# A Java Fork/Join Framework

Doug Lea
State University of New York at Oswego
Oswego NY 13126
315−341−2688
dl@cs.oswego.edu

## ABSTRACT

This paper describes the design, implementation, and performance of a Java framework for supporting a style of parallel programming in which problems are solved by (recursively) splitting them into subtasks that are solved in parallel, waiting for them to complete, and then composing results. The general design is a variant of the work−stealing framework devised for Cilk. The main implementation techniques surround efficient construction and management of tasks queues and worker threads. The measured performance shows good parallel speedups for most programs, but also suggests possible improvements.

## 1. INTRODUCTION

Fork/Join parallelism is among the simplest and most effective design techniques for obtaining good parallel performance. Fork/join algorithms are parallel versions of familiar divide−and−conquer algorithms, taking the typical form:

```
Result solve(Problem problem) {
  if (problem is small)
    directly solve problem
  else {
    split problem into independent parts
    fork new subtasks to solve each part
    join all subtasks
    compose result from subresults
  }
}
```

The `fork` operation starts a new parallel fork/join subtask. The `join` operation causes the current task not to proceed until the forked subtask has completed. Fork/join algorithms, like other divide−and−conquer algorithms, are nearly always recursive, repeatedly splitting subtasks until they are small enough to solve using simple, short sequential methods.

Some associated programming techniques and examples are discussed in section 4.4 of *Concurrent Programming in Java, second edition* [7]. This paper discusses the design (section 2), implementation (section 3), and performance (section 4) of `FJTask`, a Java$^{TM}$ framework that supports this programming style. `FJTask` is available as part of the `util.concurrent` package from `http://gee.cs.oswego.edu`.

## 2. DESIGN

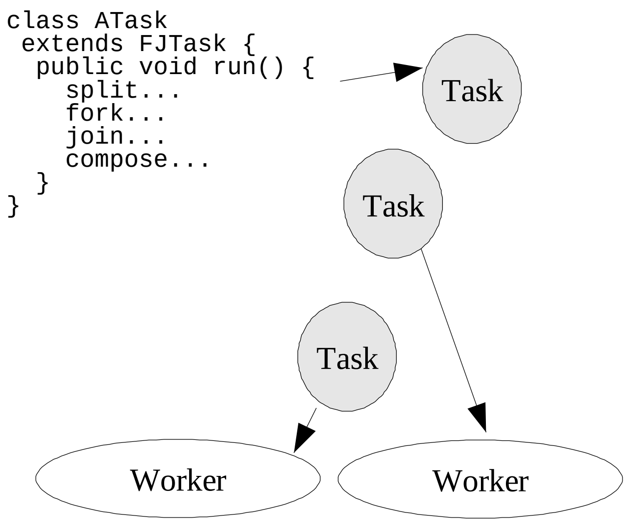
Fork/join programs can be run using any framework that supports construction of subtasks that are executed in parallel, along with a mechanism for waiting out their completion. However, the `java.lang.Thread` class (as well as POSIX pthreads, upon which Java threads are often based) are suboptimal vehicles for supporting fork/join programs:

- Fork/join tasks have simple and regular synchronization and management requirements. The computation graphs produced by fork/join tasks admit much more efficient scheduling tactics than needed for general−purpose threads. For example, fork/join tasks never need to block except to wait out subtasks. Thus, the overhead and bookkeeping necessary for tracking blocked general−purpose threads are wasted.
- Given reasonable base task granularities, the cost of constructing and managing a thread can be greater than the computation time of the task itself. While granularities can and should be subject to tuning when running programs on particular platforms, the extremely coarse granularities necessary to outweigh thread overhead limits opportunities for exploiting parallelism.

In short, standard thread frameworks are just too heavy to support most fork/join programs. But since threads form the basis of many other styles of concurrent and parallel programming as well, it is impossible (or at least impractical) to remove overhead or tune scheduling of threads themselves just for the sake of supporting this style.

While the ideas surely have a longer heritage, the first published framework to offer systematic solutions to these problems was Cilk[5]. Cilk and other lightweight executable frameworks layer special−purpose fork/join support on top of an operating system's basic thread or process mechanisms. This tactic applies equally well to Java, even though Java threads are in turn layered onto lower−level OS capabilities. The main advantage of creating such a Java lightweight execution framework is to enable fork/join programs to be written in a more portable fashion and to run on the wide range of systems supporting JVMs.

