);
  }
```

```
  fStart();
}
```

The above code is virtually identical to the main function for the matrix multiplication program. The only essential difference is that the call of `fCreatePools` has a third argument, `convergenceCheck`, the function that checks for convergence.

2.3 Adaptive Quadrature: Fork/Join Filaments

This section introduces the fork/join style of programming in Filaments. It also illustrates an application that has no natural coarse-grain counterpart.

Consider the problem of approximating the integral

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

One method to solve this problem is adaptive quadrature, which works as follows. 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 areas of both intervals and add them.

The best way to program adaptive quadrature is to use a divide-and-conquer approach. Because subintervals are independent, a new filament computes each subinterval. Hence this application uses fork/join filaments. The more rapidly the function is changing, the smaller the interval needed to obtain the desired accuracy. Therefore, the work load is not uniformly distributed over the entire interval.

The computational routine for adaptive quadrature is:

```
double quad(double a, double b, double fa, double fb, double area) {
  double left, right, fm, m, aleft, aright;
  /* compute midpoint m and areas under f() from a to m (aleft) and m to b (aright) */
  if (close enough)
    return aleft + aright;
  else {  /* recurse, forking two new filaments */
    fjFork(quad, left, a, m, fa, fm, aleft);   /* return value in left */
    fjFork(quad, right, m, b, fm, fb, aright); /* return value in right */
    fjJoin();  /* wait for children to complete */
    return left + right;
  }
}
```

The algorithm above 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 filaments to avoid recomputing the function and both areas. If the computed estimate is not close enough, the program forks a filament to compute the two subintervals, and then waits for them to complete. The results are stored in `left` and `right`.

The initialization section for adaptive quadrature is shown below.

```
void main() {
  double left, right, answer, fleft, fright, init_area;
  fInit();
  /* input left and right, then compute fleft, fright, and init_area */
```

```
    /* fork initial filament */
    fjFork(quad, answer, left, right, fleft, fright, init_area);
    fStart();  /* also serves as implicit join */
}
```

In this application, the program initially creates one filament that computes the area for the entire interval. As always, the call of `fStart` terminates when all filaments have terminated; in fork/join applications this serves as an implicit join to the initial `fjFork` call.

3 Implementation

The Shared Filaments package is currently implemented on a 4-processor Silicon Graphics Iris and a 14-processor Sequent Symmetry. The Filaments package is implemented entirely in software. It supports efficient fine-grain parallelism and the shared-memory programming model. This section describes key aspects of the implementation.

A filament can access any data that is allocated in shared memory. Variables are allocated in shared memory through a system-supplied routine (`shmalloc` on the Sequent and `malloc` on the Iris).

A filament does not have a private stack; it consists only of a (shared) function pointer, arguments to that function, and—for fork/join filaments—a few other fields such as a parent pointer and the number of children. Filaments are executed one at a time by server threads, which are traditional lightweight threads with a private stack. The server thread's stack provides a place for temporary storage so that filaments can compute partial results.

Filaments are never preempted, so only one server thread is created on each processor. All filaments are of equal importance, because parallel programs complete when *all* work is done. Hence, implementing fairness gains no advantage; it can only increase the overhead of executing filaments.

3.1 Implementing Iterative Filaments

Descriptors of iterative filaments are stored in pools represented by arrays; each element contains the arguments to be passed to the filament code. This allows an iterative filament descriptor to contain only arguments, avoiding a pointer to the next filament. This is important because having small filament descriptors leaves more room in the cache for program data. Iterative filaments are also executed in a back-and-forth order; at the end of an iteration, some of the filament descriptors and associated data will still be in the cache. Hence, on the next iteration the filaments are run in reverse order so that these descriptors and data elements do not need to be loaded into the cache.

Many Filaments programs attain good performance with little or no optimization. For example, in matrix multiplication, each filament performs a significant amount of work ($O(n)$ multiplications and additions), amortizing the filament overheads. However, achieving good performance for iterative applications that possess many filaments that perform very little work (e.g., Jacobi iteration) requires using *implicit coarsening*, which allows filaments in a pool to be executed as if the application were written as a coarse-grain program.[3] To implement implicit coarsening, we use two techniques: inlining and pattern recognition. This reduces the cost of running filaments, reduces the working set size to make more efficient use of the cache, and uses code that is easier for compilers to optimize.

[3]Systems such as Chores [EZ93] and the Uniform System [TC88] have a fine-grain specification and a coarse-grain execution model, but use preprocessor support. Filaments generates different codes at compile time and chooses among them at run time.

