Atlas: An Infrastructure for Global Computing

J. Eric Baldeschwieler* Robert D. Blumofe† Eric A. Brewer*

*Computer Science Division
The University of California at Berkeley
Berkeley, California 94720
{eric14,brewer}@cs.berkeley.edu

†Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
rdb@cs.utexas.edu

Abstract

In this paper, we present a proposed system architecture for global computing that we call Atlas, and we describe an early prototype that implements several of the mechanisms and policies that comprise the proposed architecture. Atlas is designed to execute parallel multithreaded programs on the networked computing resources of the world. The Atlas system is a marriage of existing technologies from Java and Cilk together with some new technologies needed to extend the system into the global domain.

1 Introduction

The goal of the Atlas system is to exploit the networked resources of the world as a giant distributed computer, and to develop an infrastructure that exploits idle resources both within and among institutions. The Atlas architecture strives for several desirable properties for a global computing infrastructure.

Scalability: It must scale to millions of nodes.

Heterogeneity: It must include a variety of platforms and operating systems, all of which must interoperate.

Fault tolerance: An application should tolerate most failures and should have a probability of completion consistent with traditional applications.

Adaptive parallelism: The applications must dynamically exploit a varying collection of resources, so that long-running applications can grow, shrink, and migrate as required.

Safety: The system must ensure that foreign applications obey local limits on data access and resource utilization, so that users will be willing to add their machines to the global pool.

Anonymity: When they care, users should be able to farm out their applications without risking any proprietary data.

Hierarchy: The infrastructure should exploit existing hierarchical domains, so that pooled resources are given preferentially to other members of their organization, and so that applications can exploit (higher) local bandwidth.

Ease of use: The system should be easy to use.

Reasonable performance: The performance should show linear speedup for large classes of applications and should have sufficiently low overhead so as to see a benefit even with only a few machines.

Atlas realizes these properties by combining existing mechanisms and policies from *Java* [1] and *Cilk* (pronounced "silk") [2, 4] together with some new mechanisms and policies that extend Atlas into a global computing infrastructure. Java is an object-oriented programming language together with a virtual-machine specification. Java programs are compiled to machine-independent byte codes that are executed by a virtual-machine interpreter running on the host machine. The Java virtual machine means that Java programs are

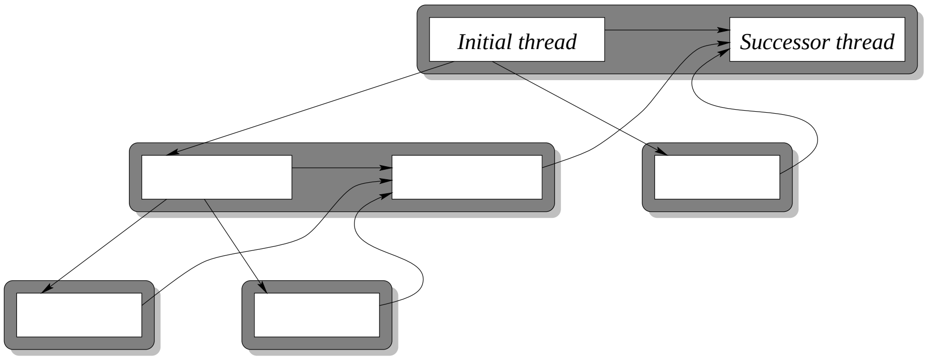


Figure 1: The ATLAS programming model. The shaded regions represent procedures. Each procedure contains one or more threads. Down-pointing, straight arrows represent spawns of children; right-pointing, straight arrows represent spawns of successors; and curved arrows represent the "returning" of values from children to successors.

essentially platform independent (heterogeneous) and can be executed safely on untrusting machines. Cilk is an algorithmic, C-based parallel multithreaded programming language together with a runtime system that employs a provably efficient thread-scheduling algorithm based on the technique of "work stealing" [2, 5] in which idle processors steal threads from randomly chosen victims. Cilk programs execute with efficient and predictable performance, guaranteed. In addition, the *Cilk-NOW* runtime system [2] provides automatic fault tolerance and adaptive parallelism for Cilk programs[1] executing on a network of workstations. ATLAS adapts the Cilk programming model to Java; it extends Cilk's work-stealing scheduler with a hierarchy; and it borrows mechanisms from Cilk-NOW to implement adaptive parallelism and fault tolerance.

