Irregular Parallel Algorithms in Java*

Brian Blount[1], Siddhartha Chatterjee[1], and Michael Philippsen[2]

[1] The University of North Carolina, Chapel Hill, NC 27599-3175, USA.
[2] The University of Karlsruhe, 76128 Karlsruhe, Germany.

Abstract. The nested data-parallel programming model supports the design and implementation of irregular parallel algorithms. This paper describes work in progress to incorporate nested data parallelism into the object model of Java by developing a library of collection classes and adding a `forall` statement to the language. The collection classes provide parallel implementations of operations on the collections. The `forall` statement allows operations over the elements of a collection to be expressed in parallel. We distinguish between `shape` and `data` components in the collection classes, and use this distinction to simplify algorithm expression and to improve performance. We present initial performance data on two benchmarks with irregular algorithms, EM3d and Convex Hull, and on several microbenchmark programs.

1 Introduction

Software technology for scalable parallel computers lags behind the rapid developments in the hardware for such systems. Current programming languages on these systems often expose architecture-specific details to the programmer, making it difficult to develop scalable and portable software. Software standards have emerged for dense array codes in the form of data-parallel languages such as High Performance Fortran [16] (HPF) and message passing libraries such as MPI [28]. However, many irregular algorithms, such as those used in sparse matrix solvers and geometric problems, can be difficult to express in these systems.

Scientific applications continue to grow in the sophistication of the data structures and algorithms they use. Techniques such as adaptive unstructured meshes [4] in computational fluid dynamics (CFD), hierarchical n-body simulations [10, 17, 22, 26], and supernodal sparse Cholesky [2] and LU [21] factorization are difficult to write in an architecture-independent manner in traditional performance-oriented languages. One programming model capable of expressing such irregular algorithms is *nested data parallelism* [5]. While data parallelism allows the expression of parallelism through operations over the elements of a collection, nested data parallelism allows the elements of a collection to themselves be collections. This allows parallelism to be expressed on complex structures, such as graphs, trees, and sparse matrices.

We are investigating the incorporation of nested data parallelism into the object model of Java [1]. We choose Java for several reasons. First, there is a growing interest

* This research is supported in part by the National Science Foundation under grant CCR-9711438.

in Java for scientific computing [14, 15, 19]. Second, Java programs may utilize efficient numerical libraries through `native` methods. Third, the object-oriented features of Java are useful in developing modular and reusable software components. Finally, Java is portable; a Java program can run on any machine that implements a Java Virtual Machine (JVM).

We enhance Java in two ways to support nested data parallelism: we add a library of collection classes and a `forall` statement. The collection classes provide parallel operations on the collections themselves (aggregate parallelism), and the `forall` statement allows the expression of parallelism on operations over the elements of a collection (elementwise parallelism). To support the nested data-parallel model, members of collections can themselves be collections, and the `forall` statements may be nested. Each collection class consists of three members: a *data* object, a *shape* object, and a *map* object. The encapsulation of these separate components allows for cleaner expression of operations on the collections and enables program optimizations.

The remainder of the paper is organized as follows. Section 2 describes our framework. Section 3 presents performance results. Section 4 discusses related work. Section 5 presents conclusions and future work.

2 Nested Data Parallelism Framework

We introduce two features into Java to support nested data parallelism: a library of collection classes and a `forall` statement. The collection classes provide parallel implementations of aggregate operations, such as reductions and parallel prefix [5], and also provide functionality similar to the collection classes in JDK 1.2 [29]. The `forall` statement allows operations over the elements of a collection to be expressed in parallel. To support nested data parallelism, collections may be members of other collections and `forall` statements may be nested.

A pre-compiler [24] translates the `forall` statements into multi-threaded Java code. Because the translated code uses threads as the source of parallelism, we prohibit the explicit use of threads in our extended language. The transformed code uses a package of classes to manage the threads at runtime. The library of collection classes also uses this package to implement its parallel aggregate operations. We derived our translation from one previously presented by one of the authors [24]. Our current version differs in two ways. First, it supports nested `forall` statements more efficiently. Second, it uses Java as a target language instead of Pizza [23]. The original translation utilized the closure mechanism of Pizza, which we replace with *inner classes* [1]. We present the translation by examining the simple example of a `forall` statement that initializes an array.

```
1   forall (int i = 0; i < n; i++) a[i] = i;
```