Inlining consists of directly executing the body of each filament rather than making a procedure call. In particular, when processing a pool, a server thread executes a loop, the body of which is the code specified for filaments in the pool. This eliminates a function call for each filament, but the server thread still has to traverse the list of filament descriptors in order to load the arguments.

The second technique is to recognize common patterns of filaments at run-time. The Filaments package recognizes regular patterns of filaments assigned to the same pool, and *at run time* switches to code that iterates over the filaments, generating the arguments in registers rather than loading from memory. The package currently recognizes a few common patterns (such as one- and two-dimensional loops) that support a large subset of regular problems; however, the general approach is capable of supporting other patterns.

3.2 Implementing Fork/Join Filaments

Fork/join filaments are used to create filaments dynamically and later wait for them to terminate and return results, making them quite different from iterative filaments. Most importantly, filaments must be capable of blocking and later resuming execution; this requires some method of saving state. We still have one server thread execute many filaments by using recursion to avoid switching context. We use 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. While waiting, the parent executes other threads. In other words, the parent *becomes* the server thread. This works correctly because parent/child relationships are ordered; i.e., children are always created after their parents and terminate before their parents.

Fork/join filaments also necessarily have a more complex structure than iterative filaments. Besides the argument list, a fork/join filament needs several other fields, such as a pointer to its parent, the number of children it has created, and where to put the return value. Fork/join filaments are also stored in linked lists to make load balancing easier (see below).

Some applications (e.g. quicksort, adaptive quadrature) that use fork/join filaments require dynamic load balancing to avoid a situation where some processors do most of the work. (In contrast, applications that use iterative filaments, which must statically distribute filaments among the processors in order to attain good performance.) Fork/join programs tend to employ a divide-and-conquer strategy. The computation starts on just one processor; all other processors are idle. To get all processors involved in the computation, new work (from forks) needs to be given to idle processors. Then, once all processors have work, additional load balancing may be required to keep the nodes busy. This is because when some tasks require more work than others, different nodes will receive different initial work allocations. For example, in adaptive quadrature, when integrating the function e^x, the amount of computation increases with x. The processor that is initially given the interval containing the right endpoint will do the most work.

Filaments first uses a sender-initiated load-balancing scheme and then sometimes employs a simple receiver-initiated, dynamic load-balancing policy (this can be enabled and disabled by the application programmer or compiler). Suppose that a fork/join application creates parallelism by two forks and then a join. The implementation employs a logical tree of processors (see Figure 1). The initial load-balancing phase works as follows. The first processor (root node in the tree) begins the computation; after forking the first two filaments, it gives one filament to its left child and keeps the other. Both processors continue the computation. When the root forks another two filaments, it gives one to its second child and keeps the other; the next time it gives a new filament to its third child and keeps the other; and so on. Each child processor follows the same pattern. Consequently, in each step the number of processors with work doubles. The initial phase continues in this way until a processor has given work to all of its children, after which it keeps all filaments it forks (this

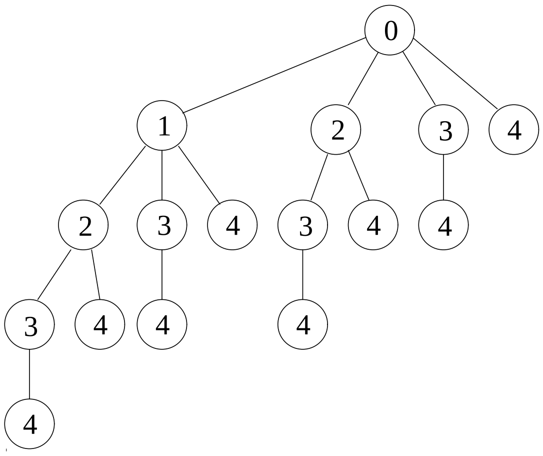

Figure 1: Logical tree of 16 processors, for applications that create parallelism by the pattern of 2 forks and a join. At each step, the number of processors with work doubles. The numbers in the figure indicates the step in which the processor first gets work.

takes $O(\log p)$ steps, where p is the number of processors).

After the initial work-distribution phase, some applications will need to employ dynamic load balancing. In Filaments this is receiver initiated; newly forked filaments are put on the tail of a processor's local list, a server thread removes work from the front of its local list, and other server threads, when idle, scan all other processors' lists and take filaments off of the list with the most filaments[4]. This works well because the largest units of work tend to be at the front of processor lists [VR88].

