The Journal of Supercomputing, 28, 91–117, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Run-Time Support for the Automatic Parallelization of Java Programs

BRYAN CHAN chanb@eecg.toronto.edu
The Edward S. Rogers Sr., Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada M5S 3G4

TAREK S. ABDELRAHMAN tsa@eecg.toronto.edu
The Edward S. Rogers Sr., Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada M5S 3G4

Abstract. We describe and evaluate a novel approach for the automatic parallelization of programs that use pointer-based dynamic data structures, written in Java. The approach exploits parallelism among methods by creating an asynchronous thread of execution for each method invocation in a program. At compile time, methods are analyzed to determine the data they access, parameterized by their context. A description of these data accesses is transmitted to a run-time system during program execution. The run-time system utilizes this description to determine when a thread may execute, and to enforce dependences among threads. This run-time system is the main focus of this paper. More specifically, the paper details the representation of data accesses in a method and the framework used by the run-time system to detect and enforce dependences among threads. Experimental evaluation of an implementation of the run-time system on a four-processor Sun multiprocessor indicates that close to ideal speedup can be obtained for a number of benchmarks. This validates our approach.

Keywords: automatic parallelization, parallelizing compilers, Java optimizations, run-time parallelization, task-level parallelism

1. Introduction

There has been considerable research during the past decade on parallelizing compilers and automatic parallelization of programs. Traditionally, this research focused on ‘‘scientific applications’’ that consist of loops and array references, typical of Fortran programs [2, 11]. Regrettably, this focus has limited the widespread use of automatic parallelization in industry, where the majority of programs are written in C, C++, or more recently in Java. These programs extensively use pointer-based dynamic data structures such as linked lists and trees, and often use recursion. Such features make it difficult to directly utilize parallelizing compiler technology developed for array structures and simple loops.

In this paper, we describe and evaluate a novel approach for the automatic parallelization of programs that use pointer-based dynamic data structures, written in Java. The approach exploits parallelism among methods by creating an asynchronous thread of execution for each method invocation in a program. The novelty of our approach stems from its use of combined compile time and run-time analyzes to automatically detect dependences and exploit parallelism among threads.

Methods in a sequential program are analyzed at compile time to determine the data they access, parameterized by methods' contexts. A description of these data accesses is transmitted to a run-time system during program execution. The run-time system uses this description to determine when a thread corresponding to an invoked method may execute, and to detect and enforce dependences among executing threads.

On the one hand, the proposed approach leads to increased run time overhead. On the other hand, it facilitates automatic parallelization of pointer-based programs, where compile-time-only approaches have been limited, mostly by the lack of precise alias and/or pointer analyses [9, 12]. We have implemented the approach in the *zJava* (pronounced "zed Java") system at the University of Toronto. Our experimental evaluation of the system on a 4-processor Sun Ultra machine indicates that scaling performance can be obtained for a number of benchmarks, which validates our approach.

The goal of this paper is to describe the run-time component of the *zJava* system. More specifically, the paper describes the symbolic access path notation and data access summaries, and how they are used to capture and represent data accesses in a method. The paper then describes the framework used at run time to create, execute, and synchronize threads. In particular, the paper describes the framework used to detect dependences among running threads, and to enforce sequential execution order.

The remainder of this paper is organized as follows. Section 2 gives an overview of the *zJava* system. Section 3 describes symbolic access paths and data access summaries. Section 4 describes the *zJava* run-time system in details. Section 5 presents our experimental evaluation of the system. Section 6 describes related work. Finally, Section 7 gives concluding remarks.

2. The *zJava* system

2.1. Model of parallelism and data sharing

The *zJava* system executes sequential Java programs, automatically extracting, packaging and synchronizing parallelism among methods. The main method of a program is considered the main thread and it starts executing sequentially. For each method invocation, an independent thread is created to asynchronously execute the body of the method. This thread may run on a separate processor, concurrently with the thread that created it, and with other threads in the system. It may in turn create child threads by invoking more methods. In general, the execution of the program may be viewed as a set of threads executing concurrently, with each thread sequentially executing the body of its associated method, and creating more threads whenever it invokes methods. A thread terminates when it reaches the end of its method. The program terminates when all threads terminate.

Threads share data in two ways. First, the actual parameters of a method invocation become input to the new thread, making it possible for a parent thread to pass values to its child threads. Second, threads may share data by accessing (reading/writing) data in the shared memory. In a Java program, class field variables are accessible by all methods of a class, and thus are shared by all threads executing these methods. In addition, threads may also share dynamically allocated data since a method may pass references to this data to other methods. In general, threads share data if they access the same data at some address in shared memory.

The flow of data in the sequential execution of the program results in data dependences among methods. Hence, synchronization of corresponding threads is necessary to preserve program correctness. The *zJava* system preserves the program's sequential semantics by enforcing serial execution order for threads performing conflicting operations on the same data. In other words, threads that write the same shared data must be executed in the same order in which their corresponding methods execute in the sequential program. Similarly, serial execution order is preserved between a thread that writes data and another thread that reads the same data. Threads accessing different data, or only reading the same data, may execute concurrently.

2.2. *System overview*

The *zJava* system extracts parallelism out of sequential Java programs and executes the resulting parallel programs on top of a standard Java Virtual Machine (JVM). A high-level overview of the system is shown in Figure 1. It consists of two main components: a compiler and a run-time system. The compiler analyzes the input sequential program to determine how shared variables and objects are used by every method in the program. The compiler captures this information in the form of symbolic access paths, and then collects these paths into a data access summary for

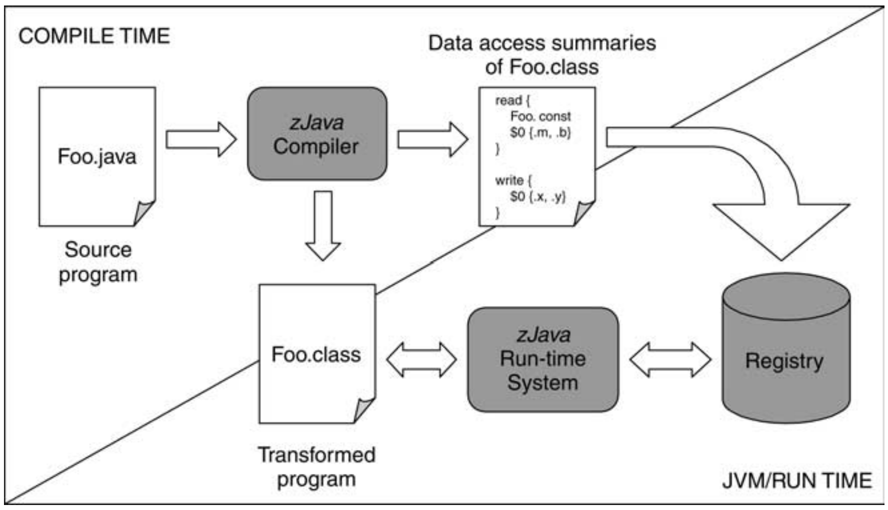


Figure 1. Overview of the *zJava* parallelization system.

each method.[1] The compiler also restructures the input program into a parallel threaded program that contains calls to routines in the run-time system, which create, execute, and synchronize threads.

The run-time system receives the data access summaries for methods in a class when the class is loaded by the JVM. It stores these summaries in a set of data structures, which we refer to collectively as the registry. The run-time system contains code to compute run time data dependences among threads using the data access summaries stored in the registry. Threads are then synchronized according to these dependences.

3. Data access summaries

The *zJava* compiler associates with every method a data access summary. In this section, we first describe the symbolic access path notation, which is used to succinctly record data accessed in a method. We then explain how data access summaries are formed from symbolic access paths.

