HyFi: Architecture-Independent Parallelism on Networks of Multiprocessors

David K. Lowenthal
Department of Computer Science
University of Georgia
dkl@cs.uga.edu

Ragavan Subramanian
Department of Computer Science
University of Georgia
ragavan@cs.uga.edu

July 17, 2002

Abstract

A network of parallel workstations promises cost-effective parallel computing. This paper presents the *HyFi* (Hybrid Filaments) package, which can be used to create architecture-independent parallel programs—that is, programs that are portable *and* efficient across different parallel machines. HyFi integrates Shared Filaments (SF), which provides parallelism on shared-memory multiprocessors, and Distributed Filaments (DF), which extracts parallelism from networks of uniprocessors. This enables parallelism on any architecture, including homogeneous networks of multiprocessors.

HyFi uses fine-grain parallelism and implicit shared-variable communication to provide a uniform programming model. HyFi adopts the same basic execution model as SF and DF; this paper discusses the modifications necessary to develop the hybrid system. In particular, HyFi modifies the signal-thread model as well as the software distributed shared memory of DF. It also unifies the SF and DF reduction operations as well as the dynamic load-balancing mechanism of fork-join filaments. Application programs written using the HyFi API can run *unchanged* on *any* architecture. Performance is encouraging on fork/join applications, where excellent speedup is achieved. Also, the fine-grain model of HyFi allows up to a 14.5% improvement due to overlap of communication and computation. Unfortunately, we find that iterative applications do not speed up well due to the inability of the Pentium Xeon architecture to efficiently support concurrent memory accesses.

Keywords: fine-grain parallelism, architecture independence, distributed shared memory.

1 Introduction

Recently, fast networks and small-scale, bus-based shared-memory multiprocessors have seen a decrease in cost and hence an increase in availability. This has made networks of multiprocessors an increasingly desirable platform for parallel computing. This kind of “hybrid” parallel machine uses hardware shared memory within a single node, but requires messages to be exchanged to access the memory of a different node. Such a parallel machine can be more cost effective than a network of uniprocessors or a single multiprocessor; in other words, if one wishes to construct a parallel machine with 16 processors, the best price/performance is probably using eight two-way multiprocessors or four four-way multiprocessors.

Unfortunately, programming such a machine efficiently is difficult, because a network of multiprocessors is not a shared- or a distributed-memory machine. It can be programmed as a distributed-memory machine, but that is difficult and may result in a loss of efficiency within a node. It cannot be programmed (without software support) as a shared-memory machine because the nodes have independent memories.

We have developed a substrate to allow programs to be written (or generated by a compiler) for networks of multiprocessors using a fine-grain, shared-memory style. This software library, called HyFi (Hybrid Filaments), uses fine-grain parallelism to allow parallel programs to be written using its natural degree of parallelism. It is implemented efficiently and allows better overlap of communication. HyFi also provides implicit shared variable communication, which allows a uniform memory model on hybrid machines. HyFi builds builds on the Filaments package [], which provided the same model to the programmer or compiler for shared- *or* distributed-memory machines, but *not* a combination of them. Accordingly, the HyFi kernel has been rewritten to allow a single, unified seamless package.

In a software distributed shared memory (SDSM) system, each processor can access any data item, without the programmer having to worry about where the data is or how to obtain its value, whereas in the native explicit message passing model, the programmer has to decide *when* a processor needs to communicate, with *whom* to communicate, and *what* to communicate.

The current implementation of HyFi supports the SPARC and X86 architectures. Performance results for HyFi has been obtained on a network of four quad-pentium Xeons, each running Solaris. HyFi extracts parallelism from all possible combinations of nodes and processors and achieves good speedup on fork/join applications but disappointing speedup on iterative applications. The latter is due to poor support for shared-memory multiprocessing in the quad-Pentium Xeon architecture. HyFi does, however, exhibit good promise for achieving good scalability by providing automatic overlap of communication and computation; in Jacobi iteration, this accounts for up to a 14.5% improvement.

The paper is organized as follows. Section 2 discusses how architecture independent user programs are written using HyFi. Section 3 discusses the implementation of HyFi, including the extensions necessary from both SF and DF. Section 4 provides the performance results using HyFi for a range of benchmarks. Section 5 discusses related work, and Section 6 concludes.

2 Architecture-Independent Parallel Programming using HyFi

HyFi 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 and LU decomposition. *Fork-join* filaments recursively create new filaments and wait for them to return results. They are used in divide-and-conquer applications such as adaptive quadrature and recursive fibonacci.

A program that uses HyFi contains three additional components relative to a sequential program: (1) declarations of variables that are to be located in shared memory, (2) functions containing the code for each filament, and (3) a section that creates the filaments, places them on processors and nodes, and controls their execution.