For applications that do not exhibit much load imbalance—such as merge sort—the cost of acquiring data outweighs the gain of load balancing, so the programmer or compiler would disable the receiver-initiated phase. On the other hand, for applications such as adaptive quadrature—where evaluating intervals can take widely varying amounts of time—dynamic load balancing is absolutely necessary.

Another optimization for fork/join filaments is *pruning*, which is analogous to implicit coarsening. When enough work has been created to keep all processors busy, forks are turned into procedure calls and joins into returns. This avoids excessive overheads due to filaments creation and synchronization. There is a danger with pruning, however. It is possible for a processor to traverse an entire subtree sequentially after other processors become idle. To avoid this, whenever another processor needs work, a processor executing sequentially returns to executing in parallel and creates new work. This is implemented by having a server thread check a flag before deciding whether to fork filaments or call them directly.

4 Applications and Performance

This section presents the performance of the Filaments package on 8 applications: matrix multiplication, Jacobi iteration, adaptive quadrature, Mandelbrot set calculation, LU decomposition, Tomcatv, Fibonacci numbers, and quicksort. Tests were conducted on a Silicon Graphics Iris 4D/340 multiprocessor, which has a 33 MHz clock, 64 Kbyte instruction cache, 64 Kbyte data

[4]This scheme suffices for small numbers of processors; a more scalable scheme is necessary for larger multiprocessors.

CPUs	1	2	4
Filaments Time	60.20	31.95	17.20
Filaments Speedup	0.99	1.86	3.45
Coarse-Grain Time	59.38	29.80	16.82
Coarse-Grain Speedup	1.00	1.99	3.53
Sequential Time	59.30		

Table 1: Matrix Multiplication, 440×440 elements (times in seconds).

cache, and 256 Kbyte secondary data cache. The `gcc` compiler, version 2.5.7, was used on all tests, with the highest possible level of optimization. For the most part, `gcc`-generated executables outperform executables produced by the vendor compiler `cc`.

For each application, we developed a sequential program, a coarse-grain program, and a fine-grain (Filaments) program. To ensure fairness and accuracy (and hence the legitimacy) of the comparison, all application code is as similar as possible. For example, the basic blocks are the same in all programs and all programs allocate memory identically. Additionally, the sequential programs were written without any parallel constructs, and the coarse-grain programs were written using only vendor subroutine libraries. Only the fine-grain programs use the Filaments package.

The performance tables in this section show execution times and speedups. The execution times are the median of at least three test runs, as reported by `gettimeofday`. These times were very consistent. The times for the fine-grain programs include the creation time for servers and filaments. The speedups are the ratio of the *sequential program* times to the other times; this explains why some single-processor tests have a speedup of less than one.

The next three subsections give details on the performance of matrix multiplication, Jacobi iteration, and adaptive quadrature in detail. Section 4.4 summarizes the performance of the other applications.

4.1 Matrix Multiplication

The execution times for matrix multiplication are shown in Table 1. The coarse-grain program creates one process per processor. Each process computes inner products for a contiguous block of the result matrix in nested `for` loops. The coarse-grain program gets speedups of 1.99 and 3.53 on 2 and 4 processors.

The fine-grain program creates and executes n^2 filaments; each filament computes a single inner product. The Filaments program is only slightly slower than the coarse-grain program, because there is sufficient work per filament (n multiplications and $n-1$ additions) to amortize the overhead of creation and execution. Furthermore, because work per filament also increases with the problem size, Filaments overhead has less of an impact on larger problems. Consequently, the Filaments program is 1.4% to 7.2% slower than the coarse-grain program. It also achieves good speedup relative to the sequential program: 1.86 and 3.45 on 2 and 4 processors, respectively. Much of the overhead in the Filaments program comes from initialization, including the creation of $193,600$ filaments.

4.2 Jacobi Iteration

Table 2 shows the performance of the Jacobi iteration programs using a 300×300 array. As in matrix multiplication, the Filaments program uses one filament per point of the solution array,

CPUs	1	2	4
Filaments Time	44.96	24.06	12.87
Filaments Speedup	0.98	1.84	3.44
Coarse-Grain Time	44.24	22.75	12.52
Coarse-Grain Speedup	1.00	1.94	3.53
Sequential Time	44.24		

Table 2: Jacobi iteration, size 300×300 (times in seconds).