3.1. Symbolic access paths

A symbolic access path [7, 13] is a pair $o \cdot f$ consisting of an object o, and a sequence of field names $f = f_1, \ldots, f_n$. Each successive field name f_i is the name of a reference-type instance variable defined within the object pointed to by f_{i-1}. The object o is the source object of the path, and the object pointed to by f_{n-1} is the destination object of the path; f_n in the destination object is a field variable, which may be of any type.

The example shown in Figure 2 illustrates this basic notation of symbolic access paths. The class `ListNode` is a linked list node class. It contains a static `head` field that is a reference to the head node of a linked list, an integer `data` field, and a `next` field that is a reference to the following node in the linked list. The method `zeroHeadNode` assigns the value 0 to the `data` field of the head node. The symbolic access path for the method is simply `ListNode.head.data`, representing the access that occurs in line 6 of the code. The source object of the access path is `ListNode.head`, and the destination object is `data`.

```
1: class ListNode {
2:    static ListNode head;
3:    int data;
4:    ListNode next;

5:    void zeroHeadNode() {
6:       ListNode.head.data = 0;
7:    }
8: }
```

Figure 2. An example illustrating symbolic access paths.

In the *zJava* system, symbolic access paths are used to represent accesses to only two types of objects in a method *m*: global objects, and objects passed to *m* as actual parameters (including the receiver object itself; i.e., `this`). Objects created locally inside a method, and do `escape` the method (i.e., objects whose lifetime extends beyond that of the method [6]), can only escape through assignment to global variables and/or parameters, or through return values [6]. Hence, accesses to these escaping objects will be captured by the symbolic access paths of these variables. In contrast, accesses to local objects constructed within *m*, but do not escape *m*, need not be represented by symbolic access paths since they cannot be shared among threads. In addition, symbolic access paths for objects returned by methods called within *m* are also not represented because the run-time system uses a mechanism called a future to synchronize accesses to them, without relying on data access summaries, as will be discussed in Section 4.3.5. Consequently, the *zJava* compiler generates symbolic access paths only for accesses to global objects and for accesses to objects passed as actual parameters to a method.

Symbolic access paths for a method are generated in terms of the formal parameters of the method. However, the actual parameters of the method vary from one call site to another. Also, the actual parameters may not be known until run time. Consequently, the actual source objects of a method's symbolic access paths may not be fully determined until run time. Therefore, we introduce a special notation to represent the source object in a symbolic access path, which we refer to as the anchor. The anchor "`$n`" denotes the `n`-th parameter of the method. Hence, "`$1`" denotes the first parameter and "`$2`" denotes the second parameter. The special anchor "`$0`" denotes the receiver object, passed to the method as the implicit parameter `this`. The actual source object of a symbolic access path can then be determined at run time using its anchor and the context of the method call. It should be noted that when the source of a symbolic access path is a static field, no anchor is required; a static field is unique within a program.

The example shown in Figure 3 illustrates anchors in access paths using the earlier example of the `ListNode` class, but with the method `UpdateNode`. This methods updates the fields of the receiver node object in the list from `otherNode`, as shown in the figure. The symbolic access paths for this method are `$0.data, $0.next`, `$1.data`, and `$1.next`, corresponding to accesses to `this.data, this.next`, `otherNode.data`, and `otherNode.next`, respectively.

```
class ListNode {
    static ListNode head;
    int data;
    ListNode next;

    void UpdateNode(ListNode otherNode) {
        this.data = otherNode.data;
        this.next = otherNode.next;
    }
}
```

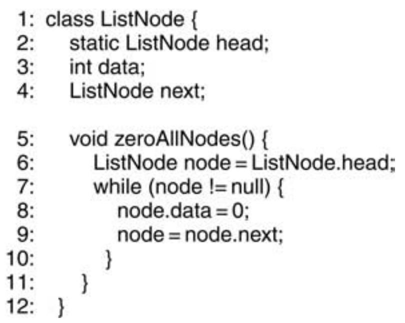
Figure 3. An example illustrating anchors in symbolic access paths.

3.1.1. Recursive accesses. In some cases, it may not be feasible to enumerate all possible symbolic access paths of a method. For example, the method may use a variable *v* to traverse a recursive data structure in a loop, and the number of times the loop iterates is determined by a condition that is evaluated at run time. The objects *v* refers to during the execution of the method are the successive elements of the recursive data structure. Hence accesses to *v* give rise to many (potentially infinite) symbolic access paths. We refer to such accesses as recursive accesses.

To address the above problem, we use a single symbolic access path to represent the successive accesses to elements of the data structure. The component of the access path that is used for the traversal is "factored out" and is annotated to indicate the traversal. This factored out component of the access path is referred to as the basis of the path. In effect, this form of the symbolic access path is a *k*-limited representation of the traversal of the data structure [7].

The example shown in Figure 4 illustrates symbolic access paths for recursive accesses. The method `zeroAllNodes` traverses the linked list and assigns 0 to the data field of every node. The access at line 8 is a recursive data access and it requires the following symbolic access paths: "`ListNode.head.data`", "`ListNode.-head.next.data`", "`ListNode.head.next.next.data`", "`ListNode.-head.next.next.next.data`", etc. The number of access paths cannot be determined at compile time because it is not possible to statically determine when the loop terminates. Instead, the access is represented by the following single symbolic access path: "`ListNode.head(.next)*.data`". The basis "`.next`" is followed by an asterisk to form a 0-limited representation of the traversal; that is to say, it may occur zero or more times using the `next` field.

3.1.2. Array accesses. Array elements are treated as individual fields in the array object. It is also possible to aggregate multiple array elements and denote the collection with a single access path in which the name of the array is suffixed with two integers, enclosed in brackets, that indicate the section of the array being accessed. The first integer marks the element that starts the array section, and the second, larger, integer marks the element that ends the section. All the elements of an


```
 1: class ListNode {
 2:    static ListNode head;
 3:    int data;
 4:    ListNode next;

 5:    void zeroAllNodes() {
 6:       ListNode node = ListNode.head;
 7:       while (node != null) {
 8:          node.data = 0;
 9:          node = node.next;
10:       }
11:    }
12: }
```

Figure 4. An example illustrating symbolic access paths for recursive accesses.

array are represented using an asterisk instead of an index range. For example, the access path "`$1.children[0-9]`" can be used to specify the first ten elements in the `children` array in lieu of the ten symbolic access paths: "`$1.children[0]`", "`$1.children[1]`", `...`, "`$1.children[9]`". Similarly, the access path "`$1.children[*]` represents all the elements of the array, and the path "`$1.children[$2-$3]` represents a section of the array whose start and end indices are the second and third parameters to the method in which the access occurs.

3.2. Constructing data access summaries

A data access summary specifies the read set and the write set of a method *m*, i.e., the sets of variables which *m* reads and writes, respectively. The summary consists of a list of entries. Each entry in the list is a (symbolic access path, access type) pair that specifies a shared variable *v*. The access type determines whether *v* belongs to the read set or the write set of *m*. If *v* belongs to both the read and write sets of *m*, then its access type is indicated as "write".

The data access summary of a method *m* includes the data access summaries of all methods called by *m*. In other words, if a method *m′* called by *m* writes to a variable *v*, then *m* is assumed to also write to *v*, even if it does not directly write to the variable in its own body. We refer to this as the inclusion property of data access summaries, and it is essential for proper synchronization, as will be discussed in Section 4.3.2.

The Java code shown in Figure 5 is used to illustrate the construction of data access summaries. Instances of the `Point` class represent points in the Cartesian plane. The `midpoint` method computes the mid-point between the point represented by the receiver object and the point represented by the parameter `p`, and returns the result as a new `Point` object, complete with a unique serial identifier, which is incremented for every `Point` object allocated.

```
class Point {
   static int nextId = 0;
   int id;
   float x, y;

   Point midpoint(Point p) {
      Point q = new Point();
      Point r = new Point();
      r.x = (this.x + p.x)/2;
      r.y = (this.y + p.y)/2;
      r.id = updateId();
      return r;
   }

   int updateId() {
      nextId = nextId + 1;
      return nextId;
   }
}
```

Figure 5. A simple method midpoint.

```
($0.x, read)
($0.y, read)
($1.x, read)
($1.y, read)
(Point.nextId, write)
```

Figure 6. Data access summary for the midpoint method.