The `FJTask` framework is based on a variant of the design used in Cilk. Other variants are seen in Hood[4], Filaments[8], stackthreads[10], and related systems relying on lightweight



```
class ATask
 extends FJTask {
  public void run() {
    split...
    fork...
    join...
    compose...
  }
}
```

executable tasks. All of these frameworks map tasks to threads in about the same way that operating systems map threads to CPUs, but exploit the simplicity, regularity, and constraints of fork/join programs in performing the mapping. While all of these frameworks can accommodate (to varying extents) parallel programs written in different styles, they optimize for fork/join designs:

- A pool of worker threads is established. Each worker thread is a standard ("heavy") thread (here, an instance of `Thread` subclass `FJTaskRunner`) that processes tasks held in queues. Normally, there are as many worker threads as there are CPUs on a system. In native frameworks such as Cilk, these are mapped to kernel threads or lightweight processes, and in turn to CPUs. In Java, the JVM and OS must be trusted to map these threads to CPUs. However, this is a very simple task for the OS, since these threads are computationally intensive. Any reasonable mapping strategy will map these threads to different CPUs.
- All fork/join tasks are instances of a lightweight executable class, not instances of threads. In Java, independently executable tasks must implement interface `Runnable` and define a `run` method. In the `FJTask` framework, these tasks subclass `FJTask` rather than subclassing `Thread`, both of which implement `Runnable`. (In both cases, a class can alternatively implement `Runnable` and then supply instances to be run within executing tasks or threads. Because tasks operate under restricted rules supported by `FJTask` methods, it is much more convenient to subclass `FJTask`, so as to be able to directly invoke them.)
- A special purpose queuing and scheduling discipline is used to manage tasks and execute them via the worker threads (see section 2.1). These mechanics are triggered by those few methods provided in the task class: principally `fork`, `join`, `isDone` (a completion status indicator), and some convenience methods such as `coInvoke` that forks then joins two or more tasks.
- A simple control and management facility (here, `FJTaskRunnerGroup`) sets up worker pools and initiates execution of a given fork/join task when invoked from a normal thread (such as the one performing `main` in a Java program).

As the standard example of how this framework appears to the programmer, here is a class computing the Fibonacci function:

```
class Fib extends FJTask {
  static final int threshold = 13;
  volatile int number; // arg/result

  Fib(int n) { number = n; }

  int getAnswer() {
    if (!isDone())
      throw new IllegalStateException();
    return number;
  }

  public void run() {
    int n = number;
    if (n <= threshold) // granularity ctl
      number = seqFib(n);
    else {
      Fib f1 = new Fib(n − 1);
      Fib f2 = new Fib(n − 2);
      coInvoke(f1, f2);
      number = f1.number + f2.number;
    }
  }

  public static void main(String[] args) {
    try {
      int groupSize = 2; // for example
      FJTaskRunnerGroup group =
         new FJTaskRunnerGroup(groupSize);
      Fib f = new Fib(35); // for example
      group.invoke(f);
      int result = f.getAnswer();
      System.out.println("Answer: " +
                         result);
    }
    catch (InterruptedException ex) {}
  }

  int seqFib(int n) {
    if (n <= 1) return n;
    else return seqFib(n−1) + seqFib(n−2);
  }
}
```

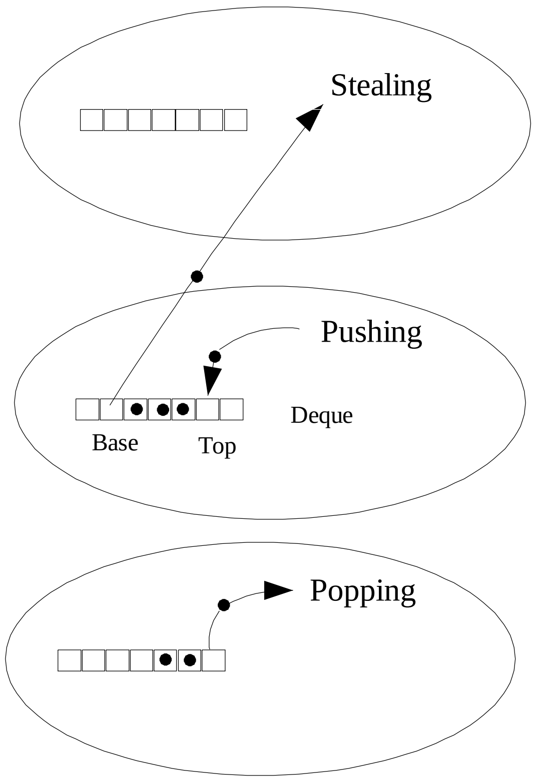
This version runs at least thirty times faster than an equivalent program in which each new task is run in a new `java.lang.Thread` on the platforms described in section 4. It does so while maintaining the intrinsic portability of multithreaded Java programs. There are only two tuning parameters of typical interest to programmers:

- The number of worker threads to construct, which should ordinarily correspond to the number of CPUs available on a platform (or fewer, to reserve processing for other unrelated purposes, or occasionally more, to soak up non−computational slack).
- A granularity parameter that represents the point at which the overhead of generating tasks outweighs potential parallelism benefits. This parameter is typically more algorithm−dependent than platform−dependent. It is normally possible to settle on a threshold that achieves good results when run on a uniprocessor, yet still exploits multiple CPUs when they are present. As a side−benefit, this approach meshes well with JVM dynamic compilation mechanics that optimize small methods better than monolithic procedures. This, along with data locality advantages, can cause fork/join algorithms to outperform other kinds of algorithms even on uniprocessors.