Iterative filaments execute in *phases*. All filaments in a phase are independent and can execute in any order. For every phase in the program, each processor on the node has one or more *pools*

```
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 (filGetVal(maxdiff) < temp)
    filGetVal(maxdiff) = temp;
}
```

```
int check() {
  filReduce(maxdiff, MAX);
  if (filGetVal(maxdiff) < epsilon)
    return F_DONE;
  swap(old, new);
  filGetVal(maxdiff) = 0.0;
  return F_CONTINUE;
}
```

Figure 1: Code for each filament (left) and post-phase code (right) for Jacobi iteration.

associated with that phase. Pools are groups of filaments that reference data with spatial locality. Using multiple pools per processor provides for better efficiency in the distributed shared memory by overlapping computation with communication; however, it is not necessary for correctness. A phase consists of a pointer to the filament code and a pointer to the *post-phase* code, which is a function that is called after each execution of all filaments in the phase. The post-phase code synchronizes the processors in all the nodes and determines whether a phase is finished.

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

Below we describe how iterative applications are programmed using HyFi and Jacobi iteration as an example. The names of HyFi library calls have been shortened for brevity. Additionally, for clarity, some details of the code fragments are omitted.

Jacobi iteration is a finite differencing scheme that can be used for many different applications; it works as follows. Discretize a 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 using 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 by n arrays, `new` and `old`, and the *reduction* variable `maxdiff`. A reduction variable is a special kind of variable with one copy per processor. The `filGetVal` macro returns the value of the reduction variable on the processor that is running the filament. Reduction variables are also used in calls to `filReduce`, which (1) atomically combines the private copy on each processor on every node into a single copy using a binary, associative operator (such as add or maximum) and (2) copies the reduced value into each private copy. A call of `filReduce` implicitly results in a barrier synchronization between nodes.

The code executed by each filament, shown in the left half of Figure 1, computes an average and difference: 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 observed thus far on this iteration of the entire computation, then `maxdiff` needs to be updated.

After all grid points are updated, procedure `check` is called to check for convergence and to swap grids. One processor on each node executes this code at the end of every update phase, i.e., after every filament in the phase has been executed once. The next iteration is performed only if `check` returns `F_CONTINUE`. The code is shown in the right half of Figure 1.

The initialization section is executed on each node, because each address space must be initialized. The call of `filCreatePhase` creates a phase, which contains filaments that execute the function `jacobi`. The other argument to `filCreatePhase` is a pointer to the post-phase code. The `filCreatePoolArray` call creates an array of pools for the specified phase, one per processor, where each pool contains space for `numFilPerProcessor` filaments. The `filCreateFilament` routine creates a single filament. Each is defined by `i` and `j`, which are passed as arguments to `jacobi`. Variables `procId`, `startRow`, and `endRow` are discussed below. The final Filaments package call, `filStart`, starts the parallelism. All previously created phases are completed before `filStart` returns.

An iterative program must distribute the filaments among the processors and nodes in order to balance the load. The initialization code above shows a general way to distribute filaments for networks of multiprocessors. The values `startRow` and `endRow` are the start and end of the the block assigned to the node. A filament is assigned to the pool of the processor specified by `procId`. This simple method of assigning filaments to pools can support all regular distributions on hybrid

```
void main() {
  numRowsPerNode = n/filNumNodes();
  numRowsPerProcessor = numRowsPerNode/filNumProcessorsPerNode();
  numFilPerProcessor = numRowsPerProcessor * n; /* n Filaments per row */

  phase = filCreatePhase(jacobi, check);
  poolArray = filCreatePoolArray(phase, numFilPerProcessor);

  startRow = filMyNode() * numRowsPerNode; /* startRow and endRow
  endRow = startRow + numRowsPerNode - 1;  /* partition work between nodes */

  for (i = startRow, currentRow = 0; i <= endRow; i++) {
    /* determine which processor to use for this row */
    procId = currentRow++ / numRowsPerProcessor;
    for (j = 1; j < n-1; j++)
      filCreateFilament(poolArray[procId], i, j);
  }
  filStart();
}
```

Figure 2: Code for the creation of filaments.