The data access summary for the `midpoint` method is shown in Figure 6. The first two entries indicate that the fields `x` and `y` of the receiver object are read by the method. The next two entries indicate that the same fields of the parameter `p` are also read. The last entry is the access to the static variable `nextId`. It is included in the data access summary of `midpoint`, even though `midpoint` does not access `nextId`, because of the inclusion property described above; the method `updateId` accesses `nextId` and is called by `midpoint`. The fully qualified name of the static variable is used as the anchor in the symbolic access path. It is marked as "write" since `updateId` reads and then increments the value of the specified field. Accesses to the variable `q` are not included in the data access summary, because it is local to the method, and the object it points to never escapes the method. Accesses to the variable `r` are also not included in the data access summary, because it is local to the method, and the object it points to only escapes as a return value.

4. The run-time system

This section describes the components of the run-time system and how they operate. The notion of regions, which are used to represent and control accesses to shared variables within the run-time system, is first defined. The main component of the run-time system, called the registry, is then introduced. Finally, the various operations performed by the run-time system, including creating, synchronizing, and terminating threads, are described.

4.1. Regions

The run-time system translates symbolic access paths into regions, which represent variables in the system and describe how they are shared. Each region corresponds to an area of memory in the heap that is potentially accessed by multiple, possibly concurrent, threads. A region is usually a variable (of an object or primitive type), but it can also be a section of an array, containing many individual elements. A region is identified by the address in memory of the variable it represents.

4.2. The registry

During the execution of the program, the *zJava* run-time system maintains a set of data structures, that we collectively refer to as the registry. The purpose of the registry is twofold: it is used to determine data dependences among methods, and it is used to synchronize the threads that have been created to execute those invoked methods.

The registry consists of a set of region nodes and a set of thread nodes. A region node r_v corresponds to the region of a variable v, and represents the variable within the registry. A thread node t_τ describes a thread τ that has been created by the run-time system, and contains the ID of its parent thread, a pointer to the method which it executes, and pointers to region nodes representing data accessed by the thread.

The registry is structured as a table of region nodes that represent all shared variables in use by the program at a given point in time. Each region node points to a list of thread nodes representing threads that access the associated region during their lifetimes. A region node is also associated with a reader/writer lock, which ensures proper synchronization of concurrent threads. The lock is designed so that multiple reader threads (threads that access, but do not write to, a given region) can share the reader lock and execute concurrently, while writer threads (threads that do write to a given region) can only execute one at a time.

Thread nodes are created and enqueued in the thread list of a region node when a new thread is created to execute a method body. The enqueuing of threads is performed so that a thread list remains always sorted by the serial execution order of the methods associated with the thread. That is, the order in which thread nodes appear on a thread list is the same as the order in which their corresponding methods are executed in the sequential program. The mechanism for ensuring that threads are enqueued in this order will be described in Section 4.3.3.

Figure 7 shows a graphical view of a simple registry. It contains three region nodes r_x, r_y, and r_z, corresponding to the variables x, y, and z, respectively. The registry also contains thread nodes t_τ and t_σ corresponding to threads τ and σ. The two threads execute methods m_τ and m_σ, respectively. Region r_x is accessed by both threads. In sequential execution, m_σ executes and accesses x before m_τ. Consequently, thread node t_σ appears ahead of (i.e., to the left of) thread node t_τ on the thread list of region r_x. In addition, m_τ accesses y, and m_σ also accesses z.

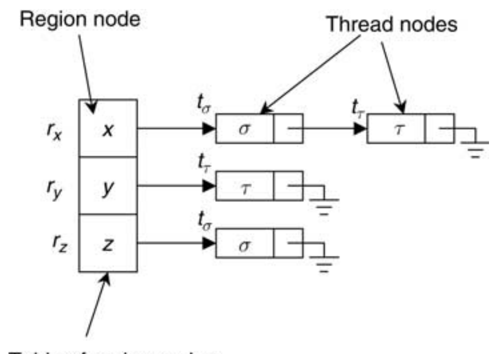


Figure 7. The *zJava* registry maintains sorted thread lists for shared regions.

Hence, thread nodes t_τ and t_σ appear on the thread lists of regions r_y and r_z, respectively.

The registry is updated dynamically; new thread nodes and region nodes are created for newly created threads and newly allocated shared variables, respectively, and are inserted into the registry. The registry is unaware of the existence of a variable v until the creation of a thread that registers v as possibly shared, at which point a region node r_v is allocated in the registry. Thread nodes are deleted from the registry as threads terminate, and region nodes are deleted when their associated objects are garbage-collected.

The following notation will be used for the remainder of this paper. Variables are denoted by x, y, and z. Their corresponding regions are denoted by r_x, r_y, and r_z, respectively. Executing threads are referred to as τ, σ, and γ. Their corresponding thread nodes are denoted by t_τ, t_σ, and t_γ, respectively, and they execute methods m_τ, m_σ, and m_γ, respectively.

4.3. Registry operations

4.3.1. Region creation. When a class is loaded by the JVM, the *zJava* run-time system receives the data access summaries of all the methods of that class. These data access summaries are stored in a look-up table. When a thread τ calls a method m_σ, a child thread σ is created. Compiler-inserted code in the parent thread τ retrieves the data access summary for m_σ from the summary look-up table, and creates regions from this summary.

The creation of regions for a method m_σ proceeds in three main steps. In the first step, the run-time system identifies the real sources of the access paths for the methods, and hence determines what objects are accessed by m_σ. We refer to this step as source resolution. This step is accomplished by replacing the anchors in the access path with actual source objects available to the method at run time, i.e., objects passed as actual parameters to the method.

In the second step, each resolved access path is expanded into all its sub-paths. Hence, an access path $o \cdot f_1, \ldots, f_n$ is expanded into n access paths, namely, $o \cdot f_1$, $o \cdot f_1 \cdot f_2$, $o \cdot f_1 \cdot f_2 \cdot f_3$ and so on, up to $o \cdot f_1, \ldots, f_n$. This expansion is necessary because the access to $o \cdot f_1, \ldots, f_n$ requires accesses (reads) to the intermediary fields $f_1, \ldots, f_{n-1}$. Consequently, these intermediary fields must be represented in the registry by individual region nodes to ensure proper synchronization with other threads that may access them. We refer to this step as path expansion. Symbolic access paths for recursive data accesses are similarly expanded. For example, the symbolic access path `$0.head{.next}*` translates to `this.head`, `this.head.next`, `this.head.next.next`, etc. The expansion stops when the current `next` field contains null.

In the third and final step of region creation, the run-time system searches the registry for the region node r_v for every variable v that was expanded from the resolved access path. If a region node does not already exist for a given variable, a new node is allocated for it and is added to the registry.

```
class Matrix {
  Reader     input;    // input file
  double[][] data;     // data in matrix
  int        nRows,    // size of matrix
             nCols;

  void readRow(int r, Reader in) {
   for (int c = 0; c < nCols; c++) {
      data[r][c] = parseDouble(in);
   }
 }

 void readMatrix() {
   for (int i = 0; i < nRows; i++) {
      readRow(i, input);
   }
 }

 double[] multRow(int r, Matrix m) {
   double[] d = new double[m.nCols];
   for (int i = 0, d[i] = 0; i < m.nCols; i++) {
      for (int j = 0; j < m.nRows; j++) {
         d[i] += data[r][j] * m.data[j][i];
      }
   }
   return d;
 }

 Matrix multiply(Matrix m) {
  Matrix product = new Matrix();
   for (int i = 0; i < nRows; i++) {
      product.setRow(i, a.multRow(i, m));
   }
 }

 public static void main(String[] args) {
   Matrix a, b, c, d, e;

   // ...some code to initialize b, d and e

   a.readMatrix();    // first fork point
   c = a.multiply(b); // second fork point
   e = a.multiply(d); // third fork point
 }
}
```

Figure 8. A matrix multiplication example.