## 2.1 Work−Stealing

The heart of a fork/join framework lies in its lightweight scheduling mechanics. `FJTask` adapts the basic tactics pioneered in the Cilk work−stealing scheduler:

- Each worker thread maintains runnable tasks in its own scheduling queue.
- Queues are maintained as double−ended queues (i.e., deques, usually pronounced "decks"), supporting both LIFO `push` and `pop` operations, as well as a FIFO `take` operation.
- Subtasks generated in tasks run by a given worker thread are pushed onto that workers own deque.
- Worker threads process their own deques in LIFO (youngest−first) order, by popping tasks.
- When a worker thread has no local tasks to run, it attempts to take ("*steal*") a task from another randomly chosen worker, using a FIFO (oldest first) rule.
- When a worker thread encounters a `join` operation, it processes other tasks, if available, until the target task is noticed to have completed (via `isDone`). All tasks otherwise run to completion without blocking.
- When a worker thread has no work and fails to steal any from others, it backs off (via yields, sleeps, and/or priority adjustment − see section 3) and tries again later unless all workers are known to be similarly idle, in which case they all block until another task is invoked from top−level.



As discussed in more detail in [5], the use of LIFO rules for each thread processing its own tasks, but FIFO rules for stealing other tasks is optimal for a wide class of recursive fork/join designs. Less formally, the scheme offers two basic advantages: It reduces contention by having stealers operate on the opposite side of the deque as owners. It also exploits the property of recursive divide−and−conquer algorithms of generating "large" tasks early. Thus, an older stolen task is likely to provide a larger unit of work, leading to further recursive decompositions by the stealing thread.

As one consequence of these rules, programs that employ relatively small task granularities for base actions tend to run faster than those that only use coarse−grained partitioning or those that do not use recursive decomposition. Even though relatively few tasks are stolen in most fork/join programs, creating many fine−grained tasks means that a task is likely to be available whenever a worker thread is ready to run it.

## 3. IMPLEMENTATION

The framework has been implemented in about 800 lines of pure Java code, mainly in class `FJTaskRunner`, a subclass of `java.lang.Thread`. `FJTasks` themselves maintain only a boolean completion status, and perform all other operations via delegation to their current worker threads. The `FJTaskRunnerGroup` class serves to construct worker threads, maintains some shared state (for example, the identities of all worker threads, needed for steal operations), and helps coordinate startup and shutdown.

More detailed implementation documentation is available inside the `util.concurrent` package. This section discusses only two sets of problems and solutions encountered when implementing this framework: Supporting efficient deque operations (`push`, `pop`, and `take`), and managing the steal protocol by which threads obtain new work.

## 3.1 Deques

To enable efficient and scalable execution, task management must be made as fast as possible. Creating, pushing, and later popping (or, much less frequently, taking) tasks are analogs of procedure call overhead in sequential programs. Lower overhead enables programmers to adopt smaller task granularities, and in turn better exploit parallelism.

Task allocation itself is the responsibility of the JVM. Java garbage collection relieves us of needing to create a special−purpose memory allocator to maintain tasks. This substantially reduces the complexity and lines of code needed to implement `FJTasks` compared to similar frameworks in other languages.

The basic structure of the deque employs the common scheme of using a single (although resizable) array per deque, along with two indices: The `top` index acts just like an array−based stack pointer, changing upon `push` and `pop`. The `base` index is modified only by `take`. Since `FJTaskRunner` operations are all intimately tied to the concrete details of the deque (for example, `fork` simply invokes `push`), this data structure is directly embedded in the class rather than being defined as a separate component.

Because the deque array is accessed by multiple threads, sometimes without full synchronization (see below), yet individual Java array elements cannot be declared as `volatile`, each array element is actually a fixed reference to a little forwarding object maintaining a single `volatile` reference. This decision was made originally to ensure conformance with Java memory rules, but the level of indirection that it entails turns out to improve performance on tested platforms, presumably by reducing cache contention due to accesses of nearby elements, which are spread out a bit more in memory due to the indirection.

The main challenges in deque implementation surround synchronization and its avoidance. Even on JVMs with optimized synchronization facilities[2], the need to obtain locks for every `push` and `pop` operation becomes a bottleneck. However, adaptations of tactics taken in Cilk[5] provide a solution based on the following observations:

- The `push` and `pop` operations are only invoked by owner threads.
- Access to the `take` operation can easily be confined to one stealing thread at a time via an entry lock on `take`. (This deque lock also serves to disable `take` operations when necessary.) Thus, interference control is reduced to a two−party synchronization problem.
- The `pop` and `take` operations can only interfere if the deque is about to become empty. Otherwise they are guaranteed to operate on disjoint elements of the array.