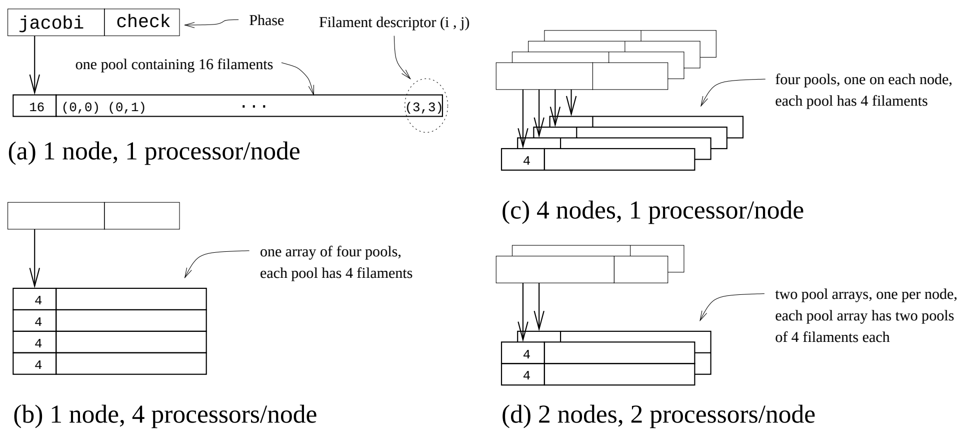
jacobi
check
Phase
Filament descriptor (i , j)
one pool containing 16 filaments
16 (0,0) (0,1) ... (3,3)
(a) 1 node, 1 processor/node
one array of four pools, each pool has 4 filaments
4
4
4
4
(b) 1 node, 4 processors/node
four pools, one on each node, each pool has 4 filaments
4
(c) 4 nodes, 1 processor/node
two pool arrays, one per node, each pool array has two pools of 4 filaments each
4
4
(d) 2 nodes, 2 processors/node

Figure 3: Distributing 16 filaments in pools on various machines.

machines. More importantly, it makes the code architecture independent; no change is needed when porting the code to different processor and node configurations. Additional information on automatic filament assignment for regular and irregular programs can be found in [ML01].

Figure 3 illustrates ways to distribute filaments between processors and nodes using phases and pools on several different architectures. The example shows the block distribution of a 4×4 matrix. It uses one phase and one pool per processor. The phase contains the filament code pointer (`jacobi`) and the post-phase code pointer (`check`). Figure 3(a) shows the pool on a uniprocessor. Since there is only one processor, there is only one pool containing all 16 filament descriptors. In the code fragment above that shows initialization, `startrow` and `endrow` would be 0 and 3, and `procId` would always be 0. Figure 3(b) shows the pools on a four-way multiprocessor. There is an array of four pools, one per processor, where each pool contains four filament descriptors; `startrow` and `endrow` would again be 0 and 3, but `procId` now will vary between 0 and 3 because `filNumProcessors()` is 4. Figure 3(c) shows the pools on a multicomputer with four nodes. There are four pools, one per node; each pool contains four filament descriptors. Here `filNumNodes()` is 4 and `filNumProcessors()` is 1. Therefore, on each node, `startrow` equals `endrow` which equals `filMyNode()`. On all nodes, `procId` is always 0. Figure 3(d) shows the pools on a hybrid machine with two nodes and two processors per node; `filNumNodes()` and `filNumProcessors()` are both 2, which results in values for `startrow` and `endrow` of 0 and 1 on one node and 2 and 3 on the other. Variable `procId` varies between 0 and 1 on both nodes. There are two arrays of two pools each, one on each node.

3 Implementation

The Hybrid Filaments (HyFi) package is currently implemented using POSIX threads on Solaris; with support for the SPARC and X86 architectures. The package is implemented entirely in software. It integrates and modifies the components of Filaments that provide shared- and distributed-memory support to efficiently support fine-grain parallelism on a symmetric network of multiprocessors.

The HyFi implementation derives certain aspects from the Shared Filaments (SF) implementation (such as locks) and others from from the Distributed Filaments (DF) implementation (such as the message substrate). It uses a modified version of the DF Software Distributed Shared Memory (SDSM). It also changes the signal-thread model and combines the SF and DF versions of reduction operations. It performs a hybrid stealing of filaments to dynamically balance load in fork-join applications.

This section discusses the key aspects of the implementation of HyFi that are significantly different from those of Filaments. The following subsection examines the basic execution model of HyFi. Section 3.2 discusses the the signal-thread model, and Section 3.3 describes the changes made to the SDSM. Section 3.4 contains a description of the changes to reduction variables, and the last section describes the changes in dynamic load balancing.

3.1 Basic Execution Model

HyFi uses a two-tiered threads model, with server threads (implemented by POSIX threads) on the first tier and user level threads called *filthreads* (implemented by HyFi) on the second tier. The chosen model has two purposes:

- Server threads can be bound to physical processors on the machine to ensure that the operating system does not schedule multiple server threads on the same processor (inhibiting parallelism).