Our approach transforms the above example into the code shown below. We use a work pile and worker threads. Worker threads start at program invocation and continuously perform any work placed on the work pile. An (anonymous) inner class encapsulates the work of a `forall` loop in its `exec` method, as shown in lines 2–8. The inner class is a subclass of `ForallWork`, which contains four variables that are initialized

in the constructor (line 2): the upper and lower bounds of the work it must perform (`lb` and `ub`), the step size of the loop (`step`), and a handle to its ForallController (`fac`), a runtime support class. The `exec` method first checks in line 4 to see if the work object should be split. This is performed by the `ForallController` and is the method by which multiple work objects are created. If the object is split, the `ForallController` places the two resulting objects on the work pile. If the object is not split, the `exec` method performs the work in line 6. Finally, in line 7, the work object informs the `ForallController` that it is complete. Line 1 of the code instantiates a `ForallController` object. Line 2 creates a work object and line 9 places it on the work pile. When all of the work objects are complete, the controller's `finalBarrier` method in line 10 returns.

```
 1   ForallController _fac = new ForallController();
 2   ForallWork _work = new ForallWork(0, n, 1, _fac) {
 3     public void exec() {
 4       if (!this.fac.split(this))
 5         for(int i = this.lb; i < this.ub; i += this.step)
 6           a[i] = i;
 7       this.fac.workDone();
 8   } };
 9   WorkPile.doWork(_fac, _work);
10   _fac.finalBarrier();
```

The translation above ignores exceptions that might be thrown inside the body of the `forall`, *e.g.*, if $a.length < n$ in the example above. If this occurs, all work objects should cease execution and the exception should be thrown in the method that the forall statement occurred in. This is handled by maintaining information in the `ForallController`. The version of `exec` below handles exceptions and replaces lines 3–8 in the code block above. Any exception thrown inside the loop body is caught and stored in lines 6–8. All the work threads are signaled to stop execution through the setting of `breakFlag`. The `finalBarrier` method throws the stored exception when all work objects are complete.

```
 1    public void exec() {
 2      if (!this.fac.breakFlag) {
 3        if (!this.fac.split(this)) {
 4          for(int i = this.lb; i < this.ub; i += this.step) {
 5            if (this.fac.breakFlag) break;
 6            try { a[i] = i; } catch(Throwable _ex) {
 7              this.fac.caughtException = _ex;
 8              this.fac.breakFlag = true;
 9      } } } }
10      this.fac.workDone();
11    }
```

Note that the semantics for a thrown exception are different for a `forall` loop than for a `for` loop. If an exception is thrown inside iteration i of a `for` loop, all iterations before i are guaranteed to have completed, and all iterations after i are guaranteed to

have never begun execution. If an exception is thrown inside iteration i of a `forall` loop, there are no guarantees regarding which iterations have executed. This may complicate handling exceptions for a programmer.

Our translation supports nested `forall` statements differently than the original. In the original, the `finalBarrier` method simply slept, waiting on other worker threads to complete work. When nested `forall` statements occur, all the worker threads may enter `finalBarrier` methods, leaving no worker threads to execute any work. At this point, a low-priority watchdog thread recognized the situation and spawned more worker threads. This approach has the disadvantage that the executing program may eventually have more worker threads than originally intended. Our translation solves this problem by modifying the `finalBarrier` method. If a thread enters this method and cannot proceed, it executes work off of the work pile until it can proceed.

The collections within our library resemble collections from previous work [8] and consist of three components: a data object, a shape object, and a map object. The data object is a one-dimensional array containing all the elements of a given collection in some arbitrary order. The shape object specifies the "shape" of the collection by defining a domain on which the collection can be indexed. The map object maps values from this index domain to locations in the data array. As an example, consider a matrix collection with r rows and c columns. Its data array is of length n, where $n \geq r \times c$. Its shape object is of type `MatrixShape`, and contains r and c as data. Its map function maps $\{(i, j) : 0 \leq i < r, 0 \leq j < c\}$ to $\{k : 0 \leq k < n\}$. Each component is a first-class object and can be manipulated and shared arbitrarily. The collections rely on sharing of data and performing actions "lazily" on index domains whenever possible.