Defining the `top` and `base` indices as `volatile` ensures that a `pop` and `take` can proceed without locking if the deque is sure to have more than one element. This is done via a Dekker−like algorithm in which `push` pre−decrements `top`:

```
 if (−−top >= base) ...
```

and `take` pre−increments `base`:

```
 if (++base < top) ...
```

In each case they must then check to see if this could have caused the deque to become empty by comparing the two indices. An asymmetric rule is used upon potential conflict: `pop` rechecks state and tries to continue after obtaining the deque lock (the same one as held by `take`), backing off only if the deque is truly empty. A `take` operation instead just backs off immediately, typically then trying to steal from a different victim. This asymmetry represents the only significant departure from the otherwise similar *THE* protocol used in Cilk.

The use of `volatile` indices also enables the `push` operation to proceed without synchronization unless the deque array is about to overflow, in which case it must first obtain the deque lock to resize the array. Otherwise, simply ensuring that `top` is updated only after the deque array slot is filled in suppresses interference by any `take`.

Subsequent to initial implementation, it was discovered that several JVMs do not conform to the Java Memory Model [6] rule requiring accurate reads after writes of pairs of `volatile` fields. As a workaround, the criterion for `pop` to retry under lock was adjusted to trigger if there appear to be *two* or fewer elements, and the `take` operation added a secondary lock to ensure a memory barrier. This suffices as long as at most one index change is missed by the owner thread (which holds here for platforms that otherwise maintain proper memory order when reading `volatile` fields), and causes only a tiny slowdown in performance.

## 3.2 Stealing and Idling

Worker threads in work−stealing frameworks know nothing about the synchronization demands of the programs they are running. They simply generate, push, pop, take, manage the status of, and execute tasks. The simplicity of this scheme leads to efficient execution when there is plenty of work for all threads. However, this streamlining comes at the price of relying on heuristics when there is not enough work; i.e., during startup of a main task, upon its completion, and around global full−stop synchronization points employed in some fork/join algorithms.

The main issue here is what to do when a worker thread has no local tasks and cannot steal one from any other thread. If the program is running on a dedicated multiprocessor, then one could make the case for relying on hard busy−wait spins looping to try to steal work. However, even here, attempted steals increase contention, which can slow down even those threads that are not idle (due to locking protocols in section 3.1). Additionally, in more typical usage contexts of this framework, the operating system should somehow be convinced to try to run other unrelated runnable processes or threads.

The tools for achieving this in Java are weak, have no guarantees (see [6, 7]), but usually appear to be acceptable in practice (as do similar techniques described for Hood[3]). A thread that fails to obtain work from any other thread lowers its priority before attempting additional steals, performs `Thread.yield` between attempts, and registers itself as inactive in its `FJTaskRunnerGroup`. If all others become inactive, they all block waiting for additional main tasks. Otherwise, after a given number of additional spins, threads enter a sleeping phase, where they sleep (for up to 100ms) rather than yield between steal attempts. These imposed sleeps can cause artificial lags in programs that take a long time to split their tasks. But this appears to be the best general−purpose compromise. Future versions of the framework may supply additional control methods so that programmers can override defaults when they impact performance.

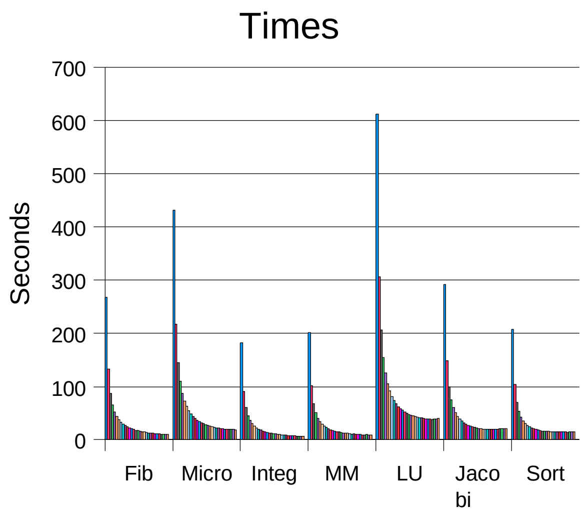
## 4. PERFORMANCE

In these times of nearly continuous performance improvements of compilers and JVMs, performance measurements are only of transient value. However, the metrics reported in this section reveal some basic properties of the framework.

A collection of seven fork/join test programs are briefly described in the following table. These programs are adaptations of those available as demos in the `util.concurrent` package. They were selected to show some diversity in the kinds of problems that can be run within this framework, as well as to obtain results for some common parallel test programs.

| ***Program*** | ***Description*** |
|---|---|
| Fib | The Fibonnaci program shown in section 2, run with argument 47 and granularity threshold 13. |
| Integrate | Recursive Gaussian quadrature of $(2 \cdot i - 1) \cdot x^{(2 \cdot i - 1)}$ summing over odd values of i from 1 to 5 and integrating from −47 to 48. |
| Micro | Best−move finder for a board game, run with a lookahead of 4 moves. |
| Sort | Merge/Quick sort (based on an algorithm from Cilk) of 100 million numbers. |
| MM | Multiply 2048 X 2048 matrices of doubles. |
| LU | Decompose 4096 X 4096 matrix of doubles. |
| Jacobi | Iterative mesh relaxation with barriers: 100 steps of nearest neighbor averaging on a 4096 X 4096 matrix of doubles. |