- Filthreads allow for overlap of communication and computation in the SDSM and give us

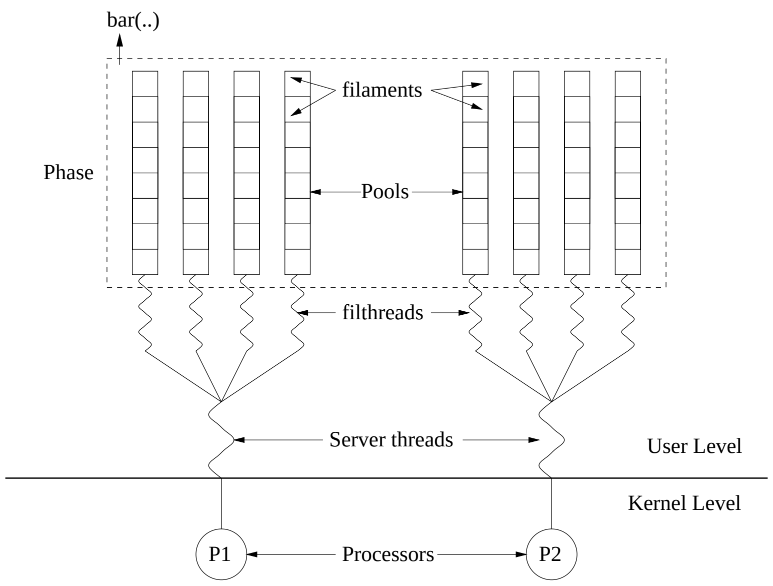

Figure 4: Basic Execution Model

control over scheduling—we control the exact order of execution (of filthreads).

Note that SF only requires server threads (because there is no internode communication) and DF only requires filthreads (because there is no need to bind threads if each node is a uniprocessor). HyFi needs both mechanisms.

There are as many filthreads as there are pools of filaments in the current phase. Filthreads are multiplexed onto the server threads. Server threads switch between filthreads only when a filthread blocks as the result of a page fault; filthreads are never preempted. The action of switching between filthreads enables overlapping of computation and communication in the SDSM.

Figure 4 shows the basic execution model on a single node with two processors. There are two server threads each bound to a processor on the machine. There are four pools of filaments (per-processor) and therefore four filthreads (per-processor). The four filthreads are multiplexed on the server thread. The server threads execute the filthreads one by one and context switch between their filthreads only if any of them page faults. Filthreads execute filaments in their pool one by one, i.e., they call the function (in the example, this is `bar(..)`), with the proper parameters, that each filament is executes.

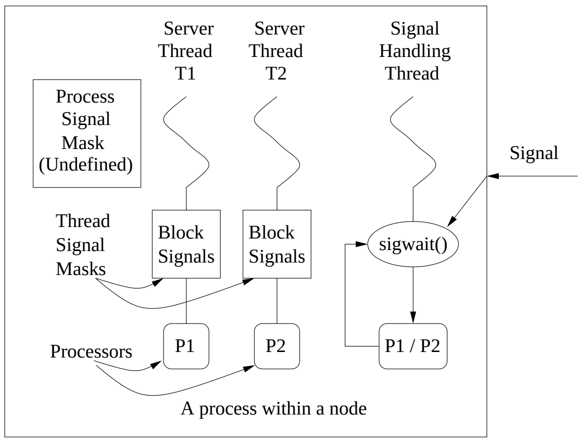

Figure 5: HyFi signal-thread model

3.2 Signal-Thread Model

In Hybrid Filaments there are as many active threads as there are processors on the node. Because all messages (such as page requests and replies, load balancing requests and replies, and other synchronization) are received through the UNIX signal interface (specifically, `SIGIO`), this necessitates a signal-thread model that can handle concurrent threads *and* signals.

The primary problem with concurrent threads and signals is that when a signal occurs, any one of the threads could execute the signal handler. In addition, that thread could hold a lock (implicitly or explicitly) a the time the signal arrives. Then, the signal code could try to acquire the same lock, which would lead to deadlock. This can happen with `malloc()`, for example, as well as any function that is not guaranteed to work in the face of asynchronous signals (called async-signal-unsafe functions in UNIX). Furthermore, trying to assign one thread only using process signal masking (e.g. `sigprocmask`) around critical sections also does not work, because these functions are undefined when using POSIX threads and would be inefficient even if they worked properly.

In the HyFi implementation, server threads block out all asynchronous signals that could be received and therefore can never be selected to process an incoming signal. A separate thread is created to handle all signal requests. This signal-handling thread loops, executing a `sigwait` on the selected signals. When a signal arrives, the signal-handling thread returns from `sigwait`, calls the appropriate handler and then loops back to do a `sigwait`. We do not bind the signal-handling thread, allowing the operating system to schedule it on any processor. (We do, however, bind the server threads to different processors on the node; this ensures that the operating system does not schedule different server threads on the same processor, which would inhibit parallelism.) The HyFi signal-thread model is shown in Figure 5.

Note that signal handling is not a problem with DF, because there is only one thread of control on a node. This means that we know the executing thread will handle signal events.