Each collection class supports parallel aggregate operations, such as reductions and prefix sum [5]. The implementation of these parallel operations is similar to the translation scheme for the `forall` statement and follows a uniform strategy: encapsulate work in a work object, and encapsulate the state of the operation in a controller object. As an example, the reduction operation `sum` uses two classes, `ReduceSumController` and `ReduceSumWork`. A `ReduceSumWork` object encapsulates the work of the sum by performing the sum operation on a portion of the collection. Similar to a `ForallController`, a `ReduceSumController` object maintains the state of the sum operation. The state includes two pieces of information: whether the sum operation is complete, and the reduced value computed by each work object. Once all the work objects are complete, the controller uses the values the work objects calculated to determine the final sum.

3 Performance

We wrote two benchmarks, EM3d and Convex Hull, and several microbenchmarks to evaluate the performance of our implementation of nested data parallelism in Java. The microbenchmarks fit into two categories: programs that measure the performance of Java's concurrency primitives, and benchmarks that measure the performance of the parallel operations added to the language. Table 1 lists the environments we used for our measurements.

	UltraSparc 170 (US170)	UltraSparc (US)	HyperSparc (HS)	Sun Enterprise 3000 (SE)	SGI Onyx 2 (SGI)
CPU	Ultra-I	Ultra-IIi	HyperSPARC	UltraSPARC	MIPS R8000
Processors	1	1	4	4	4
Clock Rate	167 MHz	300 MHz	100 MHz	170 MHz	195 MHz
Memory	96 MB	248 MB	160 MB	384 MB	2176 MB
OS	Solaris 2.5.1	Solaris 2.6	Solaris 2.6	Solaris 2.5.1	IRIX 6.4
JVM	1.1.6, 1.2beta4-K	1.1.6, 1.2beta4p	1.1.6, 1.2beta4p	1.1.6, 1.2beta4-K	3.1 (Sun 1.1.5)

Table 1. Testing Environment (1.2beta4p stands for 1.2beta4production). JDK 1.2beta4 always performed better than 1.1.6, so we do not present the results for JDK 1.1.6. All JVMs are JIT-enabled. We present results for native threads, implemented by the operating system, and for green threads, implemented by the JVM.

Our translation of nested data parallelism realizes its concurrency through Java's thread interface. There are three operations that may introduce overhead: thread instantiation, lock use, and thread scheduling. Our implementation amortizes the first overhead by instantiating threads only once, at program invocation. The second overhead depends on the implementation of locks, and is discussed below. Both the operating system and Java's scheduling primitives (such as `yield`, `wait`, and `notify`) influence the third overhead. Since our model relies heavily on `wait` and `notify`, we need to examine their costs further.

Java provides locking through its `synchronized` keyword, which can modify a method or a block of code. Synchronized code obtains a lock before it begins execution and releases a lock when it terminates execution. Many of the work pile's methods are synchronized because the worker threads access a central work pile. To quantify the cost of obtaining a lock, we compare the cost of calling a method that is synchronized with the cost of calling a method that is not synchronized. Both methods take no arguments, return no values, and perform a trivial amount of work. The results are shown in Table 2. The locking mechanism introduces significant overhead. SGI's JVM performs better than Sun's, but its overhead is still large. Several projects have examined reducing this cost [3, 20], and JDK 1.2 shows improvement over JDK 1.1.6. However, this overhead may require some redesign of our runtime classes to reduce the number of synchronized methods.

We used three microbenchmarks to measure the performance of the parallel operations added to the language. The first program executes a single `forall` statement with 64 parallel iterates. Each iterate performs a `for` loop with 100,000 sequential iterations. The second program performs a `plus_reduce` operation on a vector of 2.5 million integers. The third program performs a `plus_scan` operation on a vector of 1 million integers.

The top portion of Table 3 presents the performance of these three programs. Parallel execution of the `forall` statement introduced an overhead of 10% to 40%, because of object creation and synchronized methods introduced by the transformation. The parallel `plus_reduce` operation introduced a 15% to 60% overhead. The greater

Machine	JVM	Native Threads				Green Threads			
		Sync (μs)	No Sync (μs)	Overhead (μs)	Ratio	Sync (μs)	No Sync (μs)	Overhead (μs)	Ratio
US 170	1.2b4k	1.38	0.07	1.31	19.71	1.52	0.07	1.45	21.71
US	1.2b4p	0.62	0.01	0.61	62.00	—	—	—	—
HS	1.2b4p	0.85	0.03	0.82	28.33	—	—	—	—
SE	1.2b4k	1.52	0.06	1.46	25.33	1.50	0.07	1.43	21.43
SGI	3.1	1.13	0.22	0.91	5.14	0.58	0.21	0.37	2.76

