Run-Time Support for the Automatic Parallelization of Java Programs

Bryan Chan and Tarek S. Abdelrahman
The Edward S. Rogers Sr.
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{chanb,tsa}@eecg.toronto.edu

Abstract

The *zJava* project aims to develop automatic parallelization technology for programs that use pointer-based dynamic data structures, written in Java. The system 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 an invoked method may execute as an independent thread. The goal of this paper is to describe this run-time component of the *zJava* system and to report initial experimental results. In particular, the paper describes how the results of compile-time analysis are used at run-time to detect and enforce dependences among threads. Experimental results on a 4-processor Sun multiprocessor indicate that linear speedup may be obtained on sample applications and hence, validate our approach.

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 [1, 2]. 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. These features make it difficult to directly utilize parallelizing compiler technology developed for array structures and simple loops.

The goal of the *zJava* (pronounced "zed Java") project, at the University of Toronto, is to investigate automatic parallelization technology for programs that use pointer-based dynamic data structures and recursion, written in Java. The *zJava* system uses a novel combined compile-time/run-time approach to automatically exploit parallelism among methods in a Java program. Methods in the sequential program 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 at class load time. The run-time system uses this description to determine if and when an invoked method may execute as an independent thread. Although such an approach increases run-time overhead, it allows automatic parallelization of pointer-based programs, where compile-time-only approaches [3] have been limited.

The goal of this paper is to describe the run-time component of the *zJava* system and to report initial experimental results. In particular, the paper describes how the results of a compile-time analysis may be used at run time to detect dependences among threads and enforce dependences. The paper also presents experimental results from our implementation of the run-time system on a 4-processor Sun Ultra 4 machine. These initial results indicate that scalable performance can be obtained, and hence, validate our approach.

The remainder of this paper is organized as follows. Section 2. gives an overview of the *zJava* system and how it exploits parallelism in sequential Java programs. Section 3. describes data access summaries that are used to inform the run-time system of data used by methods in a program. 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 the 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 communicate 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 communicate by accessing (reading/writing) data in the shared memory. In a Java program, global variables are accessible by all methods, and thus are shared by all threads. 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 they 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 execute 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 the paths into a *data access summary* for each method. The compiler also transforms the input program into a parallel threaded program that contains calls to routines in the run-time system, which create and execute threads. The run-time system also contains code to compute run-time data dependences among threads from the data access summaries of associated methods, which are communicated to the run-time system when classes are loaded by the JVM. 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 the data accessed in a method. We then explain how data access summaries are formed from symbolic access paths. Finally, we give an example to illustrate these concepts.

A *symbolic access path* [4, 5] is a pair $o.f$ 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* and the object pointed to by f_{n-1} is the *destination* of the path; f_n in the destination object is a field variable, which may be of any type.

The source of an access path in a method m may be one of four kinds of objects:

1. a global object.
2. an object passed to m as an actual parameter, or the receiver object itself (i.e., `this`).

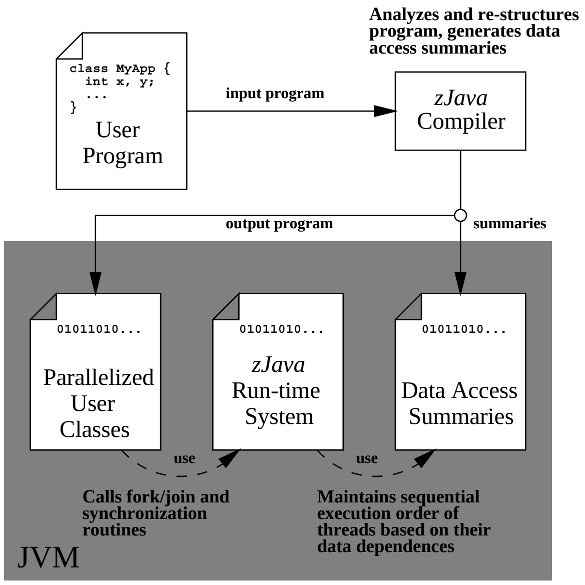
Analyzes and re-structures program, generates data access summaries
class MyApp {
int x, y;
...
}
User Program
input program
zJava Compiler
output program
summaries
01011010...
Parallelized User Classes
zJava Run-time System
Data Access Summaries
use
use
Calls fork/join and synchronization routines
Maintains sequential execution order of threads based on their data dependences
JVM

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

3. the object returned by a method called in the body of m.
4. a local object constructed within m, but does not *escape* m (i.e., the object's lifetime ends when the method exits [6]).