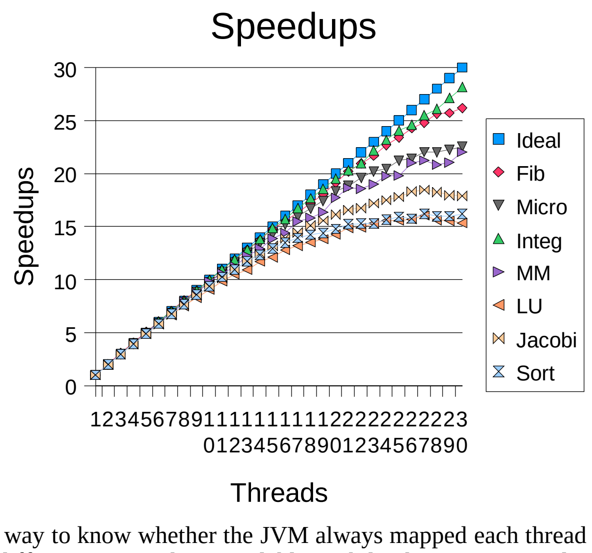
For the main tests, programs were run on a 30−CPU Sun Enterprise 10000 running the Solaris Production 1.2 JVM (an early version of the 1.2.2_05 release) on Solaris 7. The JVM was run with environment parameters selecting "bound threads" for thread mappings, and memory parameters discussed in section 4.2. A few additional measurements reported below were run on a 4−CPU Sun Enterprise 450.



Programs were run with very large input parameters in order to minimize timer granularity and JVM warm−up artifacts. Some other warm−up effects were avoided by running an initial problem set before starting timers. Most data are medians of three runs, but some (including most follow−up measurements in sections 4.2−4.4) only reflect single runs so are a bit noisy.

## 4.1 Speedups

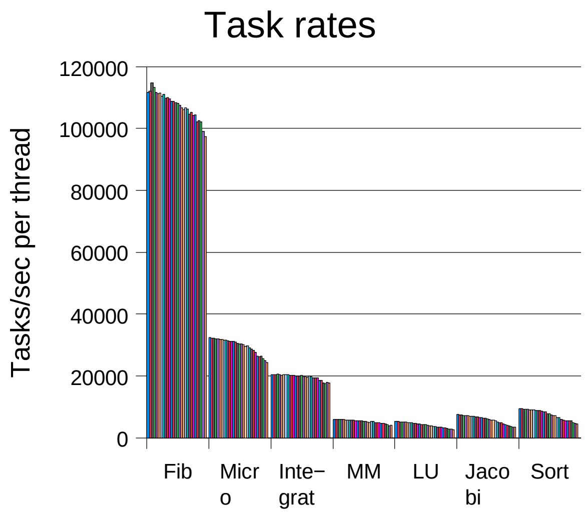
Scalability measurements were obtained by running the same problem set using worker thread groups of size 1 ... 30. There is



no way to know whether the JVM always mapped each thread to a different CPU when available. While there is no evidence about this either way, it is possible that the lags to map new threads to CPUs increased with numbers of threads and/or varied systematically across the different test programs. However, in general, the results show that increasing the number of threads reliably increased the number of CPUs employed.

Speedups are reported as $Time_n$ / $Time_1$. The best overall speedup was seen for the integration program (speedup of 28.2 for 30 threads). The worst was for the LU decomposition program (speedup of 15.35 for 30 threads).

Another way of measuring scalability is in terms of task rates, the average time taken to execute a single task (which may be either a recursive or a leaf step). The following figure shows data from a single instrumented run capturing task rates. Ideally, the numbers of tasks processed per unit time per thread should be constant. The fact that they generally slightly decrease with numbers of threads indicates some scalability limitations. Note the fairly wide difference in task rates, reflecting differences in task granularities. The smallest tasks sizes are seen in Fib, which even with a threshold setting of 13, generated and executed a total of 2.8 million tasks per second when run with 30 threads.

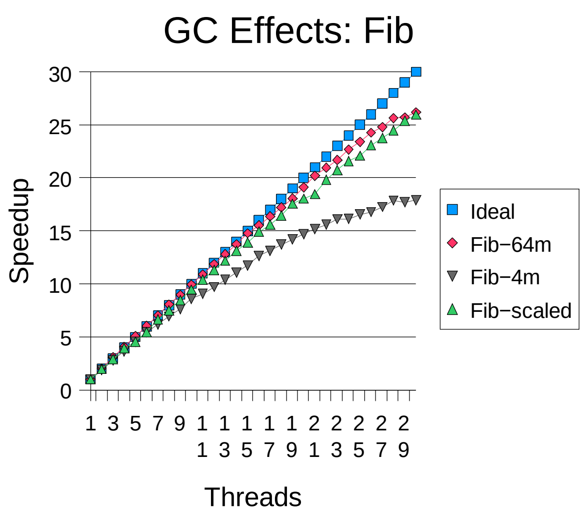


Four factors appear to account for tail−offs in speedups in those programs that did not scale linearly. Three of them are common to any parallel framework, but we'll start with one unique to `FJTask` (as opposed to Cilk etc), GC effects.