The remainder of this paper is organized as follows. In Section 2, we describe the programming model that ATLAS supports, and in Section 3, we describe the architecture of ATLAS. Then, in Section 4, we present two example ATLAS applications together with some performance data. In Section 5, we discuss how the ATLAS architecture achieves our desired properties. Finally, in Section 6, we overview some related work, and in Section 7, we conclude.

2 Programming model

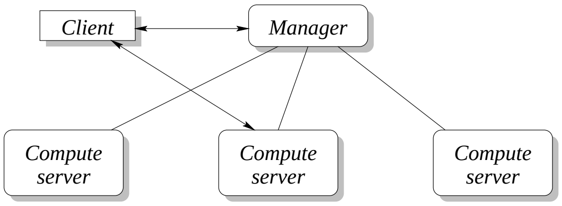
In this section, we describe the programming model that ATLAS supports. Based on multithreading, this programming model is largely borrowed from Cilk and adapted to Java. An essential feature of this programming model is that it abstracts all physical resources so that the runtime system is free to manage all physical resources in the highly dynamic manner required for global-scale execution.

ATLAS applications are written mostly in Java with some use of C or C++ when extra performance is needed. Pure Java applications have the advantage that they are heterogeneous and do not require special libraries. When a native library is used, the application is limited to run on machines that have or can dynamically link in that library, which may also limit which platforms the application can exploit.

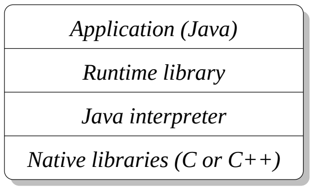
The ATLAS programming model is a straightforward adaptation of the Cilk programming model [2, 4] to Java with the addition of a URL-based file system. A complete description of this programming model is beyond the scope of this paper.

Briefly, however, an ATLAS program contains one or more ATLAS *procedures*, and as illustrated in Figure 1, an ATLAS procedure consists of a sequence of non-blocking, continuation-passing *threads*. A procedure *spawns* a child when a thread of the parent spawns a child thread that becomes the initial thread of the child procedure. A spawn is the parallel analogue of a procedure call. The difference is that a spawned child may execute concurrently with its parent (and its siblings). After spawning one or more children, a procedure may

[1] Cilk-NOW supports the Cilk-2 language which is essentially a functional subset of the recently released Cilk-4 language.



(a) Basic architecture.



(b) Compute server.

Figure 2: **(a)** The client first asks the manager for the name of a compute server. It then connects to that server to run the application. The manager will then help other servers steal work when they are idle. **(b)** Most of the application is written in Java. The server contains a Java interpreter, a runtime system written in a combination of C and Java, and a set of native libraries that provide additional functionality or performance.

synchronize with its children by blocking until one or more of its children complete. Since a thread cannot block, a procedure must employ a sequence of threads in order to obtain blocking behavior. In addition to spawning children, a thread may spawn a *successor* that is part of the same procedure. The successor thread cannot begin execution until it receives all of its arguments which are "returned" from the children. Thus, the successor thread implements the blocking behavior of procedures by waiting for arguments from the children. Program execution can be viewed as a tree of procedures with a partial execution order on the threads.

The ATLAS programmer does not specify when nor on which processor any thread executes. Correct execution requires only that each thread is executed on a processor at a time consistent with the partial execution order. Scheduling is performed automatically by the runtime system using work stealing as covered in Section 3. Thus, an ATLAS programmer employs threads to express the parallelism and locality in the program, while the runtime system takes care of the details of protocols, scheduling, and resource management.