The *zJava* compiler need not generate symbolic access paths for local objects that do not escape the method (kind 4 above) because they are not shared among threads. Further, the compiler does not generate symbolic access paths for objects returned by called methods (kind 3 above) because the run-time system uses a mechanism called *future* to synchronize accesses to them without relying on data access summaries, as will be discussed in Section 4.3.4. Consequently, the compiler generates symbolic access paths for only the first two kinds of objects.

It should be noted that objects created locally inside a method, but do escape the method, can only escape it through global variables, instance fields of parameters, or return values. Hence, accesses to these escaping objects will be captured by the symbolic access paths of these variables and/or by futures.

The *zJava* compiler analyzes the body of a method to generate a symbolic access path for every variable accessed in the method, and these symbolic access paths 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 determined until run time. Therefore we use a symbolic notation to represent the source object of each 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. The special anchor "$0" denotes the receiver object, passed to the method as the implicit parameter `this`. The source object of a symbolic access path can be determined at run time using its anchor and the context of the method call.

In some cases, it may not be possible to compute the

```
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 = nextId++;
        return r;
    }
}
```

Figure 2. `midpoint`, a simple method.

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

Figure 3. Data access summary for the `midpoint` method.

length of the actual access path of a method because, for example, it traverses a recursive data structure, such as a linked list. The traversal is often terminated by a conditional branch that is only evaluated at run time. At compile time, we represent such an access with the symbolic access path: `$0.head{.next}*`. The `{.next}` component in the symbolic access path is a 1-limited representation of the recursive traversal of the linked list using the `next` pointer field.

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, e.g., `$0.data.[0-9]` specifies the first ten elements in the array `this.data`. This has the advantage of allowing multiple threads to concurrently read or write different regions of the array if those regions do not overlap.

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 or 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 value 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, the its access type is indicated as a 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 body. This is necessary to ensure proper synchronization, as will be seen in Section 4.3.2.

The Java code shown in Figure 2 is used to illustrate symbolic access paths and 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).

The data access summary for the `midpoint` method is shown in Figure 3. The first two entries indicate that the fields `x` and `y` of the receiver object is read by the method. The next two entries indicate that the same fields of the parameter `p` are also read. The last entry is an example of an access to a static variable; the fully qualified name of the variable is included as the anchor in the symbolic access path. It is marked as "write" since the method reads and then increments the value in 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 forking, synchronizing and terminating threads, are described.

4.1 Regions

While the *zJava* compiler refers to variables with symbolic access paths, the run-time system translates the access paths into *regions*, which represent variables in the system and describe how they are shared. Each region corresponds to an area in the program's memory that potentially is accessed by multiple, possibly concurrent, threads. Section 4.3.1 describes how regions are created from data access summaries.

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*, to coordinate the creation of regions and synchronization of threads. A *region node* r_v corresponds to 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. This *thread list* is sorted by serial execution order of the methods associated with the threads, as will be described in Section 4.3.2. The 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.

Figure 4 shows a graphical view of a simple registry. It contains three region nodes, corresponding to the variables

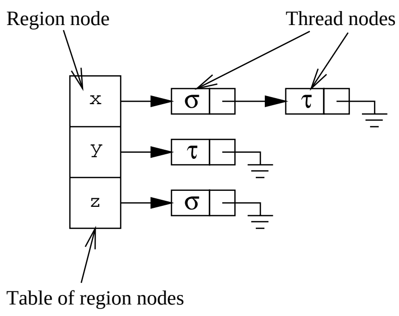

Figure 4. The *zJava* registry maintains sorted thread lists for shared regions.

x, y and z, and two thread nodes labeled τ and σ. Region x is accessed by both threads, and σ must access it before τ. Thread τ also accesses the region y, and σ also accesses the region z.

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

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. The run-time system creates regions corresponding to a summary when the associated method is invoked.