As an example, consider the program in Figure 8, in which the `main` method invokes the `readMatrix` method on the object `a`. The data access summary for `readMatrix` is shown in Figure 9. The summary includes `$0.nCols` and `$0.data` even though they are not accessed in `readMatrix` because they are accessed by `readRow`, which is called in `readMatrix`.

For each symbolic access path in this summary, the run-time system replaces its anchor with the actual reference-type argument to the method. Hence, `$0.nRows` is resolved to `a.nRows`, `$0.nCols` is resolved to `a.nCols`, etc. Since each of the symbolic access paths of the `readMatrix` method has only one field, path

```
($0.nCols, read)
($0.nRows, read)
($0.input, read)
($0.data, write)
```

Figure 9. Data access summary for readMatrix.

expansion for this example is straightforward. Each path translates to one region, and the corresponding region node is simply added to the registry. The resulting region nodes (with empty thread lists) are shown in Figure 10.

4.3.2. Thread creation. A thread τ creates a child thread σ when τ invokes a method m_σ in its body. However, before the child thread can begin to execute, regions accessed by the child thread must be determined, and thread nodes for the child thread must be allocated and inserted on the thread lists of the corresponding region nodes. These steps are performed by the run-time system; the code of m_τ is restructured to make the appropriate calls to the *zJava* run-time library.

When the parent thread τ creates the child thread σ, the data access summary of m_σ is resolved and expanded (as described in the previous section) into a set of variables V used by m_σ. Corresponding region nodes are allocated for variables in V if necessary. A thread node for σ is then inserted into the thread list of each of those region nodes. Once the insertion of the region nodes is complete, the parent thread τ continues the execution of its own body. The child thread σ can proceed to execute when it acquires the locks for its regions, as will be described in Section 4.3.4.

The parent thread τ must wait for all threads that precede it on a thread list of a region $r_v, v \in V$, and that write v, to complete before it can insert any of the thread nodes of the child thread σ. This is necessary to ensure that σ's thread nodes are indeed enqueued onto the thread lists of the correct regions. Consider, for example, an access $o \cdot f_1 \cdot f_2$ by σ, which translates into two regions, one representing the variable $o \cdot f_1$ and the other representing the variable $o \cdot f_1 \cdot f_2$. If a thread γ that precedes τ writes to $o \cdot f_1$, and τ does not wait for γ to complete, an incorrect value (with respect to sequential execution) of $o \cdot f_1$ will be used to evaluate $o \cdot f_1 \cdot f_2$ and hence, to identify the corresponding region node. Consequently, the thread node of σ will then be inserted on the thread list of the wrong region node. By waiting for γ to complete its write to $o \cdot f_1$, the correct value of $o \cdot f_1$ is used, and the thread node of σ is inserted on the thread list of the correct region node.

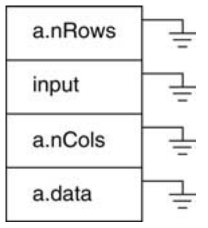


Figure 10. Region nodes are added for all regions that the `readMatrix` method will access.

The inclusion property described earlier in Section 3.2 ensures that a thread node of τ exists on the thread list of r_v even if τ does not access v in its body (since m_σ is called in the body of τ). Hence, it will always be possible to determine the threads that precede τ on the thread lists of every region accessed by σ.

4.3.3. Thread ordering. Thread nodes in any given thread list must be kept sorted by serial execution order of the associated methods to preserve program correctness. Furthermore, the order of thread nodes must be consistent among all thread lists in the registry. That is, if the thread nodes of a thread σ precede the thread nodes of another thread τ on one thread list, then thread nodes of σ must precede thread nodes of τ on all thread lists. This serial execution order of thread nodes can be achieved by inserting the thread nodes of a newly created thread immediately preceding those of its parent thread on any thread list. We present the following informal argument for justification; a formal proof can be found in Chan [4].

Consider two threads σ and τ, corresponding to methods m_σ and m_τ, respectively. We say that $\sigma > \tau$ if m_σ should access a variable v before m_τ when the two methods are invoked in the sequential program. Since we require the inclusion property of data access summaries, then, for a given thread σ in the thread list of a region r_v, all of σ's ancestor threads must also have thread nodes in the same thread list. Suppose σ is the most recent child thread of its parent τ, and that both access r_v. Since, in a sequential program, a called method must terminate before the calling method continues, once σ has been created, σ must finish accessing r_v before τ accesses r_v again, even though τ may have been using the variable before σ was created. Thus, $\sigma > \tau$, and the thread nodes of σ must precede the thread nodes of τ on the thread list of r_v. Now suppose τ creates a new thread γ after σ. Hence, $\sigma > \gamma$ and the thread nodes of σ must precede those of γ. At the same time, $\gamma > \tau$, for the same reason that $\sigma > \tau$, as described above. Hence, the thread nodes of γ must precede those of τ. Consequently, the thread nodes of γ must be inserted between those of σ and those of its parent τ. In general, the thread nodes of a thread must be inserted between those of its parent and those of its older sibling. This can only be achieved by inserting the thread nodes of a thread immediately preceding those of its parent.

The ordering of threads according to serial execution order eliminates the possibility of deadlock in the run-time system since it serial execution order imposes a total order on the threads in the system [4].

Consider the matrix multiplication program shown earlier in Figure 8. The `readMatrix` and `multiply` methods operate on a row-by-row basis; the former calls the `readRow` method `nRows` times to fill the matrix with data, and the latter calls the `multRow` method `nRows` times to multiply the matrix with the other matrix `m`. Calls to `readMatrix` and `multiply` will result in multiple threads. The call sites of these calls are the fork points for these threads.

At the first fork point (see comments in code), a new thread is created for the call to `readMatrix`, regions nodes are allocated for all regions that the invoked method will access, and the registry is modified as shown in Figure 11. It should be noted that as a new thread node is inserted into a thread list, thread nodes for all its ancestors are also inserted, if they are not already on the list. This is consistent with the requirement that the data access summary of a method include those of its callees.

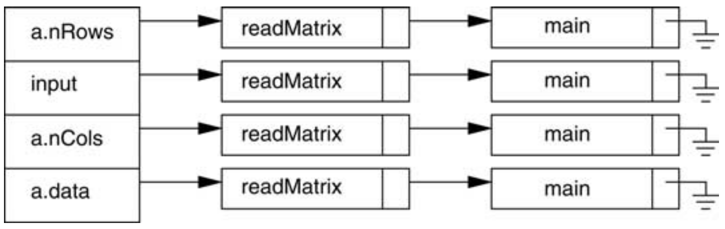


Figure 11. New thread nodes are created for the `readMatrix` thread; the nodes are inserted into the thread lists corresponding to regions accessed by the thread.

Thus, in Figure 11, a thread node for the `main` thread is automatically inserted after every node for the `readMatrix` thread.

At the second fork point, the method `multiply` is called, which calculates the product of the matrices `a` and `b`. This causes a second thread (labeled $\texttt{multiply}_1$ in Figure 12) to be created. Region nodes are allocated for `b.nRows, b.nCols`, and `b.data`, which are in the read set of the new thread, but for which there are no existing region nodes. The new thread also writes to the product matrix `c`, so a region node is created for `c` as well. A thread node for $\texttt{multiply}_1$ is then inserted into each of the thread lists of these regions. A thread node is also inserted into the thread lists of existing region nodes for `a.nRows` and `a.data` because $\texttt{multiply}_1$ reads the corresponding variables. The new thread node is placed immediately preceding that of its parent `main`, and thus behind that of its sibling thread running `readMatrix`. This is consistent with the serial execution order in which the method `readMatrix` must complete before the call to `multiply` occurs.