ATLAS also provides a global file system based on URLs and local caching with coherence. The file system provides versions of the standard Unix file routines (e.g., `fopen`) that take URLs as file names [14]. This allows applications to access files on the home file system from anywhere. The file system uses block-based cache coherence to achieve reasonable performance, and future versions will provide ticket-based access control.

3 Architecture

The ATLAS system architecture consists of clients, managers, and compute servers. As shown in Figure 2(a), a client with an application to run contacts the local manager to find an up (and idle) compute server. It then connects to this server to run the application and provides feedback to the user while the application is running. During execution, idle servers steal work from those that are busy, so that the application eventually spreads to all available resources.

The structure of a compute server is shown in Figure 2(b). The runtime library is responsible for work stealing, thread management, and marshalling and unmarshalling objects for communication to other (heterogeneous) servers. The Java interpreter runs the application and the Java half of the runtime system. The native libraries (if any) extend the server with new capabilities that applications may exploit or even require. Native libraries are optional, since not all platforms have to run all applications.

Each compute server runs as a daemon. It runs all the time, either working on some application thread or pinging its manager and siblings for threads to steal. The server process runs with low priority and relinquishes the node whenever there is non-ATLAS work to do. Porting the runtime system to a new platform requires a Java interpreter for that platform and conversion of the native part of the runtime system, which includes the URL-based file system and some support routines.

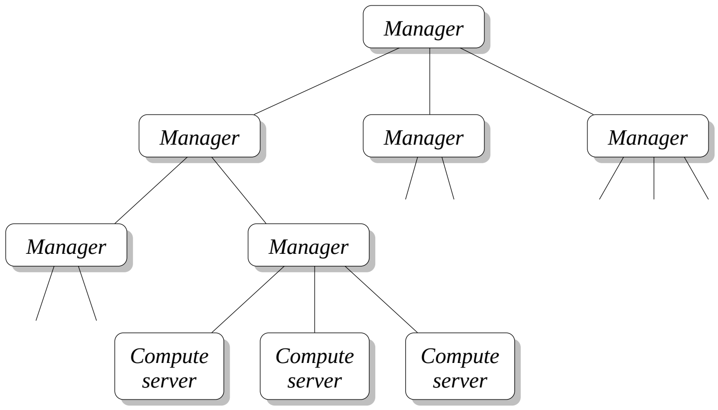


Figure 3: Hierarchical Stealing. The manager keeps track of when its entire subtree is idle. The manager then steals work for its subtree from its manager and siblings. Similarly, when a subtree has too much work, it can allow stealing from above.

3.1 Work stealing

The runtime system employs the provably efficient technique of work stealing to perform thread scheduling. The work-stealing scheduling algorithm is fully distributed, scalable, and efficient. Execution begins with the initial compute server executing the initial thread of the program's root procedure. Other idle compute servers perform work stealing as described below.

In general, a compute server with threads to work on executes in a depth-first, stack-based manner. Specifically, the compute server repeatedly pops the bottommost thread of its stack and then executes the thread. If the thread spawns a child, the compute server pushes the child onto the bottom of its stack. If the thread "returns" a value to a waiting successor thread in its parent procedure and if this successor thread now has all of its arguments, then the compute server pushes the successor thread onto the bottom of its stack.

If a compute server finds that its stack is empty, it becomes a thief and performs work stealing. The thief chooses a victim compute server at random and requests a thread from the chosen victim. If the victim has more than one thread on its stack, then the victim sends its topmost (oldest) thread to the thief. Otherwise, the victim rejects the request, and the thief must choose another random victim. Thus, compute servers execute locally in depth-first order, and they steal in breadth-first order.