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 in two steps. The first step is to *resolve* each symbolic access path in the data access summary of the method m_σ. This is done 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. Consequently, the sources of each access path, and hence, the actual objects that will be accessed by σ are determined.

The second step is to *expand* each resolved access path into one or more regions. The access path $o.f_1 \ldots f_n$ is expanded into n access paths, namely, $o.f_1$, $o.f_1.f_2$, $o.f_1.f_2.f_3$ and so on, up to $o.f_1 \ldots f_n$. This is because, to access the field $o.f_1 \ldots f_n$, the fields f_1 to f_{n-1} must be read, and consequently must be represented with individual region nodes in order to synchronize threads that may access them. The 1-limited 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. Expansion stops when the current next field contains null.

Finally, 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 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 a, b, d and e

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

Figure 5. A matrix multiplication example.

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

Figure 6. Data access summary for readMatrix.

As an example, consider the program in Figure 5, in which the main method invokes the readMatrix method on the object a. For each symbolic access path in the data access summary of readMatrix (Figure 6), the run-time system replaces its anchor with the actual reference-type argument to the method; i.e., $0.nRows gets resolved to a.nRows, etc. Since all the symbolic access paths of the readMatrix method are only one field in length, the path expansion process is simple. 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 7.

4.3.2 Thread Forking

Spawning a child thread involves determining regions accessed by the child thread, allocating thread nodes for the child, and inserting thes nodes on the thread lists of the corresponding region nodes according to serial execution order.

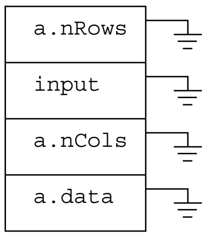

Figure 7. Region nodes are added for all regions that the `readMatrix` method will access.

When a thread τ spawns a child thread σ, the resolved and expanded access paths of σ is used to determine if it can proceed to execute. Thread nodes for the child thread are inserted into each of the thread list of the region nodes corresponding to these access paths. Once this process is done, the parent thread τ can continue on its own execution. Thread σ can proceed to execute when it acquires the appropriate locks on its regions.

Prior to inserting a thread node t_σ into the thread list of a region $o.f_1 \ldots f_n$, the run-time system must ensure that τ has unblocked read access to $o.f_1 \ldots f_{n-1}$, i.e., there is not a node of another writer thread in front of the thread node t_τ in the thread list of the region node of $o.f_1 \ldots f_{n-1}$. This is necessary because if a reference f_i in an access path is being modified by another writer thread when the thread σ is spawned, t_σ may be inserted into the thread list of a wrong region node. Hence, the run-time system iterates over the fields $f_1 \ldots f_n$ in the path and ensures that for every field f_i, the parent thread has read access to f_i, i.e., τ has acquired the read lock on the region r_{f_i}. If a region node exists for a field variable f_i, but τ does not have a read lock on the region, then τ is blocked. When τ acquires the read lock after f_i has been modified, it can read the updated reference and insert t_σ into the thread list of the region $o.f_1 \ldots f_i.f_{i+1}$.

Thread nodes in a thread list must be kept sorted by serial execution order of the associated threads to preserve program correctness. For two threads t_1 and t_2, we say that $t_1 > t_2$ if t_1 should access a variable before t_2 when the corresponding methods are invoked in the original sequential program. Since we require that the read and write sets of a parent thread contain the union of those of its child threads, for a given thread t in the thread list of a variable v, all of its ancestor threads must also be in the same thread list. Suppose t is the most recent child thread of its parent p. Since, in a sequential program, a called method must terminate before the calling method resumes, once t has been spawned, t must finish accessing v before p accesses v again, even though p may have been using the variable before t was spawned. Thus, $t > p$. By this reasoning, since the main thread is the ancestor of all threads in a program, its thread node is always the last one of every thread list.

Now suppose p spawns a new thread t', a younger sibling of t, so that $t > t'$. We can see that $t' > p$, for the same reason that $t > p$. Hence the only possible position in the thread list that t' can be placed is between t and its parent p. Consequently, to maintain serial execution order in any given thread list, a newly spawned thread must be placed immediately in front of its parent thread. A more formal proof may be found in [7].