3.3 Multithreaded SDSM

A software distributed shared memory (SDSM) is a virtual-memory system extension that provides the abstraction of shared-memory on a distributed-memory machine. When a nonlocal memory access is made, a fault is generated and the page is requested; the owner of the page supplies the faulting node with the page. The HyFi SDSM is multithreaded, meaning that multiple page faults can be outstanding; other work can be done by a filthread while the remote fault is pending. Because there are multiple processors, there are multiple filthreads active at a given point in time. In particular, multiple filthreads can fault on the *same* page.

The synchronization to protect concurrent access to the page table is straightforward; the page table is protected with a per page lock to prevent simultaneous access by (1) different threads concurrently faulting on the same page, and (2) the signal-handling thread, which may be receiving a particular page at a time when a server thread is faulting on it. The per-page granularity of the lock maintains maximum concurrency in the SDSM.

There are also nontrivial aspects, including (1) avoiding excess page faults and (2) knowing when all work on a node is done. DF uses a single counter that keeps track of the number of outstanding requests on a node is used to ascertain whether all the work on the node is completed for that particular phase. However, in HyFi, we must make sure that all work on each processor on a node is finished in order to determine that work for the whole node is done. This means that in HyFi, we must make the outstanding requests counter is a per-processor data structure. The *pending* field of the page table is the data structure that indicates whether we should modify the outstanding requests counter; HyFi uses a per-processor data structure as well, for reasons described below.

On a page fault, a page is requested from a remote node only if the page is not pending. The page could be pending if any thread, running on any processor on the node, has previously faulted on the same page. Determining whether to request a page is implemented by checking the pending field of every processor in the page table of the faulting page. If one is set, the page is pending and not re-requested; otherwise, the page request is sent.

To determine if the outstanding requests counter should be incremented, we check the pending field of the processor on which the faulting thread is running. If it is set, another filthread (that was running on the same processor) is waiting for the same page, and when the page arrives both the filthreads will be awakened. In this case, we do not increment the outstanding requests counter for the processor. If the pending field of the processor in not set then it is set and the outstanding requests counter for that processor is incremented.

When a page arrives, the page table for that page is examined and the outstanding requests counter is decremented for all processors for which the pending field is set (because all filthreads waiting for the page will be awakened). The pending fields of all processors are then reset.

3.4 Reduction Variables

As described in Section 2, a *reduction* is a special kind of variable that has one copy per processor. A HyFi reduction is implemented by combining the two different types of Filaments reductions (shared- and distributed-memory reductions). The reduction data structure is an array of P elements, where P is the number of processors. Each node first performs a shared-memory reduction by assigning a master processor to reduce all values on that node. Then, a distributed-memory reduction is performed between all nodes using a tournament scheme [HFM88].

3.5 Load Balancing

Fork-join filaments are used to create filaments dynamically and later wait for them to terminate and return results. As mentioned in Section 2, workload and locality are generally unpredictable. This means that processors can receive differing amounts of work; to prevent idling nodes, we need to implement load balancing.

In fork-join filaments, work has to be distributed initially to every processor on each node until

Program	*Work Load*	*Data Sharing*	*Synch.*	*Computation/Thread*
Matrix Multiplication	Static	Light	None	Heavy
Jacobi Iteration	Static	Medium	Medium	Light
LU Decomposition	Static	Heavy	Heavy	Medium
Adaptive Quadrature	Dynamic	None	Medium	Medium
Fibonacci	Dynamic	None	Medium	Light

Table 1: Application kernels and their characteristics.

all have work. HyFi uses an efficient implementation that employs a logical tree of processors. Each element in the tree (a processor) is mapped to a particular processor on a particular node. During startup, filaments are created specifically to be sent to a particular processor on a particular node, as determined by the mapping.

After the initial work-distribution phase, some applications will need to employ dynamic load balancing. In HyFi, this is receiver initiated. Newly forked filaments are put on the tail of a processor's local-list. When a processor asks for work, it first requests filaments from processors on the same node, and then if necessary, from processors on a different node.

Many different strategies were considered for inter-node stealing. The basic idea behind the currently implemented strategy is to equalize load between the requesting node and the donor node. On receiving a request, the donor node gives the requesting node half the number of filaments from the total of all its ready lists. Filaments are one by one taken off the ready list of every processor on the donor node in a round robin fashion until the required number is met. Those filaments are then sent to the requestor. On receipt, filaments are unpacked at the receiving node and distributed equally among all processors on the node.

4 Performance

This section reports the performance of five applications: Jacobi iteration, matrix multiplication, LU decomposition, adaptive quadrature, and recursive fibonacci. The first three applications have a statically determinable workload and use iterative filaments, whereas the last two have dynamic workload and use fork-join filaments for parallelism. The main emphasis of this section is to show that HyFi can extract parallelism from any architecture: shared, distributed, or hybrid.