The registry is similarly updated for the third forked thread, which runs `multiply` a second time, but on the matrices `a` and `d`, producing the matrix `e`.

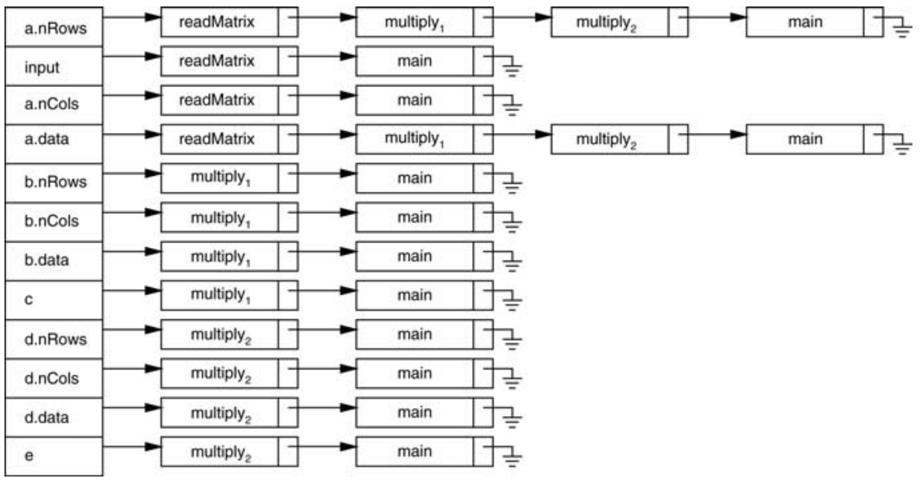


Figure 12. The registry is updated as the two `multiply` threads are created; the thread lists of shared regions keep the threads in serial execution order.

Region nodes are allocated for the fields of `d` and the matrix `e`, which are accessed by the new thread (labeled $\texttt{multiply}_2$). The new thread node is inserted into the thread list of the regions that it uses, including those of `a.nRows` and `a.data`, where the run-time system again preserves the serial execution order by placing the thread nodes of the new thread immediately preceding those of its parent `main`. The resulting registry is shown in Figure 12.

4.3.4. Thread synchronization.

A reader method (or thread) is defined as a method (or thread) that accesses a shared variable (or region) but never writes to it. In contrast, a writer (or thread) is a method (or thread) that accesses a shared variable (or region) and possibly writes to it. It should be stressed that even if a writer thread does not actually write to a region at run time, perhaps due to branching from a conditional statement, or the throwing of an exception, it may not be possible to predict such behavior at compile time. Consequently, the compiler must be conservative and classify a method as a writer of the variable, and at run time, the thread executing this method must be synchronized as a writer.

As described earlier, every region node in the registry is associated with a reader/writer lock. Each reader thread must acquire a reader lock on the region node r_x prior to accessing the region x. The reader lock allows concurrent reads of the same region by multiple threads, as explained below. In contrast, a writer thread must acquire a writer lock, which is an exclusive lock, i.e., acquiring the lock blocks the execution of all other threads that access the same region. If a thread is unable to acquire a (reader or writer) lock, its execution must block until it can.

A reader thread τ is immediately granted the reader lock of a region r_x if its thread node t_τ is the first in the thread list of r_x, or if all preceding other threads on the thread list are also readers. Otherwise τ is blocked and waits for a preceding thread to signal it. When τ acquires the reader lock on r_x, it checks if the successor of t_τ belongs to another waiting reader thread. If it does, τ signals the waiting reader thread and grants it a reader lock as well. A writer thread σ is granted a writer lock only if its thread node t_σ is the first node in the thread list of r_x. This means that all other accesses to the variable x must complete before m_σ may write. This preserves data dependences and ensures correctness of execution. When the writer thread terminates (or otherwise relinquishes its writer lock), it signals the next thread (if any) on the thread list of r_x, and grants it the lock it has been waiting for. This scheme allows multiple threads that read the same variable to execute concurrently, yet prevents any conflicting accesses to occur at the same time.

This synchronization scheme is illustrated using the example in Figure 8. Suppose the program is restructured to create a thread for every call to the `readRow` and `multRow` methods. The thread executing the `i`-th call to the `readRow` method, which modifies row `i` of the matrix `a`, must terminate before either of the two threads running the `multiply` method makes the `i`-th call to the `multRow` method to perform multiplication on the same row. Otherwise, the thread(s) executing the `multRow` method will use values from an undefined row in its calculation, and yield incorrect results. However, once the entire matrix `a` has been read in, both `multiply` threads, and their child threads running `multRow`, can proceed with their multiplications concurrently. None of the threads modifies the shared `data`

```
void readRow(int r, Reader in) {
   Region.getRegion(data[r]).acquire();

   for (int c = 0; c < nCols; c++) {
      data[r][c] = parseDouble(in);
   }

   Region.getRegion(data[r]).release();
}

double[] multRow(int r, Matrix m) {
   double[] d = new double[m.nCols];

   Region.getRegion(data[r]).acquire();

   for (int i = 0, d[i] = 0; i < m.nCols; i++) {
      for (int j = 0; j < m.nRows; j++) {
         d[i] += data[r][j] * m.data[j][i];
      }
   }

   Region.getRegion(data[r]).release();

   return d;
}
```

Figure 13. The *zJava* compiler inserts code at synchronization points to ensure that data dependences are not violated at run time.

array contained in the matrix `a`; in other words, they are all readers of the shared array. Hence there is no conflicting data accesses among them, and they may execute concurrently.

To facilitate inter-thread synchronization, the *zJava* compiler inserts region-based synchronization primitives into the bodies of methods. Figure 13 shows the definition of the `readRow` and the `multRow` methods after such a transformation. The synchronization routines are provided by the run-time system in normal Java classes.

4.3.5. Thread termination. A thread terminates when the method the thread is executing returns. The thread stores any return value of the method in a future [3], waits for its child threads to terminate, and finally removes its own thread nodes from thread lists of all region nodes the registry. At that point, the registry has no more reference to the thread, and the thread can terminate itself.

The future is a synchronization device for handling return values from child threads. If the parent thread attempts to get the value of the future before it has been set by the child thread, the parent thread is blocked until the child thread returns. This maximizes the concurrency between the parent thread and the child thread, in those cases where the return value from the child thread is not used immediately (or used at all) by the parent thread.

5. Experimental evaluation

We implemented the *zJava* run-time system in Java. It comprises 49 Java classes of a total of 10,532 lines of code, including JavaDoc comments. The run-time system is implemented as a user-level library and, as a result, requires no changes to the JVM. This makes the run-time system portable—it can be used on any multiprocessor on which a JVM with native threading[2] exists. We used Sun Microsystems' Java 2 Software Development Kit, Standard Edition, version 1.2.2, both to develop the run-time system and to experiment with it. The Java compiler provided in this version of the Java 2 SDK is conservative, and performs very little optimization on the code.

We evaluated the performance of the system on a dedicated Sun Microsystems Enterprise 450 server equipped with four 296 MHz UltraSPARC II CPUs and 1.5 GB of main memory. In this section, we report the performance of the run-time system using five benchmarks. Since the analyses required to compute data access summaries are not yet fully implemented in our compiler, the data access summaries were manually computed (mimicking what the compiler does) for these benchmarks. Each program was also manually restructured to transmit the data access summaries to the run-time system and to insert thread creation and synchronizations calls. The benchmarks are summarized in Table 1. It should be noted that while some of these benchmarks use arrays as the main data structure, their code uses references and dynamically creates its objects. Traditional array-based parallelizing compiler technology would fail to to extract parallelism in these benchmarks, even though they may manipulate arrays in loops.