Consider the naïve matrix multiplication program in Figure 5. The `readMatrix` and `multiply` methods op-

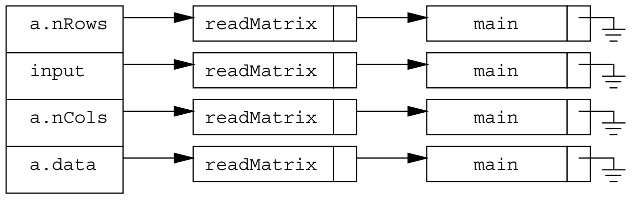

Figure 8. New thread nodes are created for the `readMatrix` thread; the nodes are inserted into the thread lists corresponding to regions accessed by the thread,

erate 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 8.

It should be noted that as a new thread node is inserted into a thread list, the nodes for all its ancestors are also inserted, if they are not already. This is consistent with the requirement that the data access summary of a method include those of its callees. Thus, in Figure 8, 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 9) to be created. Region nodes are allocated for `b.nRows`, `b.nCols`, `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. The thread node for $\texttt{multiply}_1$ is then inserted into the thread lists of these regions. It is also inserted into the thread list of the existing region node for `a.nRows` and `a.data`. The new thread node is placed behind the node of its sibling thread running `readMatrix`, because in serial execution order, the method `readMatrix` must exit 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`. 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 new thread after the ones that were spawned before itself. The resulting registry is shown in Figure 9.

4.3.3 Thread Synchronization

As described earlier, every region node in the registry is associated with a reader/writer lock. A thread that accesses a region but does not write to it is called a reader thread.

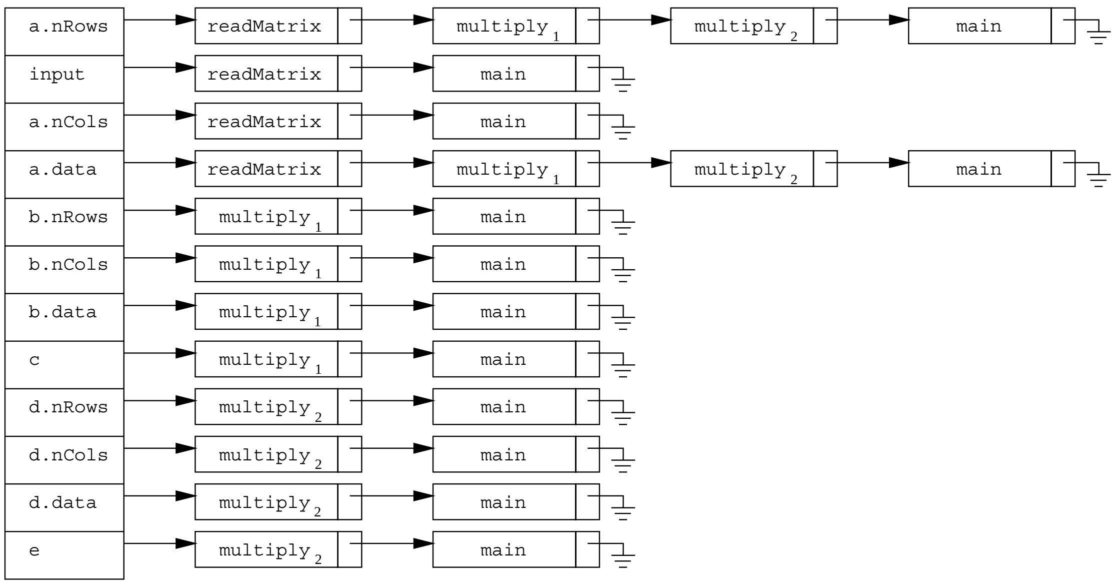

Figure 9. The registry is updated as the two `multiply` threads are spawned; the thread lists of shared regions keep the threads in serial execution order.

A thread that accesses a region and possibly writes to it is called a writer thread.

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. 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 granted a reader lock immediately if its thread node t_τ is the first in the thread list of r_x, or if the predecessor of t_τ belongs to another reader thread which is already running; otherwise τ waits for its predecessor to signal it. When τ acquires a 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 σ may overwrite its value. This ensures the correctness of the execution because data dependences are preserved. 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 5. Suppose the program is restructured to fork 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.