Table 2. Overhead of synchronized methods. Both the synchronized method and the non-synchronized method were called 2048 times and the average cost was taken. We placed a small amount of work inside the methods to verify that the method calls were not optimized away. Performance is similar for synchronized blocks.

overhead may be due to the fact the reduce controller performs more work than the forall controller. The overhead of the parallel `plus_scan` operation was approximately 300%. This is because the parallel operation performs a three step algorithm, and each work object is placed on the work pile twice. The implementation of this operation must be re-evaluated to decrease the overhead. On machines with one processor, there were no parallel improvements. On the machines with four processors, near-linear speed up was seen for most operations. Some further speedup was seen with additional threads. The optimal number of threads varies among the machines, and requires further investigation.

Several peculiarities are seen in the data. On the Enterprise, the operating system scheduled the threads sequentially for the `forall` microbenchmark, so no speedup was seen. We are currently investigating the reason for this anomaly. Also, multiple threads did not provide speedup on the SGI until 4 or 8 threads were used. This is probably due to the scheduling algorithm of the operating system. While our implementation should be efficient independent of the scheduling algorithm used, it does depend on the minimum expectation of threads being scheduled on multiple processors when run on a multiple processor machine.

We have thus far implemented two irregular algorithms in our extended Java language: EM3d and Planar Convex Hull. The EM3d kernel [13] (EM3d) models the propagation of electromagnetic waves through objects in three dimensions. We include it in our benchmark suite because of its popularity and the availability of a number of implementations of it on various systems. The implementation of this algorithm in our extended language consists of two nested `forall` statements and an aggregate `plus_reduce` operation. The planar convex hull problem (PCH) is as follows: given n points in the plane, determine the points that lie on the perimeter of the smallest convex region containing them. The problem has many applications ranging from computer graphics to statistics. We implement the quickhull algorithm [6] for this problem. Our program uses both `forall` statements and aggregate operations.

The bottom of Table 3 presents the results for these two algorithms. EM3d executed 100 iterations on a graph with 4000 H nodes and 4000 E nodes, where each node had 30 neighbors. PCH executed 100 iterations on a set of 100000 points contained inside

Program	Machine	JVM	Sequential Version (s)	Parallel Version Number of Threads 1 (s)	2 (s)	4 (s)	8 (s)	16 (s)
Forall	US170	1.2b4k	0.45	0.62	0.68	0.66	0.65	0.66
	US	1.2b4p	0.25	0.27	0.29	0.31	0.32	0.32
	HS	1.2b4p	0.61	0.74	**0.37**	**0.19**	**0.23**	**0.23**
	SE	1.2b4k	0.44	0.61	0.58	0.58	0.57	0.57
	SGI	3.1	1.74	1.79	1.79	**0.89**	**0.61**	**0.56**
Plus-Reduce	US170	1.2b4k	1.08	1.71	1.74	1.77	1.80	1.74
	US	1.2b4p	0.23	0.30	0.34	0.35	0.34	0.35
	HS	1.2b4p	0.86	1.01	**0.51**	**0.26**	**0.30**	**0.31**
	SE	1.2b4k	1.04	1.64	**0.92**	**0.55**	**0.49**	**0.51**
	SGI	3.1	0.90	1.26	1.41	**0.80**	**0.53**	**0.52**
Plus-Scan	US170	1.2b4k	0.49	1.50	1.56	1.61	1.79	1.61
	SE	1.2b4k	0.49	1.44	0.75	0.52	0.50	0.50
	SGI	3.1	0.41	1.21	1.31	1.27	0.57	0.48
EM3d	US170	1.2b4k	1.21	**1.09**	1.22	1.22	1.27	1.23
	US	1.2b4p	0.26	0.38	0.28	0.28	**0.23**	**0.24**
	HS	1.2b4p	0.53	0.56	**0.29**	**0.17**	**0.20**	**0.21**
	SE	1.2b4k	1.07	**1.03**	**0.60**	**0.45**	**0.43**	**0.44**
	SGI	3.1	1.05	1.11	1.24	1.33	1.58	1.57
PCH	US170	1.2b4k	2.80	2.91	3.08	2.95	3.02	3.02
	SE	1.2b4k	2.77	2.85	**2.37**	**2.03**	**2.04**	**2.19**
	SGI	3.1	2.86	2.97	3.09	3.02	8.33	3.23