## 4.2 Garbage Collection

In many ways, modern GC facilities are perfect matches to fork/join frameworks: These programs can generate enormous numbers of tasks, nearly all of which quickly turn into garbage after they are executed. At any given time, deterministic fork/join programs require only at most *p* (where *p* is the number of threads) times the bookkeeping space that would be required for sequential versions of these programs[10]. Generational semispace copying collectors (including the one in the JVM used for these measurements − see [1]) cope with this well because they only traverse and copy non−garbage cells. By doing so, they evade one of the trickiest problems in manual parallel memory management, tracing memory that is allocated by one thread but then used by another. The garbage collector is oblivious to the origin of memory, so need not deal with such issues.

As a simple indicator of the general superiority of generational copying collection, a four−thread run of Fib that executes in 5.1 seconds when using the memory settings used in the main experiments reported here runs in 9.1 seconds if the generational



copying phase is disabled on this JVM (in which case this JVM relies entirely on mark−sweep).

However, these GC mechanisms turn into scaling problems when memory is allocated at such a high rate that threads must be stopped nearly continuously to perform collections. The following figure shows the difference in speedups across three memory settings (this JVM supports optional arguments to set memory parameters): with the default 4 megabyte semispaces, with 64 megabyte semispaces, and with the amount of memory scaled to the number of threads (2 + 2p megabytes). With smaller semispaces, the overhead of stopping threads and collecting garbage starts to impact scaling as the garbage−generation rate climbs due to additional threads.

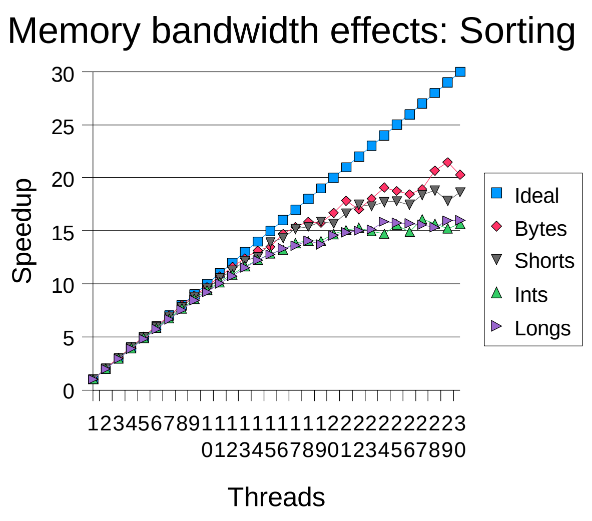
In light of these results, 64m semispaces were used for all other test runs. A better policy would have been to scale the amount of memory to the number of threads in each test. (As seen in the figure, this would have caused all speedups to appear slightly more linear.). Alternatively, or in addition, program−specific task granularity thresholds could be increased proportionally with numbers of threads.

## 4.3 Memory Locality and Bandwidth

Four of the test programs create and operate on very large shared arrays or matrices: Sorting numbers, and multiplying, decomposing or performing relaxation on matrices. Of these, sorting is probably the most sensitive to the consequences of having to move data around processors, and thus to the aggregate memory bandwidth of the system as a whole. To help determine the nature of these effects, the Sort program was recast into four versions, that sorted bytes, shorts, ints, and longs respectively. Each version sorted data only in the range of 0...255, to ensure that they were otherwise equivalent. The wider the data elements, the more memory traffic.

The results show that increased memory traffic leads to poorer speedups, although they do not provide definitive evidence that this is the only cause of tail−offs.

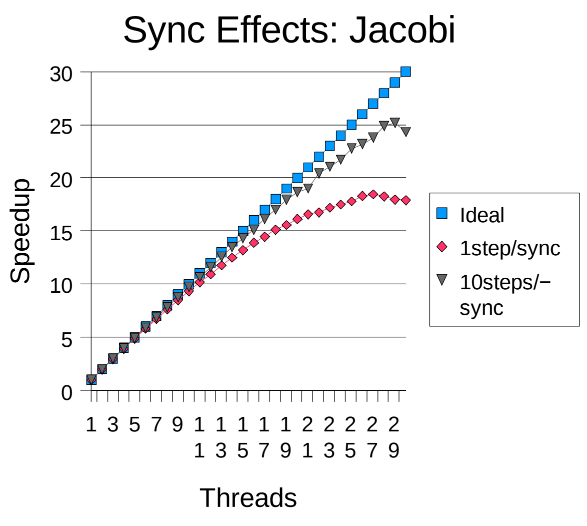
Element width also impacts absolute performance. For example, using only one thread, sorting bytes took 122.5 seconds, while sorting longs took 242.4 seconds.