CPUs	1	2	4
Filaments Time	61.92	31.70	16.91
Filaments Speedup	0.99	1.93	3.62
Coarse-Grain Time	61.34	30.91	16.30
Coarse-Grain Speedup	1.00	1.98	3.76
Sequential Time	61.28		

Table 3: Adaptive quadrature, size 35 (times in seconds).

and the coarse-grain program uses one thread per processor. This program represents a challenge for Filaments: there is very little work per filament. Unlike matrix multiplication, the work per filament is constant, independent of the problem size. However, on one node the Filaments program is only 1.6% slower than the sequential program. This is because of implicit coarsening and other optimizations in the Filament package. Both the fine- and coarse-grain programs scale well relative to the sequential program: 1.84 and 3.44 on 2 and 4 processors for the fine-grain program and 1.94 and 3.53 for the coarse-grain program.

Even though this is potentially a worst-case situation, the Filaments program performs better relative to the coarse-grain program than in matrix multiplication. This is because the problem is iterative; therefore, the Filaments package can effectively employ implicit coarsening. After the first iteration, the arguments to the filaments are generated rather than loaded from memory. Thus the fine-grain program is very similar to the coarse-grain for the second and subsequent iterations.

4.3 Adaptive Quadrature

Adaptive quadrature can lead to load imbalance if the processors (servers threads) receive unequal amounts of work, which could happen if certain intervals change more rapidly than others. In order to stress the load-balancing mechanism, we used the function $f(x) = e^x \sin x$. This function has more work at the right part of the interval because the oscillations are much sharper there.

The coarse-grain adaptive quadrature program uses the dynamic "bag-of-tasks" paradigm [CGL86, And91] (which is not trivial to implement). 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 (an interval) from the bag and approximates the area for the interval. It accepts the result if the approximation was good enough. Otherwise, it keeps one subtask and puts the other back into the bag. Eventually, some server will remove this task. The program is finished when the bag is empty and all servers are idle. Like the Filaments program, 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). The Filaments

	Seq	Filaments			Coarse-Grain		
CPUs	1	1	2	4	1	2	4
Mandelbrot	52.38	53.39 (0.98)	27.78 (1.89)	14.57 (3.60)	52.64 (1.00)	27.03 (1.94)	13.96 (3.75)
LU Decomp	45.02	46.53 (0.97)	24.87 (1.81)	14.79 (3.04)	45.36 (0.99)	23.79 (1.89)	13.80 (3.26)
Tomcatv	108.1	110.3 (0.98)	57.37 (1.88)	34.24 (3.16)	108.1 (1.00)	55.46 (1.95)	30.21 (3.58)
Fibonacci	48.08	48.40 (0.99)	24.89 (1.93)	12.85 (3.74)	48.08 (1.00)	24.41 (1.97)	12.41 (3.87)
Quicksort	46.10	46.77 (0.99)	24.89 (1.85)	13.75 (3.35)	46.10 (1.00)	27.57 (1.67)	16.32 (2.82)

Table 4: Other applications. Time given in seconds; speedup shown in parentheses.

program uses fork/join filaments as described in Section 2.3.

Table 3 gives the results for adaptive quadrature, using an interval on the x axis of 1 to 35. The sequential program uses ordinary recursion. Both the coarse- and fine-grain programs get excellent speedup. This is because both programs prune the parallelism; when pruning is disabled, the execution times increase dramatically. With problems such as Fibonacci (see next section), where each filament does very little work, pruning is vital to amortize the cost of creating, managing, and synchronizing filaments.

Although, the bag-of-tasks program gets excellent speedup for the tests shown, our experience is that as the number of processes accessing the bag increases, the performance dramatically decreases. This is because access to the bag becomes a bottleneck—even with pruning.

4.4 Other Applications

We also tested five other applications: Mandelbrot set calculation, LU decomposition, Tomcatv, Fibonacci numbers, and quicksort. The results are summarized in Table 4. Tests results for many of these applications on this platform and on a Sequent Symmetry are given in [Low93].

The Mandelbrot set is in the two-dimensional plane of complex numbers [Dew85]. The points in the Mandelbrot image take different amount of work to calculate and do not depend on any other points. Therefore, the programs can exhibit an imbalanced load and have no data locality. The coarse-grain program assigns the rows cyclically to the processors (the `i`th row goes to processor `i mod p`). This results in a relatively load-balanced implementation, because adjacent rows have nearly the same amount of work. The Filaments program creates one filament per point and also assigns rows of filaments to processors cyclicly. There is very little difference in the execution times between the fine- and coarse-grain programs—always less than one second and at most 4.4%.

