Irregular Parallel Algorithms in Java*

Brian Blount, Siddhartha Chatterjee, and Michael Philippsen

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 [17] (HPF) and message passing libraries such as MPI [22]. 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 [10,18,23,27], 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.

* This research is supported in part by the National Science Foundation under grant CCR-9711438.
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 in Java for scientific computing [14–16]. 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.

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 [25] 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 [25]. 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 [24]. 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 5. Finally, in line 6, 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 8 places it on the work pile. When all of the work objects are complete, the controller’s finalBarrier method in line 9 returns.

```java
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) a[i] = i;
6              this.fac.workDone();
7          }
8          WorkFile.doWork(_fac, _work);
9          _fac.finalBarrier();
10  }
```

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 3–7 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.

```java
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              }
11          }
12          this.fac.workDone();
13      }
```

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 \texttt{forall} loop, there are no guarantees regarding which iterations have executed. This may complicate handling exceptions for a programmer.

Our translation supports nested \texttt{forall} statements differently than the original. In the original, the \texttt{finalBarrier} method simply slept, waiting on other worker threads to complete work. When nested \texttt{forall} statements occur, all the worker threads may enter \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{sum} uses two classes, \texttt{ReduceSumController} and \texttt{ReduceSumWork}. A \texttt{ReduceSumWork} object encapsulates the work of the sum by performing the sum operation on a portion of the collection. Similar to a \texttt{ForallController}, a \texttt{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.

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 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.

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

Table 1. Testing Environment (1.2beta4p stands for 1.2beta4 production). JDK 1.2 beta4 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.
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.

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 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 single processor machine.

We have thus far implemented two irregular algorithms in our extended Java language: EM3d and Planar Convex Hull. The EM3d kernel [12] (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
<table>
<thead>
<tr>
<th>Program</th>
<th>Machine</th>
<th>JVM</th>
<th>Sequential Version (s)</th>
<th>Parallel Version Number of Threads</th>
<th>(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
</tr>
<tr>
<td>Forall</td>
<td>US170</td>
<td>1.25-4k</td>
<td>0.45</td>
<td>0.62</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>CS</td>
<td>1.25-4p</td>
<td>0.25</td>
<td>0.27</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>FS</td>
<td>1.25-4p</td>
<td>0.64</td>
<td>0.74</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td>SE</td>
<td>1.25-4k</td>
<td>0.44</td>
<td>0.64</td>
<td>0.68</td>
</tr>
<tr>
<td>SGI</td>
<td>3.1</td>
<td></td>
<td>1.74</td>
<td>1.79</td>
<td>1.79</td>
</tr>
<tr>
<td>Plus-Reduce</td>
<td>US170</td>
<td>1.25-4k</td>
<td>1.08</td>
<td>1.71</td>
<td>1.74</td>
</tr>
<tr>
<td></td>
<td>FS</td>
<td>1.25-4p</td>
<td>0.86</td>
<td>1.01</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>SE</td>
<td>1.25-4k</td>
<td>1.04</td>
<td>1.64</td>
<td>0.52</td>
</tr>
<tr>
<td>SGI</td>
<td>3.1</td>
<td></td>
<td>0.90</td>
<td>1.26</td>
<td>1.41</td>
</tr>
<tr>
<td>Plus-Scan</td>
<td>US170</td>
<td>1.25-4k</td>
<td>0.49</td>
<td>1.50</td>
<td>1.56</td>
</tr>
<tr>
<td></td>
<td>SE</td>
<td>1.25-4k</td>
<td>0.49</td>
<td>1.44</td>
<td>0.75</td>
</tr>
<tr>
<td>SGI</td>
<td>3.1</td>
<td></td>
<td>0.41</td>
<td>1.21</td>
<td>1.31</td>
</tr>
<tr>
<td>EM3d</td>
<td>US170</td>
<td>1.25-4k</td>
<td>1.21</td>
<td>1.69</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>US</td>
<td>1.25-4p</td>
<td>0.26</td>
<td>0.38</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>FS</td>
<td>1.25-4p</td>
<td>0.53</td>
<td>0.56</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>SE</td>
<td>1.25-4k</td>
<td>1.07</td>
<td>1.08</td>
<td>0.60</td>
</tr>
<tr>
<td>SGI</td>
<td>3.1</td>
<td></td>
<td>1.06</td>
<td>1.11</td>
<td>1.24</td>
</tr>
<tr>
<td>PCH</td>
<td>US170</td>
<td>1.25-4k</td>
<td>2.80</td>
<td>3.91</td>
<td>3.08</td>
</tr>
<tr>
<td></td>
<td>SE</td>
<td>1.25-4k</td>
<td>2.77</td>
<td>2.85</td>
<td>2.37</td>
</tr>
<tr>
<td>SGI</td>
<td>3.1</td>
<td></td>
<td>2.86</td>
<td>2.97</td>
<td>4.09</td>
</tr>
</tbody>
</table>

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.

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 one hundred iterations on a set of 100000 points contained inside 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 processors. 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 [28] 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. Any two statements within a conc block may be executed concurrently if there are no data dependencies between the two. ICC++ also supports collection objects.

High Performance Fortran (HPF) [17] 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 [26], 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. [19] 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 each work object. 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.

Fourth, 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