## 4.4 Task Synchronization

As discussed in section 3.2, work−stealing frameworks sometimes encounter problems dealing with frequent global synchronization of tasks. The worker threads continue to poll for more work even though there isn't any, thus generating contention, and, in `FJTask`, sometimes even forcing threads into idle sleeps.

The Jacobi program illustrates some of the consequent issues. This program performs 100 steps, where in each step, all cells are updated according to a simple nearest neighbor averaging formula. A global (tree−based) barrier separates each step. To determine the magnitude of synchronization effects, a version of

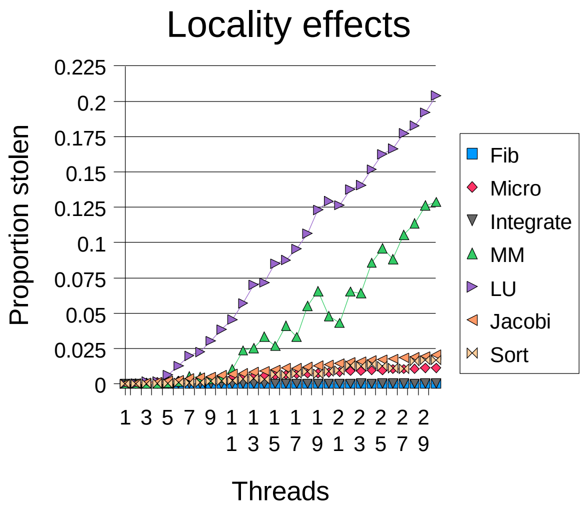


this program was written to synchronize only after every 10 steps. The scaling differences show the impact of the current policy, and indicate the need to include additional methods in future versions of this framework to allow programmers to override default parameters and policies. (Note however that this graph probably slightly exaggerates pure synchronization effects, since the 10−step version is also likely to maintain somewhat greater task locality.)

### 4.5 Task Locality

`FJTask`, like other fork/join frameworks, is optimized for the case where worker threads locally consume the vast majority of the tasks they create. When this does not occur, performance can suffer, for two reasons:

- Stealing tasks encounters much more overhead than does popping tasks.
- In most programs in which tasks operate on shared data, running your own subdivided task is likely to maintain better locality of data access.

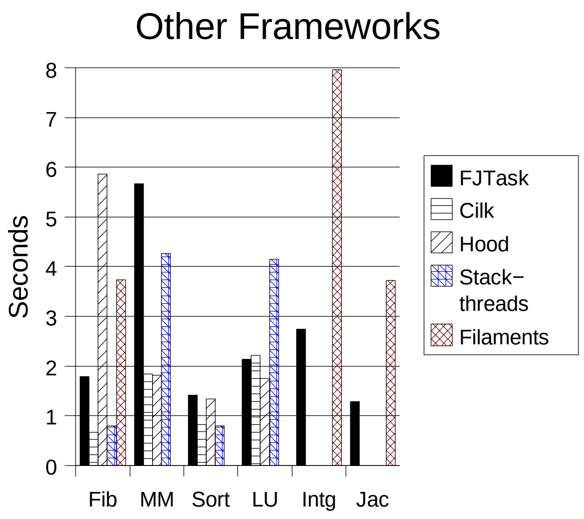


As seen in the  figure, in most programs, the relative number of stolen tasks is at most a few percent. However, the LU and MM programs generate larger imbalances in workloads (and thus relatively more stealing) as the number of threads increase. It is possible that some algorithmic tuning of these programs could reduce this effect, and in turn lead to better speedups.

### 4.6 Comparisons to Other Frameworks

It is impossible to perform definitive or even very meaningful measurements comparing programs across languages and frameworks. However, relative measurements can at least show some of the basic advantages and limitations of the `FJTask` framework versus similar frameworks written in other languages (here, C and C++). The  following chart compares performance of programs that were either based on or are similar to those supplied with the Cilk, Hood, Stackthreads, and/or Filaments distributions. All of these were run with 4 threads on a 4−CPU Enterprise 450. To avoid reconfiguring other frameworks or their test programs, all tests were run with smaller problems sets than used above. All results represent the best of three runs, using the compiler and run−time settings that appeared to provide the fastest times. Fib was run without any granularity threshold; i.e., an implicit threshold of 1. (The "prune" setting in Filaments Fib was set to 1024, which causes it to behave more similarly to the other versions.)

Speedups going from one to four threads were very similar (between 3.0 and 4.0) across the different frameworks and test programs, so the accompanying chart focuses on differences in absolute performance. However, because the multithreading aspects of all these frameworks are all fast, most of these differences reflect differences in application−specific code quality stemming from different compilers, optimization switches, and configuration parameters. In fact, it is likely that



different choices among these than used here could produce nearly any relative performance ranking among these frameworks in many high−performance applications.