LU decomposition is used to solve the linear system $Ax = b$ [PFTV88]. Decompose A into lower- and upper-triangular matrices such that $A = LU$, and the linear system becomes $A = LUx = b$. The solution, x, is obtained by solving two triangular systems $Ly = b$ and $Ux = y$, using back-substitution. For an $n \times n$ matrix, n iterations are performed. Each iteration has two phases: one to compute a pivot value, and the other to do an elimination. In this application, the work decreases by one row and column each iteration: $(n - k + 1)^2$ points are updated on the kth iteration. Thus towards the end of the computation there is very little work, limiting the potential for parallel speedup. Both the fine- and coarse-grain programs do static load balancing by assigning the rows cyclicly, yielding well-balanced programs. The Filaments program uses two pool sets, one to perform the pivot and one the elimination. Neither program gets excellent speedup on four processors due to the necessity of two barriers per iteration, and the coarse-grain program is slightly faster than the fine-grain program.

Tomcatv is an application taken from the SPEC benchmark. It is similar to Jacobi iteration;

however, it uses 7 $n \times n$ matrices (rather than 2), and has two phases per iteration. Like LU, the Filaments program creates two pool sets, one for each phase. It has 2 to 4 seconds overhead relative to the coarse-grain program, which is a 13.4% overhead on four nodes.[5]

The other two applications are 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 compares every element in the subarray to the pivot element. The poorer speedup on quicksort is due to the lengthy, sequential partitioning phase in the first few filaments that are created. Moreover, this sequential part grows with the problem size, whereas the sequential part is Fibonacci is constant. So the potential speedup of quicksort is less than that of Fibonacci or adaptive quadrature. Fibonacci stresses the pruning mechanism in Filaments because each filament does almost no work. For both applications, the coarse-grain programs use the bag-of-tasks paradigm. Both Fibonacci programs achieve good speedup, with the fine-grain program having slightly more overhead. The fine-grain quicksort program scales much better than the coarse-grain program; we believe this is because of contention for the bag.

5 Related Work

There are many existing threads packages. The initial attempts to support efficient parallelism were lightweight thread packages such as Threads [Doe87], Presto [BLL88], μSystem [BS90], μC++ [BDSY92], and Sun Lightweight Processes [SS92]. We will call these *standard* packages because they 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 support preemption to provide fairness. However, the use of stacks requires context switching code, which is inherently machine dependent. Most of the above packages also use a central ready queue.

Standard threads packages cannot support fine-grain parallelism efficiently. Consider, for example, a Jacobi iteration program written with a thread per point. Each thread performs a handful of instructions and suspends. Creating large numbers of threads (and stacks) wastes space and is inefficient. Furthermore, the work a thread does is very small, so context switching time will constitute a large overhead. Preemption adds overhead and can cause additional context switching. The use of central ready queues hurts locality and also affects cache usage. Lastly, the generality of standard packages increases their size and overhead. Filaments is more efficient for fine-grain parallelism, because it has only one stack per server thread (processor), implements inexpensive context switching, does not preempt threads, uses local ready queues, and provides optimizations such as implicit coarsening and pruning for applications that need them.

Several researchers have proposed ways to make standard thread packages more efficient. Anderson et al. [ALL89] discuss the gain from using local ready queues, and [ABLL92] shows how to do user-level scheduling. Schoenberg and Hummel [HS91] explain how to avoid allocating a stack per thread and switching contexts in nested parallel `for` loops. Markatos et al. [MB92] present a thorough study of the tradeoffs between load balancing and locality in shared memory machines. Keppel [Kep93] describes a portable threads package that supports efficient barrier synchronization and non-preemptive threads.

Many threads packages support more fine-grain parallelism. The Uniform System [TC88], built for the BBN Butterfly, has several things in common with Filaments: There are no private stacks per

[5] This is a simple application, and the Filaments program should perform well. However, for reasons we cannot explain, it has the worst performance.

thread, no context switches, and threads are not preemptable. The Uniform System's synchronous mode supports a simple form of iterative threads, and their finalization code is equivalent to our sequential code. However, iterative filaments are more powerful, and they need be created only once. The Uniform System does not support fork/join filaments. It also uses a central ready queue; in contrast, Filaments uses local ready queues, which minimizes contention and enhances data locality. The Uniform System also employs task generators (a related collection of tasks) and hence has a coarse-grain execution model. However, the Uniform System cannot efficiently support thread-per-point decompositions for problems like Jacobi, as Lin [LS90] and others have noted. Filaments can execute the fine-grain execution model efficiently in many cases, and when it needs to switch to a coarse-grain execution model, it does so at run time.