All tests were run on a network of four 550 MHz quad-Pentium Xeon nodes connected by a fast ethernet. All nodes were running Solaris 2.8, and all programs were compiled with `gcc` using the `-O2` flag. The application programs were run on 1, 2, 4, 8, and 16 processors using the various available combinations of nodes and processors. The configurations used were 1×1, 1×2, 2×1, 2×2, 2×4, 4×1, 4×2, and 4×4; where $n \times p$ means n nodes and p processors per node. All speedups were calculated using the HyFi 1×1 program as the baseline[1]. We chose sizes so that the execution time of the single processor programs was between 2 and 3 minutes.

Note that HyFi does incur a small overhead relative to Shared Filaments (SF) or Distributed Filaments (DF) on configurations on which the latter systems can execute ($n \times 1$ or $1 \times p$, respectively). This is due for example to locking and extra conditionals at certain points in HyFi that are not necessary in SF or DF. However, this overhead is extremely small (never more than 1-2%). We feel this is acceptable given that HyFi is can support $n \times p$ configurations for arbitrary values of n and p.

Each test was run at least three times, and the reported result is the median. The `gettimeofday` function was used to time the tests. Considerable effort was taken to make sure that no other users were using the system when tests were run, and the only active processes were Unix daemons. In practice, we found test results to be very consistent—the other UNIX daemon processes did not interfere.

Below is the performance of the five programs; their characteristics are summarized in Table 1.

4.1 Jacobi Iteration

This section examines the performance of the Jacobi iteration application discussed in Section 2. In order to update any point in the matrix in Jacobi iteration, we need to access points in rows above and below the current row. This leads to a *nearest-neighbor sharing pattern* which can be further sped up by overlapping communication of the border rows with computation of the interior rows; multiple pools per processor were therefore used in the implementation.

A filament is created for each point in the matrix and assigned to the appropriate pool. Jacobi

[1]We intended to use pure sequential versions of the programs as the baseline but did not because the compiler generates sequential code that runs *slower* than HyFi versions.

Configurations	1x1	1x2	1x4	2x1	2x2	2x4	4x1	4x2	4x4
Jacobi Time	125	73.8	82.8	73.5	48.5	40.8	44.0	32.1	31.6
Jacobi Speedup	1.00	1.69	1.50	1.69	2.58	3.06	2.84	3.89	3.95
LU Time	152	92.7	66.5	88.0	58.7	46.3	50.5	36.0	31.1
LU Speedup	1.00	1.64	2.28	1.72	2.59	3.28	3.01	4.22	4.88
Matrix Mult. Time	193	113	87.0	116	76.1	66.2	61.4	42.4	36.4
Matrix Mult. Speedup	1.00	1.71	2.21	1.66	2.54	2.91	3.14	4.55	5.30
Quad Time	128	62.4	32.1	62.4	32.2	16.0	31.1	16.1	8.30
Quad Speedup	1.00	2.05	3.98	2.05	3.98	8.00	4.11	7.95	15.4
Fib Time	145	72.4	36.4	72.4	36.2	18.4	36.2	18.3	9.43
Fib Speedup	1.00	2.00	3.98	2.00	4.00	7.88	4.00	7.92	15.3

Table 2: Performance results for our five applications in seconds. Jacobi iteration uses a problem size of 1024×1024 and iterates 500 times. LU decomposition and matrix multiplication use a size of 1800×1800. Fibonacci computes the 46^{th} fibonacci number. Quad computes the area under $f(x) = x^6$ between 1 and 70.

Configurations	2x1	2x2	4x1	4x2
Time/iteration, no overlap (msec)	67.3	42.0	36.2	24.2
Time/iteration, with overlap (msec)	63.8	38.5	32.7	20.7
Improvement from overlapping	5.2%	8.3%	9.6%	14.5%

Table 3: Improvement from overlapping communication and computation for Jacobi iteration.

iteration has a static workload and a reduction operation between iterations. As can be seen in Table 2, the speedups obtained using HyFi are disappointing. In particular, the shared-memory improvements (especially with 4 processors) either get virtually no speedup or even slow down (in the case of the 1×4).

We examined this problem in depth and were able to determine the root cause. Iterative filaments programs such as Jacobi iteration (as well as LU decomposition and matrix multiplication) are often array intensive. The versions of these three programs that we used spend the majority of their computation accessing arrays. The arrays are too large to fit entirely in cache; hence, on each iteration, the entire array streams through the cache. All processors therefore access main memory for a significant portion of the time, leading to contention. The quad Pentiums we used (Xeons) do not exhibit good performance in this case. Separate benchmarks showed near-perfect speedup when using smaller array sizes (that fit entirely in cache) as well as lack of any speedup on 4 processors for a synthetic program that performed only (unique) memory accesses. We also made a best effort to improve the memory bandwidth by improving the memory system interleaving and re-executing all experiments[2]. Still, memory-intensive programs on this particular architecture will not speed up well within a node. This is not a problem with HyFi; coarse-grain parallel programs that we wrote to use one process per processor exhibit the *same* poor speedup. The general memory contention problem is well known, but we were surprised to encounter it on just 4 processors.