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

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` 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 10 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.

4.3.4 Thread Termination

The *zJava* compiler does not insert code into the compiled program to terminate threads explicitly. A thread terminates when the method the thread is executing exits.

At the exit of a method, the thread stores any return value of the method in a *future* [8], waits for its child threads to terminate, and finally removes its own thread node from the registry. 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 at all) by the parent thread.

5. Experimental Evaluation

We implemented the *zJava* run-time system in Java to make it portable. The system is implemented as a user library and, hence, requires no specific changes to the JVM. We evaluated the performance of the system on a Sun Ultra 4 machine, equipped with four processors. In this section, we report the performance of the run-time system using two benchmarks. For both benchmarks, the data access summaries were manually computed. Each program was also manually restructured to transmit the data access summaries to the run-time system and to insert thread forking and synchronizations calls.

The first benchmark, called `ReadTest`, 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. It consists of a loop that spawns m threads, each of which reads (but does not write) only one object and performs n computations with it[1]. Hence the benchmark is a good indicator of the overhead of parallelizing a program without thread blocking. Its performance provides an upper bound on the speedups that can be obtained by the system.

The second benchmark, called `Matrix` is the matrix multiplication application described earlier in the paper. It is used to demonstrate the performance of *zJava* in a somewhat more realistic scenario. It's is a slightly modified version of the one shown in Figure 5. Although it is mainly an array-based application, the code itself uses references and dynamically creates its objects. Traditional array-based parallelizing compiler technology would fail to to extract parallelism in this application, even though it manipulates arrays in loops.

The speedup (defined as the ratio of the execution time of the sequential program to the execution time of the parallel program) of `ReadTest` on 4 processors is shown in Figure 11 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 10 milliseconds). In fact, the performance reflects a

[1]Specifically, each of the m threads calls `Math.sine` n times with the value of a shared `Double` as the argument.

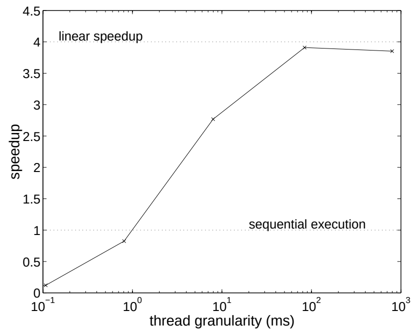

Figure 11. The effect of thread granularity on the performance of the run-time system.

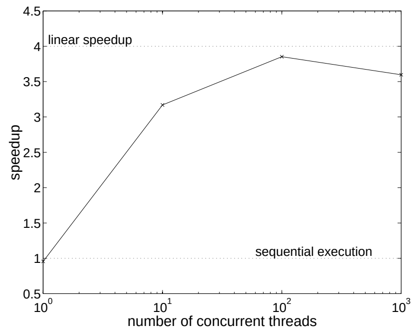

Figure 12. The effect of the number of threads on the performance of the run-time system.

slow-down in execution time. This is because at small granularities, the start-up and synchronization costs outweigh benefits gained from parallelism. However, when the granularity of a thread becomes greater than 50 milliseconds, close to linear speedup is achieved. Indeed, for thread granularity greater than 100 milliseconds, the speedup is practically linear.

The effect of the number of threads in the `ReadTest` benchmark (i.e., the value of m) is shown in Figure 12. 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 small (about 100), linear speedup can be achieved. However, when the number of threads is increased, performance slightly degrades, due to the contention resulting from sharing the registry and from the locking of regions and thread lists by parent threads as they create child threads.

The speedup of the `Matrix` application is shown in Figure 13. The program spawns 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, 1000 threads are spawned, each of which performs a million additions and multiplications. The speedup of for the largest matrix size is 3.6 at 4 processors.

The speedup of `Matrix` for a 1000×1000 matrix for different number of processors is shown in Figure 14. Although the speedup of `Matrix` is less than linear (3.6 at 4 processors), the speedup increases linearly with the number of processors.