The performance of the benchmarks is reported using speedup, which is defined as the ratio of the execution time of the sequential program (running without the run-time system) to the execution time of the parallel program, for a given number of processors. The speedup is ideal when it equals the number of processors used.

The speedup of the benchmarks is shown in Figure 14.[3] The figure indicates that while the benchmarks do not achieve ideal speedup, scalable performance is achieved: the speedup of each benchmark increases with increasing numbers of processors. The main sources of speedup degradation are run time overhead and lack of parallelism in some benchmarks.

There are three main sources of run time overhead. First, a parent thread incurs overhead to create a child thread. The parent thread must resolve and expand the data access summary of the child thread to determine its regions. Further, the parent

Table 1. Summary of the benchmarks and their sequential execution time

Benchmark name	Sequential execution time (seconds)
`ReadTest` (100, 800)	80
`Matrix` (1000×1000)	188.7
`TSP`	566.0
`Mergesort` (2M, 32K)	6.4
`15-puzzle`	53.0

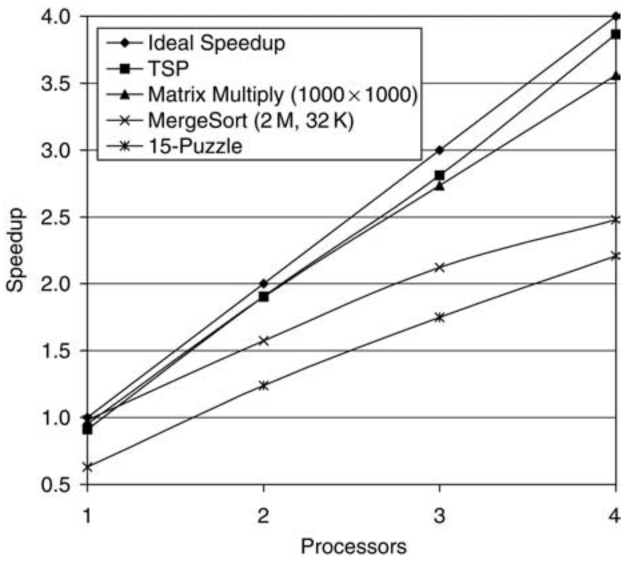


Figure 14. The speedup of the benchmarks.

thread must insert thread nodes of the child thread onto the thread lists of the corresponding region nodes. In the process, the parent thread may block, waiting for preceding writers on a thread list to finish. Also, the parent must create the child thread itself. Second, the registry is a shared data structure in itself, and it must be accessed atomically. Consequently, overhead is incurred at run time to coordinate registry access by the running threads. Third, a child thread uses the registry in order to determine when it can execute, which introduces some run time overhead. There are other sources of overhead, such as that incurred at class load time to transmit data access summaries to the run-time system, but they occur only during initialization, and can be assumed negligible. It is difficult to non-intrusively profile the run-time system to obtain a breakdown of the contribution of each of these sources of overhead. Hence, we describe them collectively as run time overhead for creating, executing, and synchronizing threads. The overhead varies from one benchmark to another depending on the number of threads created and the nature of the benchmark. We consider the performance of each benchmarks in some details in the following sections.

5.1. `ReadTest`: *A synthetic micro-benchmark*

The `ReadTest` program is a micro-benchmark that aims to measure the performance improvement that can be achieved by executing multiple reads (of the same data item) in concurrent threads. The benchmark consists of a loop that creates *m* threads, each of which reads (but does not write) only one object and performs *n* computations with it.[4] Hence the benchmark is a good indicator of the

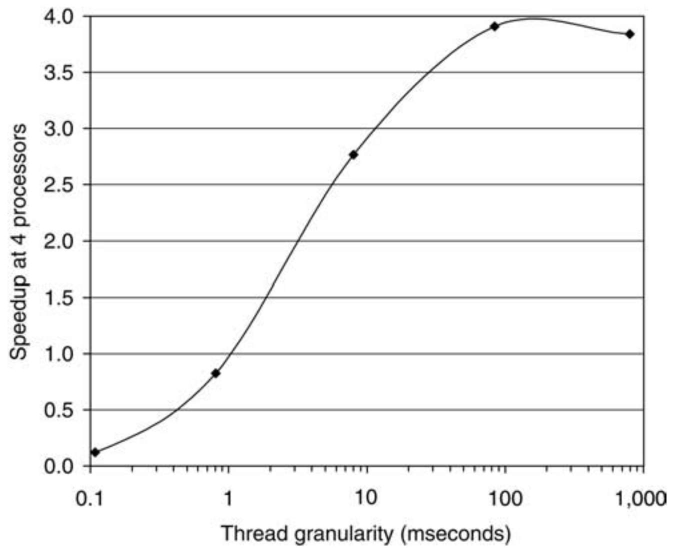


Figure 15. The effect of thread granularity on the performance of the run-time system.

overhead of parallelizing a program without thread blocking. Its performance provides an upper bound on the speedups that can be obtained by the system, and is also indicative of the overhead incurred in accessing and using the registry of the run-time system.

The speedup of `ReadTest` at four processors is shown in Figure 15 as a function of thread granularity (i.e., different values of *n*), when the number of threads is equal to 100. The granularity of each thread is expressed in milliseconds. The figure indicates that the performance of the benchmark is poor when the granularity of a thread is very small (less than 1 millisecond). Indeed, the performance reflects a slow-down in execution time. This is because at such small granularities, the costs of creating, executing, and synchronizing threads outweigh benefits gained from parallelism. However, when the granularity of a thread becomes greater than 50 milliseconds, close to ideal speedup is achieved. Indeed, for thread granularity greater than 100 milliseconds, the speedup is close to ideal (3.6 at four processors).

The effect of the number of threads in the `ReadTest` benchmark (i.e., the value of *m*) on its speedup is shown in Figure 16. The speedup of the `ReadTest` benchmark when the granularity of each thread is 800 milliseconds is shown for different number of threads in the system. The figure indicates that when the number of threads is very small (less than 10), the speedup of the benchmark is low. This is because at such small number of threads, parallelism is limited. When the number of threads is increased (to about 100), close to ideal speedup can be achieved. However, when the number of threads is large (about 1,000), speedup degrades slightly. This is due to the contention resulting from sharing the registry and from the blocking of parent threads as they create child threads.

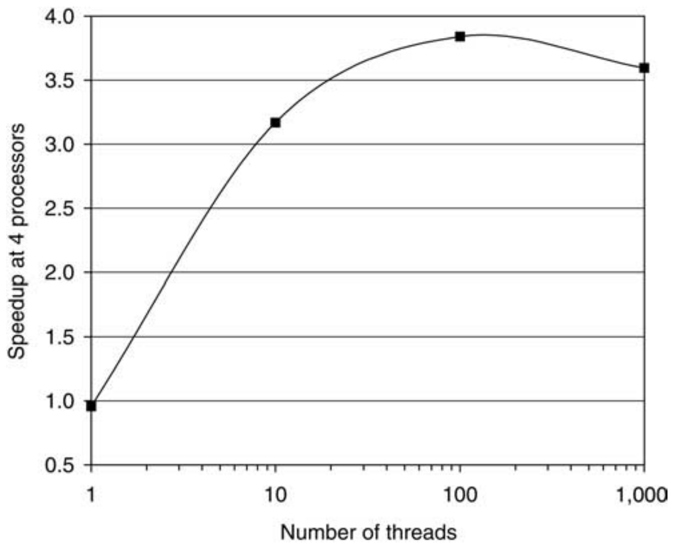


Figure 16. The effect of the number of threads on the performance of the run-time system.

5.2. *The* `Matrix` *benchmark*

The `Matrix` benchmark program is the matrix multiplication application described earlier in the paper. It's is a slightly modified version of the one shown in Figure 8. The speedup of `Matrix` for a 1000×1000 matrix for different number of processors was shown earlier in Figure 14. Although the speedup of `Matrix` is less than ideal (3.6 at four processors), the speedup increases linearly with the number of processors.