In the ATLAS programming model, a procedure may synchronize only with its children, and for such programs, this work-stealing algorithm is efficient both in theory [2, 5] and in practice [2, 4]. Specifically, programs achieve linear speedup whenever the average parallelism is greater than the number of processors. Moreover, programs are guaranteed to make efficient use of stack space, and the scheduler existentially minimizes communication to within a constant factor. These properties allow a program to utilize a large-scale and distributed set of compute resources with guaranteed efficiency.

3.2 Hierarchical work stealing

A key part of the global scalability is hierarchical stealing, which is shown in Figure 3. Just as servers steal from each other, managers balance work at the next level up. Thus, when a subtree has too much work, threads will be stolen "from above" to run on other clusters. Similarly, subtrees that are idle will steal work from their siblings.

The hierarchy has two main goals. First, it tends to keep computations localized by forcing applications to use sibling resources first. Local clusters typically have both more bandwidth and lower latency than a more distributed group. Thus, localization makes better use of resources than a more random or distributed

```
void FibRecurse(SyncVar resultV, int n) {
    if (n<2) {
        resultV.set(n);
    } else {
        SyncVar aV = spawn(FibRecurse, n-1);
        SyncVar bV = spawn(FibRecurse, n-2);
        spawnSuccessor(FibSum, resultV, aV, bV);
    }
}

void FibSum(SyncVar resultV, int a, int b) {
    resultV.set(a + b);
}
```

Figure 4: Basic ATLAS code for `Fib`.

	Base	1 Node	3 Nodes	8 Nodes
`Fib`(24)	1.3	80	40 (2.0)	31 (2.6)
`POV-Ray`	20700	21000	—	2700 (7.8)

Table 1: Performance data for the `Fib` and `POV-Ray` programs. Execution times are reported in seconds. The numbers in parentheses are speedups over the 1-node case.

approach would achieve. The second goal is administrative: the subtrees map naturally to existing administrative domains, which has two important effects. First, administrators can explicitly and independently control whether or not threads can enter or leave their subtree. The main reason for limiting outflow would be concern for proprietary data, which is partially addressed later. Reasons for limiting inflow include security fears and the desire for a controlled environment for testing.

The hierarchy has a related social advantage, which is that it will encourage organizations to provide servers at least for themselves. For example, there are many groups in the Computer Science Division at UC Berkeley, and each group already has several machines that would form a cluster. With the hierarchy, a job would expand first to the group's cluster, then to other groups in the division, then to other divisions in the college. Thus Berkeley users get all of the power of Berkeley's machines before any outsiders do. Only if there are not threads available at Berkeley would outside threads be stolen.

4 Examples: `Fib` and `POV-Ray`

In this section, we cover two example programs based on an initial prototype version of ATLAS.

The first example is the `Fib` program, which uses double recursion to compute Fibonacci numbers. Though double recursion is a terrible way to compute Fibonacci numbers, this program is a useful benchmark. Since there is essentially no work per thread, this program stresses the thread creation and synchronization overhead. Figure 4 shows the basic code for the `Fib` program, while Table 1 shows the performance on different size clusters. The overhead is about a factor of 60 over the pure Java ("base") version. Although high, this overhead is the worst case—we have all of the thread creation and synchronization overhead and no work over which to amortize it.

The second example is a parallelized version of the `POV-Ray` ray-tracing engine [11], which has relatively large threads consisting of groups of scan lines. `POV-Ray` is a public domain ray tracer that has already been ported to many platforms. For this application, we converted the engine to a native library and wrote the partitioning software in Java. Thus, the base case is to render part of a scene and then combine partial scenes as the threads come back together. Unlike `Fib`, `POV-Ray` has a relatively large thread size. As shown in Table 1, we achieve a near linear speedup of 7.8 on 8 nodes. The overhead is only 2 percent over the stand-alone ("base") `POV-Ray` executable, which confirms that the overhead is reasonable for large-grain threads.

5 Discussion

In this section, we discuss how the ATLAS architecture achieves the desired properties listed in the introduction.