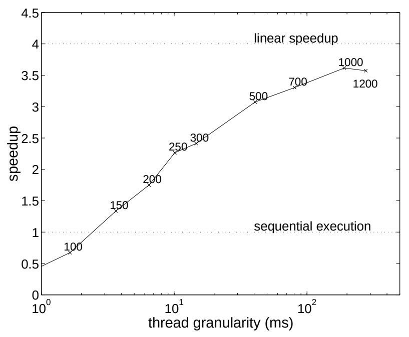

Figure 13. Speed-up of matrix multiplication on 4 processors as a function of matrix size (number of threads).

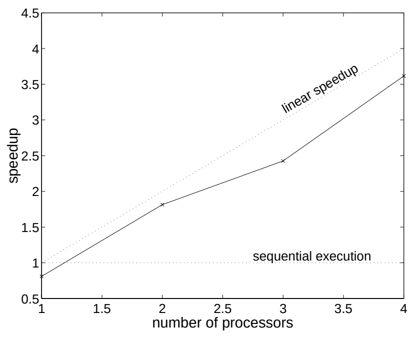

Figure 14. Speed-up of matrix multiplication for a 1000×1000 matrix as a function of the number of processors.

The speedup results of both the micro-benchmark and the matrix multiplication program indicate that scalable performance may be achieved when the number of threads is small and the granularity of each thread is moderate.

6. Related Work

Abdelrahman and Huynh [9] 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 [10] describe the Jade system, which allows parallelization of C programs at various granularities. Their work requires that programs be manually annotated to specify parallelism and synchronization. In contrast, we extract and represent shared data access from the program and automatically exploit parallelism.

Rugina and Rinard [11] employ compile-time-only analysis to automatically exploit parallelism in array-based programs that use divide-and-conquer. In contrast, we exploit parallelism in non-array-based programs, albeit at higher overhead at run-time.

Deutsch [4] devises a version of symbolic access paths that can capture object access information more accurately and more concisely than ours, and uses it for interprocedural may-alias analysis. Nonetheless, our work extends the use of access paths for computing run-time data dependences among concurrent threads.

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. The *zJava* compiler performs a similar analysis to determine objects shared among threads.

7. Concluding Remarks

The *zJava* system exploits parallelism among methods in sequential Java programs. In this paper, we described the design and implementation of the run-time component of the system, and presented initial experimental results. The run-time system utilizes a central data structure, called the registry, to maintain threads in the system and to enforce synchronization. Initial experimental results indicate that linear speedup may be obtained when the number of threads created by the system is small and the granularity of the threads is moderate.

We are currently profiling the run-time system to gain a better understanding of the sources of overhead, and hence formulate a plan to further improve the performance of the run-time system. We will then experiment with larger and more realistic applications and examine performance for large numbers of processors.

References

[1] W. Blume and et al., “Parallel programming with Polaris,” *IEEE Computer*, vol. 29, no. 12, pp. 78–82, 1996.

[2] M. Hall and et al., “Maximizing multiprocessor performance with the SUIF compiler,” *IEEE Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[3] R. Ghiya and L. Hendren, “Putting pointer analysis to work,” in *Proc. of POPL*, pp. 121–133, 1998.

[4] A. Deutsch, “Interprocedural may-alias analysis for pointers: Beyond k-limiting,” in *Proc. of PLDI*, pp. 230–241, 1994.

[5] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt, “The Geneva Convention on the treatment of object aliasing,” *OOPS Messenger*, vol. 3, no. 2, pp. 11–16, 1992.

[6] J.-D. Choi and et al., “Escape analysis for Java,” in *Proc. of OOPSLA*, pp. 1–19, 1999.

[7] B. Chan, “Run-time support for the automatic parallelization of Java programs,” Master’s thesis, University of Toronto, 2001. In progress.

[8] D. Callahan and B. Smith, “A future-based parallel language for a general-purpose highly-parallel computer,” in *Proc. of LCPC*, pp. 95–113, 1990.

[9] T. Abdelrahman and S. Huynh, “Exploiting task-level parallelism using ptask,” in *Proc. of PDPTA*, pp. 252–263, 1996.

[10] M. Lam and M. Rinard, “Coarse-grain parallel programming in Jade,” in *Proc. of PPoPP*, pp. 94–105, 1991.

[11] R. Rugina and M. Rinard, “Automatic parallelization of divide and conquer algorithms,” in *Proc. of PPoPP*, pp. 72–83, 1999.