As a result, increasing the number of nodes is *more* effective than increasing the number of processors, a nonintuitive result. This is because adding a node does not increase memory contention. It does cause communication via page faults, which does not occur when increasing the number of processors, but the performance is still superior.

One key advantage to using fine-grain parallelism is potential for overlap of communication and computation (see Table 3). For 2 and 4 nodes (with both 1 and 2 processors per node), it shows the times with and without overlapping. (We did not experiment with 4 processors per node because that test did not achieve any parallel speedup at all, as described above.) Overlapping provides up to a 14.5% improvement on the 4×2 case. The improvement increases with the number of processors and nodes, because the work per processor and node decreases, making communication relatively more expensive. Hence, achieving overlap is important to achieve good scalability.

4.2 LU Decomposition

LU decomposition calculates the lower and upper triangular matrices L and U of a matrix A, where $A = L{\cdot}U$. For a $n{\times}n$ matrix, n iterations are performed. In this application, the work decreases by one row and column in each iteration: $(n - k + 1)^2$ points are updated in the k^{th} iteration. Consequently, towards the end of the computation there is very little work, limiting the potential for speedup. HyFi uses a cyclic distribution of rows between nodes and processors, yielding a program with well-balanced load.

The LU program uses two phases, one to perform a pivot operation and the other to perform the elimination operation. A single filament is created for each row of the input matrix for each phase. The execution times and speedups for LU are shown in Table 2. The shared-memory speedup is

[2]The original configuration had 512M of main memory per node divided into 4 128M increments. We replaced this with 16 32M increments, which filled all the banks.

somewhat better than in Jacobi because the computation is slightly less array intensive. However, the speedup is somewhat restrained by the necessity of two barriers per iteration (one after each of the pivot and elimination phases), and also because of the decreasing workload of the application.

4.3 Matrix Multiplication

Matrix multiplication (Matmult) is used in many numerical calculations and is a prime candidate for parallelization. The matrix multiplication program used to test HyFi calculates $C = A \times B$, where A, B, and C are $n \times n$ matrices. Each point in the result matrix C can be calculated in parallel, and the workload is uniform. We create n^2 filaments, one for each point in the C matrix. The filaments are assigned to the nodes and processors using a regular *block* distribution. In order to calculate a point $C(i, j)$, we need the i^{th} row of A and the j^{th} column of B; consequently, every nodes needs *read* copies of the entire B matrix and the corresponding rows of the A matrix that the node is calculating. A master node initializes the A and B matrices and the other nodes get data transparently via the SDSM when filaments fault and request pages containing that data. The execution times and speedups for Matmult are shown in Table 2. Matmult achieves better speedup than Jacobi and LU due to several factors. The work per filament is much larger in Matmult, and there is no sharing after the data is initially distributed. Also, there are no barrier synchronization (other than at the end of the program), because this application is not iterative. The main overhead is in acquiring the matrices (part of A and C as well as all of B); the overhead due to initial distribution cannot be amortized over several iterations as it can in Jacobi iteration and LU decomposition.

4.4 Adaptive Quadrature

Adaptive quadrature (Quad) is an algorithm to compute the area under the curve defined by a continuous function. It works by dividing the an interval in half, approximating the areas of both halves, and then subdividing further if the approximation is not good enough. The program tested evaluates the area under the curve $f(x) = x^{2w}$ from point a to point b.

The natural algorithm for this problem uses divide-and-conquer, so our implementation uses fork-join filaments. The program also uses an optimization for fork-join filaments called *pruning.*

When enough work is created to keep all processors busy, forks are turned into procedure calls and joins into returns. This avoids excessive overheads due to filament creation and synchronization. The execution times and speedups for Quad are shown in Table 2. Excellent speedup (superlinear in some cases) is obtained with increasing number of processors because of the dynamic load balancing (stealing filaments) and pruning features of the package. Also, there is no data sharing between processors as well as significant computation per subproblem.

4.5 Recursive Fibonacci

The Fibonacci value of an integer n can be defined inductively as follows:

```
Fib(n) = Fib(n-1) + Fib(n-2), where Fib(1) = Fib(2) = 1
```

Though there is a simple iterative solution, a recursive version of is a good benchmark to test parallel systems that support fine-grain parallelism. Fibonacci is different from adaptive quadrature because each call takes little time. Because of this, Fibonacci stresses the pruning mechanism as well as the dynamic load balancing feature in the package. The execution times and speedups for Fibonacci are shown in Table 2. The speedup obtained is almost linear with the number of processors used.