WorkCrews [VR88] supports fork/join parallelism on small-scale, shared-memory multiprocessors. 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 [CGSv93] 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 iterative parallelism by means of iterative 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 subroutine library, which is used to specify parallelism in a traditional, imperative way.

Chores [EZ93] is similar to both Filaments and 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.

Distributed Filaments [FLA94] implemented the fine-grain model on a distributed-memory machine. [Shu95] and [NS95] describe later systems that allow the efficient management of many threads on distributed-memory machines.

6 Conclusion

This paper has shown that fine-grain parallelism provides a simple programming model for parallel applications and has presented an efficient implementation for shared-memory machines. The implementation uses the Filaments package, which is written entirely in software and has been ported to several machines. The main techniques that make Filaments efficient are stateless threads, implicit coarsening of iterative filaments, and pruning and load balancing of fork/join filaments.

The Filaments package can be used as a stand-alone library, as here, or it can be used as a compiler target. We have also developed an efficient implementation of Filaments for distributed-memory machines, including clusters of workstations [FLA94, LFA96]. The distributed implementation employs additional techniques for overlapping communication and computation and for adaptive data placement. We are continuing to work on the project to make the approach even more useful, portable, and efficient. In particular, we are working to implement the package on a network of multiprocessors and simultaneously exploit both shared- and distributed-memory parallelism.

Acknowledgements

Dawson Engler implemented the original Filaments package on the Sequent and Iris, and David Koski implemented the current package.

References

[ABLL92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *acmtocs*, 10(1):53–79, February 1992.

[ALL89] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, California, 1991.

[BDSY92] Peter A. Buhr, Glen Ditchfield, R.A. Stroobosscher, and B.M. Younger. uC++: concurrency in the object oriented language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992.

[BLL88] B.N. Bershad, E.D. Lazowska, and H.M. Levy. PRESTO: a system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[BS90] Peter A. Buhr and R.A. Stroobosscher. The uSystem: providing light-weight concurrency on shared memory multiprocessor computers running UNIX. *Software Practice and Experience*, pages 929–964, September 1990.

[CGL86] Nicholas Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in Linda. In *Thirteenth ACM Symp. on Principles of Programming Languages*, pages 236–242, January 1986.

[CGSv93] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM—a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, August 1993.

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

[Doe87] Thomas W. Doeppner. Threads: a system for the support of concurrent programming. Technical Report CS-87-11, Brown University, June 1987.

[EZ93] Derek L. Eager and John Zahorjan. Chores: Enhanced run-time support for shared memory parallel computing. *ACM Transactions on Computer Systems*, 11(1):1–32, February 1993.

[FLA94] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201–212, November 1994.

[Fre95] Vincent W. Freeh. Writer-Owns: a new page consistency protocol for dynamically controlling thrashing on distributed-shared memory systems. December 1995.

[HS91] S.F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journel of Research and Development*, 35(5):743–765, September 1991.

[Kep93] David Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[LA96] David K. Lowenthal and Gregory R. Andrews. An adaptive approach to data placement. In *Proceedings of the 10th International Symposium on Parallel Processing*, April 1996.

[LFA96] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. TR 96-2, The University of Arizona, January 1996.

[Low93] David K. Lowenthal. Performance experiments for the Filaments package. TR 93-13, University of Arizona, September 1993.

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

[MB92] Evangelos P. Markatos and Thomas L. Blanc. Load balancing vs. locality management in shared-memory multiprocessor. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume I, Architecture, pages I:258–267, Boca Raton, Florida, August 1992. CRC Press.

[NS95] Richard Neves and Robert B. Schnabel. Runtime support for execution of fine grain parallel code on coarse-grain multiprocessors. In *Fifth Symposium on the Frontiers of Massively Parallel Computing*, February 1995.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambrigde, 1988.

[Shu95] Wei Shu. Runtime support for user-level ultra lightweight threads on massively parallel distributed memory machines. In *Fifth Symposium on the Frontiers of Massively Parallel Computing*, February 1995.

[SS92] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX 1992*, June 1992.

[TC88] Robert H. Thomas and Will Crowther. The Uniform system: an approach to runtime support for large scale shared memory parallel processors. In *1988 Conference on Parallel Processing*, pages 245–254, August 1988.

[VR88] M. Vandevoorde and E. Roberts. Workcrews: an abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.