Scalability: All parts of the system are scalable. The scalability of the scheduling and the file system comes from partitioned responsibility and local caching. The scalability of the applications comes from the programming model, which has no predefined limit on the usable resources, such as a target number of processors.

Heterogeneity: The basic heterogeneity comes from Java for free, however, we must be able to port the server and all of its libraries to each platform. For each library, we must either port it to every platform or rewrite it in Java. In the case of `POV-Ray`, we have decided to leave it in C because it has already been ported to many platforms. In the future, we expect the number of quality Java libraries will increase dramatically.

Fault tolerance: The fault tolerance comes from the tree structure of the ATLAS programming model. For any stolen thread, we refer to the subtree rooted at that thread as a subcomputation. Any subcomputation that does not complete will time out and be recomputed from a checkpoint. Each subcomputation maintains its own checkpoint file, so checkpointing is fully distributed. Moreover, each subcomputation is a transaction, so that its side effects are made visible outside the subtree in an atomic action that can be undone [2].

Adaptive parallelism: The work-stealing scheduler and the tree-structured programming model allow programs to run on a set of compute servers that grows and shrinks over time. The owner of a compute server can set the policy to determine when the compute server is idle. When idle, the compute server automatically joins the execution of a parallel program by work stealing. When the compute server is no longer available, all of its subcomputations are easily moved to another compute server. Moving a subcomputation requires only updating the information linking the subcomputation to its parent and child subcomputations [2].

Safety: The safety of ATLAS depends on the safety of Java and of the native libraries. We take both of these for granted. In the case of `POV-Ray`, we were careful to limit the interface of the `POV-Ray` library, so that all calls to external functions, much like kernel crossings, are carefully checked. We limit local file access to `/tmp`, which we use for caching and temporary files. We limit network access to the predefined protocols for stealing and file access. Complete coverage of security and safety is beyond the scope of this paper.

Anonymity: We provide a relatively weak form of anonymity. A host executing a stolen subtree cannot determine the original source of the computation, since all results and file accesses are performed through the manager (using encrypted URLs). We believe it is sufficient in practice to trust the managers but not the hosts, much like trusting routers but not clients. It is possible using physical security devices to build a network of trusted managers, but we do not intend to do so. Finally, users that really care can prevent computations from moving outside their top-level manager, much like a firewall router. Note that the manager could still steal work from outside.

Hierarchy: The manager hierarchy achieves our goal of giving preference for idle resources to local users. For example, an application started on our machines would spread first to machines in our group, then to the Computer Science Division, then to the College of Engineering, then UC Berkeley and finally to other institutions. Equivalently, managers only steal up the hierarchy if their entire subtree is underutilized. This encourages using local bandwidth over (scarce) wide-area bandwidth.

Ease of use: The interface, which is not completed, has two parts. First, users submit jobs to their local manager using a special shell. Second, users can check on their application through a web page generated by this shell. In the case of `POV-Ray`, the web page contains the partially complete image. The page will also provide status on the number of active subtrees, the running time, and average concurrency. For `POV-Ray`, we are also building a web page that uses HTML forms and CGI to submit jobs.

Reasonable performance: The overhead in the initial prototype version is quite high, but it can be amortized for large threads, as in the `POV-Ray` example. We were also able to obtain near linear speedups for `POV-Ray`.

6 Related work

A number of systems have been developed to support parallel computation on distributed networks of workstations, but none provide all of the properties required for parallel computing on a global scale. The DIB system [8] uses a heuristic form of work stealing to schedule backtrack-search applications on a network of workstations. The Benevolent Bandit Laboratory (BBL) [7] provides adaptive parallelism for applications running on a network of PCs and is probably the first example of a system that provides adaptive parallelism. The Piranha system [9] provides adaptive parallelism for Linda [6] programs running on a network of workstations. (The authors of Piranha appear to have coined the term "adaptive parallelism.") The Sam system [12] provides transparent fault tolerance for shared-memory parallel programs running on a network of workstations. While some of these systems provide adaptive parallelism or fault tolerance, none provide both. In addition, none of these systems support safe or heterogeneous execution, and lacking any notion of hierarchy, the scalability of all of these systems is questionable. The Cilk-NOW system [2], from which much of Atlas is derived, provides both adaptive parallelism and fault tolerance, but it does not support safe or heterogeneous execution, and it does not have any notion of hierarchy. It is exactly these shortcomings that motivates Atlas.