The speedup of `Matrix` as a function of the matrix size is shown in Figure 17. The program creates one thread to compute each row of the product matrix. The speedup is shown for various matrix sizes, and hence, varying number of threads and granularity of each thread. For example, for a 1000×1000 matrix, 1,000 threads are created, each of which performs a million additions and multiplications. The speedup of for the largest matrix size is 3.6 at four processors.

5.3. *The* `Mergesort` *benchmark*

The `Mergesort` benchmark implements a two-way mergesort algorithm. It divides the input into two ‘‘units’’, recursively sorts each unit, and finally merges the two units into a single-sorted unit. When the size of a unit drops below a user-defined threshold, the sorting method provided by the Java class library[5] is invoked to sort the unit. In the *zJava* parallelized program, a thread is used to sort a unit whose size drops below the threshold. Hence, the unit size essentially determines the number of threads that will be created to perform the sort, as well as the granularity of each thread.

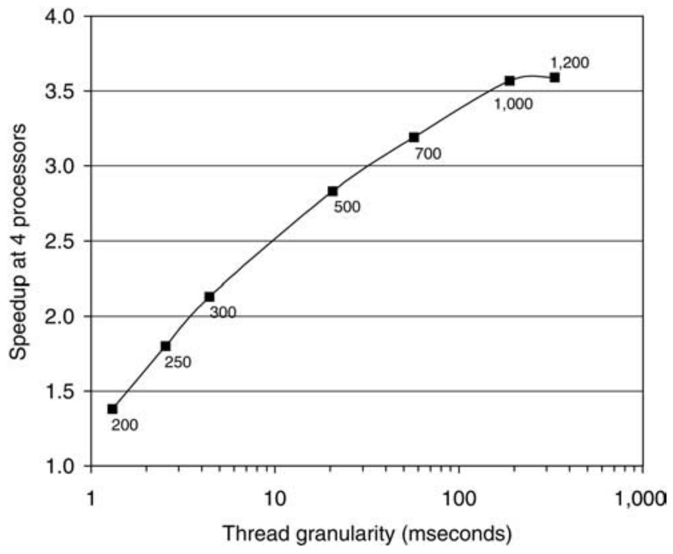


Figure 17. The speedup of `Matrix` at four processors as a function of matrix size.

The benchmark is used to sort 2M (2^{21}) randomly generated integers. The speedup of the benchmark with a unit size of 32,768 (2^{15}) integers was shown earlier in Figure 14. The speedup is low, but it is close to what is attainable from the divide-and-conquer parallelism that mergesort offers [18]. That is, the main source of speedup degradation in this benchmark is lack of parallelism.

The effect of the unit size, and hence the number of threads and their granularity, on the speedup of the benchmark is shown in Figure 18. For small unit sizes, there are more units, and hence, more threads. This leads to more finer-grained parallelism, but also to increased thread creation and synchronization overhead. Conversely, for large unit sizes, there are less units, and hence, less threads. This leads to limited coarser-grained parallelism, but less synchronization overhead. Since we experimented with only four processors, there is little benefit to increasing the number of threads at the expense of higher overhead. Thus, the performance of the benchmark improves as the unit size in increased, as shown in Figure 18.

5.4. *The* `TSP` *benchmark*

The `TSP` benchmark program uses a branch-and-bound algorithm to compute an optimal tour for a travelling salesman problem. Our implementation of the `TSP` benchmark is similar to the one in Huynh [15]; essentially the three top-most levels of the search tree are flattened into a 3D array, and individual threads are created to expand a sub-tree from each element of the array. The speedup of the benchmark was shown earlier in Figure 14. The cost of the best solution discovered at any time

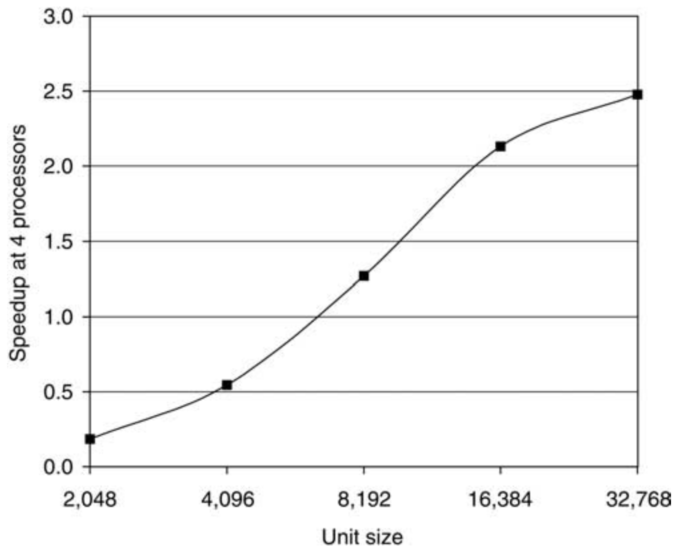


Figure 18. The speedup of `Mergesort` on 2M integers as a function of number of the unit size.

during the search is used to prune the tree. The figure indicates that scaling speedup is achievable for this benchmark. Indeed, the speedup is close to ideal.

In this benchmark, it is important to note that the total number of tree nodes expanded by the parallel benchmark is slightly higher than the number of tree nodes expanded by the sequential benchmark. Thus, we believe that no "search anomalies" exist, and indeed that the speedup of the benchmark stems from parallel execution as opposed to a reduction in computations.

5.5. *The* `15-puzzle` *benchmark*

The `15-puzzle` benchmark solves the 15-puzzle problem[6] using an iterative deepening A* (IDA*) depth-first search algorithm. The benchmark was written as a parallel program by Hui et al. [14]. We reverted it back into a sequential program, which we then automatically parallelized using the *zJava* system. We report on the performance of the automatically parallelized program, and also compare it to the performance of the original parallel program of Hui et al. [14].

The benchmark solves a 15-puzzle problem instance by dynamically building and examining a search tree. The IDA* algorithm builds a search tree of a given depth, and searches for a solution to the problem using depth-first search. If a solution is not found, the depth of the tree is increased, and the search is restarted. The original parallel program expands the game tree up to a threshold level and then creates a separate thread to further explore each of the sub-trees rooted at this level.

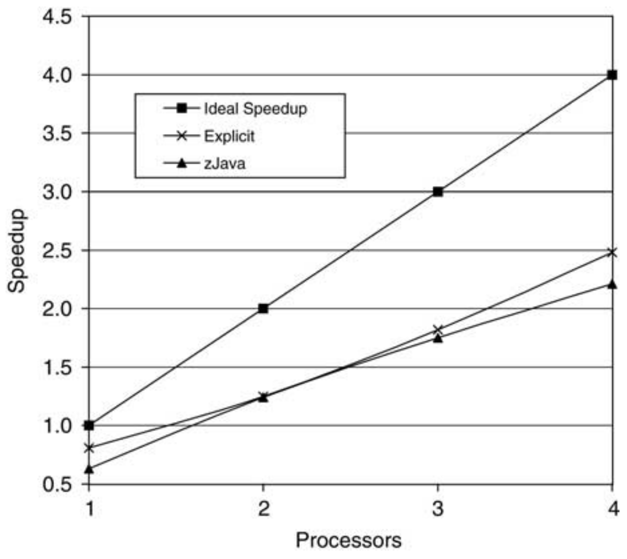


Figure 19. The speedup of the 15-puzzle application.

The speedup of the *zJava* parallelized `15-puzzle` benchmark is shown in Figure 19. The figure indicates that although the resulting speedup is less than the ideal speedup, the speedup of the application increases with increasing the number of processors. However, as the figure also indicates, the speedup of the *zJava* parallelized program is close to that of the original explicitly parallelized program. That is, the *zJava* system automatically exploited parallelism as well as the manually parallelized program.