Table 3. Performance of microbenchmark and benchmark programs. Each program executed 100 iterations, and the average cost is shown. Times are in seconds. Results where parallel performance is better than sequential performance are shown in boldface.

a circle of radius 10000. Of the 100000 points, 63 fell on the hull. Parallel execution of the two programs introduces almost no overhead because the amount of work in the algorithm hides the overhead introduced in parallel execution. The EM3d performance resembles the performance of the `forall` benchmark. Because the inner forall loop is small (only 30 iterations), we run it sequentially. The same applies to its reduce operation. The PCH benchmark shows little or no speedup on any of the machines. This may be because much of the parallelism in the algorithm is not exploited. The algorithm recursively calls itself twice. The two calls are independent and may be executed in parallel. Because there is no way to express this easily in the language, the two calls are run sequentially.

4 Related Work

NESL [7] is a functional language designed to support nested data parallelism. It is a strongly typed, polymorphic language with a type inference system, which makes it possible to write generic functions that work on any type of sequence. Because NESL

is purely functional, it suffers the usual problems when updating subsets of aggregate variables, and it makes it impossible for users to manage memory use. Also, NESL's runtime system is incompatible with other languages.

The Amelia [27] project introduces nested data parallelism into C++. We use many of the same techniques as the Amelia project, but do not suffer the problems with unrestricted pointer semantics of C++. The pC++ [11] language defines a new programming model called the "distributed collection model." This model is not quite data-parallel and does not support nested data parallelism. Since collections may not be members of other collections, the data types available to a programmer are somewhat limited. The ICC++ [12] language is an extension to C++ that supports concurrency through *conc* blocks and collection objects. The implementation uses advanced compiler optimizations and a custom runtime system to gain high performance. Any two statements within a *conc* block may be executed concurrently if there are no data dependences between the two.

High Performance Fortran (HPF) [16] defines a collection-oriented parallel language that supports one collection type: the multi-dimensional array. Parallelism in HPF programs comes through the use of a `forall` construct and special parallel operations on arrays, such as array permutation and reduction. The language is not oriented towards nested data parallelism, although pointer types can be used to support nested collections [25], and certain routines in the standard HPF library can be used to support nested data-parallel operations.

Titanium [30] is a dialect of Java designed for SPMD parallel programming. While Titanium uses Java as a base, it adds many extensions to the language, such as immutable classes, parallel constructs, and multi-dimensional arrays. However, Titanium provides no direct support for nested data parallelism and does not use Java as a target language. Hummel *et al.* [18] explore mechanisms and propose an API for writing SPMD programs in Java, targeting a parallel Java runtime system on the IBM POWERparallel System SP machine. Their focus is on flat array-based data parallelism.

5 Conclusions and Future Work

This paper presents a method to incorporate nested data parallelism into Java. It describes a scheme for implementing collection classes, and a method for implementing `forall` statements and aggregate operations using inner classes and threads. The microbenchmarks examined demonstrate that these implementations can achieve good performance. The two irregular algorithms demonstrate that our model is well-suited for expressing most types of parallelism.

We are pursuing several future directions for this work. First, we are investigating two ways to reduce the overhead of the parallel operations. First, we want to modify the implementation of the `prefix sum` operation to reduce its overhead to the level of the other operations. Second, we may use multiple work piles to reduce the number of worker threads waiting on the one work pile. This model would provide one work pile for each worker thread, and the work piles could perform work stealing similar to the method implemented in Cilk [9].

Second, we are experimenting with various techniques for determining the optimal granularity for work objects. Our current strategy simply defines the optimal granularity as a constant number of iterations, and divides the work objects until they contain fewer than this number of iterations. A more reasonable approach would factor both number of iterations and amount of work performed in an iteration in determining optimal granularity.

Third, we are considering adding another parallel construct to our language: the `spawn` keyword. This would allow a limited amount of task parallelism, such as the recursive calls to quickhull described above, to be expressed easily. A `forall` statement is appropriate to express parallelism over tasks when the number of tasks is unknown. However, when the number of parallel tasks is known, such as the recursive calls in quicksort, quickhull, and Strassen's matrix multiplication, the `spawn` keyword provides a much simpler mechanism to express the parallelism. This will have substantial ramifications for semantics and implementations that need to be carefully examined.