`FJTask` generally performs worse (on this JVM) on programs that mainly do floating point computations on arrays and matrices. Even though JVMs are improving, they are still not always competitive with powerful back−end optimizers used in C and C++ programs. Although not shown in this chart, `FJTask` versions of *all* programs ran faster than versions of programs in these other frameworks when they were compiled without optimization enabled, and some informal tests suggest that most of the remaining differences are due to array bounds checks and related runtime obligations. These are, of course, issues and problems that are attracting much attention and effort by JVM and compiler developers. The observed differences in code quality for computationally intensive programs are likely to diminish.

## 5. CONCLUSIONS

This paper has demonstrated that it is possible to support portable, efficient, scalable parallel processing in pure Java, and to provide a convenient API for programmers who can exploit the framework by following only a few simple design rules and design patterns (as presented and discussed in [7]). The observed performance characteristics of the sample programs measured here both provide further guidance for users of the framework, and suggest some areas of potential improvement to the framework itself.

Even though scalability results were shown only for a single JVM, some of the main empirical findings should hold more generally:

- While generational GC generally meshes well with parallelism, it can hinder scalability when garbage generation rates force very frequent collections. On this JVM, the underlying cause appears to be that stopping threads for collection takes time approximately proportional to the number of running threads. Because more running threads generate more garbage per unit time, overhead can climb approximately quadratically with numbers of threads. Even so, this significantly impacts performance only when the GC rate is relatively high to begin with. However, the resulting issues invite further research and development of concurrent and parallel GC algorithms. The results presented here additionally demonstrate the desirability of providing tuning options and adaptive mechanisms on multiprocessor JVMs to scale memory to the number of active processors.

- Most scalability issues reveal themselves only when programs are run using more CPUs than are even available on most stock multiprocessors. `FJTask` (as well as other fork/join frameworks) appear to provide nearly ideal speedups for nearly any fork/join program on commonly available 2−way, 4−way, and 8−way SMP machines. The present paper appears to be the first reporting systematic results for any fork/join framework designed for stock multiprocessors run on more than 16 CPUs. Further measurements are needed to see if the patterns of results seen here hold in other frameworks as well.
- Characteristics of application programs (including memory locality, task locality, use of global synchronization) often have more bearing on both scalability and absolute performance than do characteristics of the framework, JVM or underlying OS. For example, informal tests showed that the careful avoidance of synchronization in deques (discussed in section 3.1) has essentially no impact on programs with relatively low task generation rates such as LU. However, the focus on keeping task management overhead to a minimum widens the range of applicability and utility of the framework and the associated design and programming techniques.

In addition to incremental improvements, future work on this framework may include construction of useful applications (as opposed to demos and tests), subsequent evaluations under production program loads, measurements on different JVMs, and development of extensions geared for use with clusters of multiprocessors.

## 6. ACKNOWLEDGMENTS

This work was supported in part by a collaborative research grant from Sun Labs. Thanks to Ole Agesen, Dave Detlefs, Christine Flood, Alex Garthwaite, and Steve Heller of the Sun Labs Java Topics Group for advice, help, and comments. David Holmes, Ole Agesen, Keith Randall, Kenjiro Taura, and the anonymous referees provided useful comments on drafts of this paper. Bill Pugh pointed out the read−after−write limitations of JVMs discussed in section 3.1. Very special thanks to Dave Dice for reserving time and performing test runs on the 30−way Enterprise.

## 7. REFERENCES

[1] Agesen, Ole, David Detlefs, and J. Eliot B. Moss. Garbage Collection and Local Variable Type−Precision and Liveness in Java Virtual Machines. In *Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[2] Agesen, Ole, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, and Derek White. An Efficient Meta−lock for Implementing Ubiquitous Synchronization. In *Proceedings of OOPSLA ’99*, ACM, 1999.

[3] Arora, Nimar, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 28 − July 2, 1998.

[4] Blumofe, Robert D. and Dionisios Papadopoulos. *Hood: A User−Level Threads Library for Multiprogrammed Multiprocessors*. Technical Report, University of Texas at Austin, 1999.

[5] Frigo, Matteo, Charles Leiserson, and Keith Randall. The Implementation of the Cilk−5 Multithreaded Language. In *Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[6] Gosling, James, Bill Joy, and Guy Steele. *The Java Language Specification*, Addison−Wesley, 1996.

[7] Lea, Doug. *Concurrent Programming in Java, second edition*, Addison−Wesley, 1999.

[8] Lowenthal, David K., Vincent W. Freeh, and Gregory R. Andrews. Efficient Fine−Grain Parallelism on Shared−Memory Machines. *Concurrency−Practice and Experience*, 10,3:157−173, 1998.

[9] Simpson, David, and F. Warren Burton. Space efficient execution of deterministic parallel programs. *IEEE Transactions on Software Engineering*, December, 1999.

[10] Taura, Kenjiro, Kunio Tabata, and Akinori Yonezawa. "Stackthreads/MP: Integrating Futures into Calling Standards." In *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, 1999.