There are at least two interesting cases of large-scale heterogeneous applications. On the large scale, the Eclipse system [13] was used to run a Monte Carlo simulation using 192 distributed SPARC, RS6000, and i860 processors. On the global scale, the factoring of RSA-129 involved a huge number of machines spread across many countries [10]. In both cases, significant administration was required to run the application. Both of these applications fit well within the Atlas programming model, and Atlas would allow the global-scale execution of these applications with almost no administrative effort.

7 Limitations and future work

In this section, we conclude with a discussion that addresses both limitations in the proposed Atlas architecture and limitations in the current prototype Atlas implementation. We also indicate current plans for future work to lift some of these limitations.

Much the Atlas system's functionality derives from structure in the Atlas programming model which is limited to "tree-structured" computations such as divide-and-conquer, backtrack search, branch-and-bound, and game-tree search. Specifically, the Atlas architecture incorporates policies, such as work stealing and "transactional subcomputations," that are specialized for tree-structured computations. Generalizing the Atlas programming model while still preserving the important properties, such as scalability, adaptive parallelism, and fault tolerance, remains a daunting open problem.

The current prototype Atlas system has a number of limitations that we plan to address in future work.

- The current prototype has no shared variables among threads. Fortunately, our model enables a form of weak consistency (dag consistency [3]) that we will incorporate into future versions.
- The current global file system is read only. We are currently building a coherent read/write version.

- The current prototype has not been tested on a heterogeneous cluster, though we already use a network representation that is machine independent.
- Fault tolerance interacts with shared variables and files in a way that would ideally be solved with transactions. We are actively investigating the combination of shared state and fault tolerance.

The initial prototype implementation of the ATLAS system indicates that a system for global applications and resource sharing is viable. In particular, we enumerate and propose solutions for a wide variety of challenges including fault tolerance, heterogeneity, safety, and scalability.

Acknowledgments

We would like to thank Peter Mattis and Spencer Kimball for help with the `POV-Ray` example. Thanks also to Amin Vehdat who wrote the global file-system library. Finally, we thank Charles Leiserson of MIT whose work on the Cilk project is the basis for much of the ATLAS architecture.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996. More information about Java and the Java virtual machine is available on the World-Wide Web at `http://java.sun.com`.

[2] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[3] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 132–141, Honolulu, Hawaii, April 1996.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995. More information about Cilk, including papers and software releases, is available on the World-Wide Web at `http://theory.lcs.mit.edu/~cilk`.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[6] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[7] Robert E. Felderman, Eve M. Schooler, and Leonard Kleinrock. The Benevolent Bandit Laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 7(2):303–311, February 1989.

[8] Raphael Finkel and Udi Manber. DIB—a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[9] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 417–427, Washington, D.C., July 1992.

[10] S. Levy. Wisecrackers. *Wired*, 4(3), March 1996.

[11] POV-Ray. `http://www.povray.org`.

[12] Daniel J. Scales and Monica S. Lam. Transparent fault tolerance for parallel applications on networks of workstations. In *Proceedings of the USENIX 1996 Annual Winter Technical Conference*, San Diego, California, January 1996.

[13] V. S. Sunderam and Vernon J. Rego. EcliPSe: A system for high performance concurrent simulation. *Software—Practice and Experience*, 21(11):1189–1219, November 1991.

[14] A. Vehdat and T. E. Anderson, 1996. Personal communication.