Finally, we are examining other irregular computations, such as conjugate gradient, supernodal sparse Cholesky and LU factorizations, and CFD problems using adaptive unstructured meshes. These algorithms will provide a better suite of test cases to evaluate the expressiveness and performance of our framework.

References

1. K. Arnold and J. Gosling. *The Java^{TM} Programming Language*. The Java^{TM} Series. Addison-Wesley Publishing Company, 1996.
2. C. C. Ashcraft. The domain/segment partition for the factorization of sparse symmetric positive definite matrices. Enginering Computing & Analysis Technical Report ECA-TR-148, Boeing Computer Services, Seattle, WA, Nov. 1990.
3. D. F. Bacon *et al.* Thin locks: featherweight synchronization for Java. In *Proc. PLDI'98*, pages 258–268, 1998.
4. R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13:437–452, 1994.
5. G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, Cambridge, MA, 1990.
6. G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Apr. 1993. Updated version of CMU-CS-92-103, January 1992.
7. G. E. Blelloch *et al.* Implementation of a portable nested data-parallel language. *JPDC*, 21(1):4–14, Apr. 1994.
8. B. L. Blount and S. Chatterjee. An evaluation of Java for numerical computing. In *Proc. ISCOPE'98*, Dec. 1998. LNCS 1505, pp. 35-46, Springer Verlag.
9. R. D. Blumofe *et al.* Cilk: An efficient multithreaded runtime system. In *Proc. PPoPP'95*, pages 207–216, Santa Barbara, CA, July 1995. ACM.
10. J. A. Board Jr. *et al.* Scalable variants of multipole-accelerated algorithms for molecular dynamics applications. Technical Report TR94-006, Department of Electrical Engineering, Duke University, Durham, NC, 1994.
11. F. Bodin *et al.* Implementing a parallel C++ runtime system for scalable parallel systems. In *Proc. SC'93*, pages 588–597, November 1993.
12. A. A. Chien and J. Dolby. ICC++: A C++ dialect for high-performance parallel computation. In *Proc. ISOTAS'96*, Mar. 1996.

13. D. E. Culler *et al.* Parallel programming in Split-C. In *Proc. SC'93*, pages 262–273, Nov. 1993.
14. G. C. Fox. Java for high performance scientific and engineering computing. http://www.npac.syr.edu/projects/javaforcse/.
15. D. B. Gannon. High Performance Java. http://www.extreme.indiana.edu/hpJava/index.html.
16. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1993.
17. Y. Hu, S. L. Johnsson, and S.-H. Teng. High Performance FORTRAN for Highly Irregular Problems. In *Proc. PPoPP'97*, pages 13-24, June 1997.
18. S. F. Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 9(6):621–631, June 1997. Special issue on Java for computational science and engineering—simulation and modeling.
19. Java Grande Forum. The Java Grande Forum charter document. http://www.npac.syr.edu/javagrande/jgfcharter.html.
20. A. Krall and M. Probst. Monitors and Exceptions: How to implement Java efficiently. *Concurrency: Practice and Experience*, 10(11):837–850, Sept. 1998.
21. X. S. Li. *Sparse Gaussian Elimination on High Performance Computers*. PhD thesis, Department of Computer Science, University of California at Berkeley, Berkeley, CA, Sept. 1996. Available as technical report CSD-96-919.
22. L. S. Nyland, J. F. Prins, and J. H. Reif. A data-parallel implementation of the fast multipole algorithm. In *Proc. DAGS'93*, pages 111–122, Hanover, NH, June 1993.
23. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. POPL'97*, Jan. 1997.
24. M. Philippsen. Data parallelism in Java. In J. Schaefer, editor, *High Performance Computing Systems and Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1998.
25. J. Prins, S. Chatterjee, and M. Simons. Expressing irregular computations in modern Fortran dialects. In D. O'Hallaron, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 1–16. Springer, 1998. LNCS 1511.
26. K. E. Schmidt and M. A. Lee. Implementing the fast multipole algorithm in three dimensions. *Journal of Statistical Physics*, 63(5/6), 1991.
27. T. J. Sheffler and S. Chatterjee. An object-oriented approach to nested data parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 203–210, McLean, VA, Feb. 1995.
28. M. Snir *et al. MPI: The Complete Reference.* MIT Press, 1996.
29. The Java collections framework. http://java.sun.com/products/jdk/1.2/docs/api/index.html.
30. K. Yelick *et al.* Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11):825–836, Sept. 1998.