Similar to the `TSP` benchmark, the number of tree nodes expanded in parallel execution of `15-puzzle` is higher than the number of tree nodes expanded in the sequential execution. Thus, the improvement in performance with increasing numbers of processors stems from the benefit of parallelism as opposed to a reduction in computation caused by search anomalies.

6. Related work

The notion of access paths has been widely described in the literature [13, 17]. Several pointer and alias analyzes [7, 17] use access paths to identify memory locations and alias pairs at compile time. For example, Deutsch [7] devises a version of symbolic access paths that can concisely capture object access information, and uses it for inter-procedural may-alias analysis. We have based our symbolic access path notation on earlier notations, but with the additional objectives of making our notation simpler to evaluate at run time and more flexible to represent array regions.

Furthermore, our work is a novel application of access paths in that we use them at run time to compute data dependences among concurrent threads.

The *zJava* system exploits parallelism at the method level, rather than the loop level, traditionally exploited by parallelizing compilers. There has been a number of systems that exploit parallelism at a non-loop level, commonly referred to as task level parallelism. Abdelrahman and Huynh [1, 15] implement a system for automatically parallelizing array-based C programs at the method level. Our system builds on theirs, but extends their work to address automatic parallelization of pointer-based programs that employ dynamic data structures.

Rinard and Lam [16] describe the Jade system, which allows parallelization of C programs at various granularities. Programmers of Jade must manually annotate programs to specify parallelism and synchronization. In contrast, we extract and represent shared data access from the program and automatically exploit parallelism. Nonetheless, our run-time system bears some similarity to that of Jade, which maintains object queues in serial execution order.

Gross et al. [10] design and implement a compiler based on High Performance Fortran (HPF) called `Fx`. It extends HPF to allow programmers to specify task parallelism. However, it requires programmers to supply directives that define input/output parameters of each task, and the mapping of tasks onto processors. In contrast, we automatically extract and exploit parallelism.

Rugina and Rinard [20] employ compile-time-only analysis to automatically exploit parallelism in array-based programs that use divide-and-conquer. In contrast, we can exploit parallelism in non-array-based programs, albeit at higher overhead at run time.

Leiserson et al. [8, 19] design and implement is the Cilk System, which supports multithreaded extensions to C. The system allows programrs to focus on exposing parallelism and exploiting locality, leaving scheduling, load balancing, and communication protocols to the runtime system. The multithreaded extensions in Cilk allow programrs to specify functions that are to execute concurrently and to synchronize these functions. In contrast, our system automatically exploits parallelism in Java programs through the use of data access summaries. Nonetheless, the Cilk run-time system employs more elaborate mechanisms for scheduling and load balancing, some of which could be incorporated in our system.

There has been recent interest in the design of multithreaded processor architectures, which concurrently execute multiple threads belonging to one or more programs, possibly providing hardware support for synchronizing the threads [21]. These architectures require compiler support to identify threads within a program, and may lead to complex processor implementations when thread synchronization is enforced in hardware. The *zJava* system relies on software to synchronize threads, allowing the system to run on processors with or without multithreading capabilities. Nonetheless, multithreaded processors complement our system by providing hardware support that can significantly reduce the overhead of thread synchronization in the system.

The *zJava* system relies on a number of compiler analysis to extract symbolic access paths. The most significant of them is escape analysis. Choi et al. [6] implements escape analysis for the Java programming language, which determines if the lifetimes of objects exceeds those of their declaring scopes. We rely on a similar analysis to determine objects shared among threads.

7. Concluding remarks and future work

We presented and evaluated a novel approach for the automatic parallelization of programs that use pointer-based data structures, written in Java. The approach is novel in that it combines compile time analysis with run time support to extract parallelism among methods in a sequential Java program; an asynchronous thread of execution is created for each method invocation in the program. To ensure correct program execution, methods are analyzed at compile time to determine the data they access, parameterized by their context. A description of these data accesses is transmitted to a run-time system during program execution. The run-time system utilizes this description to determine dependences among threads and enforce sequential execution semantics. In this paper, we described the design and implementation of the run-time component of the system. We also presented the results of its experimental evaluation using a number of application benchmarks that we believe are representative of larger Java programs.

Experimental results indicate that although the system incurs run time overhead, close to ideal speedup is obtained for a number of benchmarks. Indeed, the speedup of all the benchmarks is scaling, i.e., it increases with increasing number of processors. Hence, the benefits of parallelism outweigh the penalty of run time overhead. This indicates that our approach is viable for the automatic parallelization of target programs.

A number of future research directions exist. The system currently does not handle native methods and adding such support will allow more applications to benefit from the system. The system also implements basic policies for managing parallelism and for scheduling. These basic policies may be replaced with more elaborate one, allowing the system to further improve performance, or allowing users more control over resources. For example, the execution time of methods may be profiled at run-time to determine if a method possesses sufficient granularity to warrant its execution by an independent thread. A method that is of small granularity gets executed by the calling thread instead of by an independent thread. Similarly, the knowledge of data accessed by a method and its associated thread may be exploited by the scheduler to improve data locality. Knowing the data accessed by threads that most recently executed on each processor, the scheduler can assign a thread that is ready to execute to the processor with the largest degree of data overlap.

Acknowledgments

This work has been supported by research grants from NSERC and CITO. The authors are grateful to the anonymous reviewers for their useful and constructive comments.

Notes

1. At present, there is no support for native methods.
2. Non-native threads, or the so-called "green threads" in Java, are scheduled by the JVM at the user level, and cannot actually make use of multiple processors.
3. Each benchmark is executed multiple times and the average execution time is used to compute the speedup.
4. Specifically, each of the *m* threads calls `Math.sine` *n* times on the value of a shared `Double` object.
5. `java.util.Arrays.sort ()` implements a tuned quicksort that offers $n \times \log(n)$ performance on most data sets [5].
6. The 15-puzzle is a game in which 15 tiles numbered 1 to 15 are placed in random order within a 4×4 grid, leaving one tile space empty. The player is required to arrange the tiles in ascending order of their numbers by repeatedly sliding tiles, one at a time, into the empty space.

References

1. T. Abdelrahman and S. Huynh. Exploiting task-level parallelism using ptask. In *Proceedings of Parallel and Distributed Processing Techniques and Applications*, pp. 252–263, 1996.
2. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.
3. D. Callahan and B. Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Proceedings of Languages and Compilers for Parallel Computing*, pp. 95–113, 1990.
4. B. Chan. Run-time support for the automatic parallelization of Java programs. Master's thesis, University of Toronto, 2002.
5. P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries*, 2nd edn. Vol. 1, Supplement for Java 2 Platform, Addison Wesley, Reading, MA, 1999.
6. J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 1–19, 1999.
7. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 230–241, 1994.
8. M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 212–223, 1998.
9. R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Proceedings of the Symposium on Principles of Programming Languages*, pp. 121–133, 1998.
10. T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel and Distributed Technology: Systems and Applications*, 2(3):16–26, 1994.
11. M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.
12. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of Workshop on Program Analysis For Software Tools and Engineering*, pp. 54–61, 2001.
13. J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
14. W. Hui, S. MacDonald, J. Schaeffer, and D. Szafron. Visualizing object and method granularity for program parallelization. In *Proceedings of Parallel and Distributed Computing and Systems*, pp. 286–291, 2000.
15. S. Huynh. Exploiting task-level parallelism automatically using pTask. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 1996.
16. M. Lam and M. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pp. 94–105, 1991.

17. J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 21–34, ACM, 1988.
18. T. Lewis and H. El-Rewini. *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, NJ, 1992.
19. K. H. Randall. Cilk: Efficient multithreaded computing. Ph.D. Thesis, MIT, Department of Electrical Engineering and Computer Science, 1998.
20. R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pp. 72–83, 1999.
21. G. Sohi and A. Roth. Speculative multithreaded processors. *IEEE Computer*, 34(4):66–73, 2001.