5 Related Work

The four systems most similar to ours are an OpenMP implementation on a network of multiprocessors [HLCZ99], Strings [RC98], SoftFLASH [ENCH96], and MGS [YKA96]. The OpenMP port is [HLCZ99] is probably the most similar to HyFi. Similar to HyFi, it provides a single API for parallelization within a node, between nodes and a combination of the two. It also uses POSIX threads within a node for portability, and a per-page-mutex for the page table to provide maximum concurrency. The key difference between HyFi and OpenMP is that OpenMP supports efficiently only the iterative model of parallel execution whereas HyFi supports both the iterative *as well as* fork-join model of computation.

Strings [RC98] is a high performance SDSM for symmetric multiprocessor clusters. It is based on the Quarks SDSM [SSC98]. It also uses POSIX threads and is implemented entirely in software and uses a dedicated thread to handle signals. The primary difference between HyFi and Strings is that

HyFi supports fork/join computation and allows for overlapping communication and computation within a single thread.

The SoftFlash [ENCH96] system is a kernel DSM implementation. SoftFlash optimizes sending and receiving pages from one node to another by eliminating memory-to-memory copies within a node. In contrast, HyFi is a user level implementation and relatively more portable. Like HyFi, the SoftFLASH SDSM is transparent to the user application. The SoftFLASH system configuration dedicates additional processors for network interrupt processing. These processors do not run applications and are not counted when assessing speed up. Unlike HyFi, SoftFLASH supports only iterative programs and does not provide support for true fork-join parallel applications.

MGS [YKA96] was one of the first systems to explore coupling of small to medium scale shared memory multiprocessors through software to synthesize larger logically shared memory systems. Similar to Strings and the OpenMP port, it handles iterative applications well but does not support fork-join programs.

There are other systems that bear some resemblance to HyFi, except that they use processes to implement parallelism within a node. Each of these processes have a separate address space but have shared memory regions mapped between the processes. These systems include Cashmere-2L [SDH$^+$97] and HLRC-SMP [SBIS98]. Our implementation uses a single process per node and uses POSIX threads for parallelism within a node. This enables the automatic use of hardware shared memory within a node for better efficiency.

There is a wealth of related work on fine-grain parallelism and software distributed shared memory systems; two representative papers that cover these topics are [TTY99] and [Ift98]. We cannot further cover research in these areas due to space limitations.

Finally, as described in previous sections, HyFi is a combination of Shared Filaments [LFA98] and Distributed Filaments [LFA96].

6 Conclusion

This paper has presented a library package called HyFi, which provides a substrate for programming or generating code for a network of multiprocessors. This is accomplished by efficiently implementing fine-grain parallelism and implicit shared variable communication. One of the distinguishing features of HyFi is its support for *both* the iterative and fork-join models of parallelism. Performance showed that execution times for HyFi were excellent for fork-join applications and somewhat disappointing for iterative applications. The primary problem with the latter was found to be that the Pentium Xeon architecture cannot handle efficiently memory-intensive programs, even on a small scale.

References

[ENCH96] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct 1996.

[HFM88] D. Hansgen, R. Finkel, and U. Manber. Two algorithms for barier synchronization. *Int. Journal of Parallel Programming*, 17(1):1–18, February 1988.

[HLCZ99] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. In *Proceedings of the Thirteenth International Parallel Processing Symposium*, pages 28–31, Apr 1999.

[Ift98] Liviu Iftode. *Home-Based Shared Virtual Memory.* PhD thesis, Princeton University, June 1998.

[LFA96] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, 37:41–54, November 1996.

[LFA98] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Efficient support for fine-grain parallelism on shared-memory machines. *Concurrency: Practice and Experience*, 10(3):157–173, March 1998.

[ML01] Donald G. Morris and David K. Lowenthal. Accurately computing redistribution cost in distributed shared memory systems. In *Principles and Practice of Parallel Programming*, pages 62–71, June 2001.

[RC98] Sumit Roy and Vipin Chaudhary. Strings: A high-performance distributed shared memory for symmetrical multiprocessor clusters. In *Proceedings of HPDC*, July 1998.

[SBIS98] R. Samanta, A. Bilas, L. Iftode, and J. Singh. Home-based SVM protocols for SMP clusters: design and performance. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, Feb 1998.

[SDH$^+$97] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–183, Oct 1997.

[SSC98] Mark Swanson, Leigh Stoller, and John Carter. Making distributed shared memory simple, yet efficient. In *Proc. of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environment*, March 1998.

[TTY99] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 60–71, May 1999.

[YKA96] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: A multigrain shared memory system. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.