`javar` – a prototype Java restructuring compiler*

Aart J.C. Bik and Dennis B. Gannon
Computer Science Dept., Indiana University
Lindley Hall 215, Bloomington, Indiana 47405-4101, USA
`ajcbik@cs.indiana.edu`

Abstract

In this paper we show how implicit parallelism in Java programs can be made explicit by a restructuring compiler using the multi-threading mechanism of the language. In particular, we focus on automatically exploiting implicit parallelism in loops and multi-way recursive methods. Expressing parallelism in Java itself clearly has the advantage that the transformed program remains portable. After compilation of the transformed Java program into bytecode, speedup can be obtained on any platform on which the implementation of the JVM (Java Virtual Machine) supports the true parallel execution of threads. Moreover, we will see that the transformations presented in this paper only induce a slight overhead on uni-processors.

1 Introduction

To obtain true portability, a program written in the Java programming language [13] is compiled into the architectural neutral instructions (bytecode) of an abstract machine, the JVM (Java Virtual Machine) [21], rather than into native machine code. In this manner, a compiled Java program can run on any platform on which an implementation of the JVM is available. Although the interpretation of bytecode is substantially faster than the interpretation of most high level languages, still a performance penalty must be paid for portability. For many interactive applications, this is not a major drawback. In other situations, however, performance may be more essential. In these cases, so-called 'just-in-time compilation' can be useful, where *at run-time* the bytecode is compiled into native machine code. With this approach, performance close to the performance of compiled languages can be obtained. However, because the demand for more computing power is likely to remain, other means to speedup Java programs have to be found.

In this paper, we show how some forms of implicit parallelism in Java programs can be made explicit by a restructuring compiler using the multi-threading mechanism of the language. In particular, we focus on automatically exploiting implicit parallelism in loops and multi-way recursive methods. Obviously, letting a compiler deal with the transformations that make implicit parallelism explicit simplifies the task of the programmer and makes the parallelization less error-prone. Moreover, because parallelism is expressed in Java itself, the transformed program remains portable and speedup can be obtained on any platform on which the Java bytecode interpreter supports the true parallel execution of threads (typically a shared-address-space architecture [17]), whereas we will see that the transformations presented in this paper only induce a slight overhead on uni-processors.

In figure 1, we illustrate our approach to automatically exploiting implicit parallelism in Java programs. A Java program `MyClass.java` is used as input of our source to source Java restructuring compiler `javar`.

*This project is supported by DARPA under contract ARPA F19628-94-C-0057 through a subcontract from Syracuse University. This paper has been published as journal paper in [6].

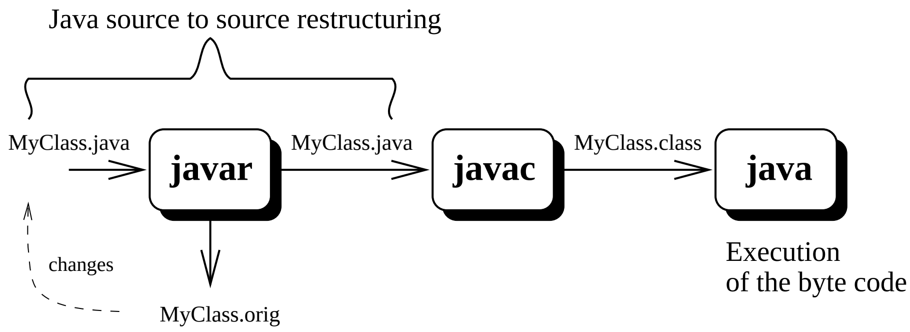

Figure 1: Restructuring, compiling, and interpreting

First, the compiler identifies the loops and the multi-way recursive methods that can exploit implicit parallelism. Parallel loops are either detected automatically by means of data dependence analysis, or are identified explicitly by the programmer by means of annotations. Because automatically detecting implicit parallelism in multi-way recursive methods can be very hard, in this paper we simply assume that such parallelism is always identified explicitly by means of annotations. Thereafter, the compiler transforms the input program into a form that uses the multi-threading mechanism of Java to make all implicit parallelism explicit. Because parallelism is expressed in Java itself, the transformed program can still be compiled into bytecode by any Java compiler (`javac` in the figure), and subsequently interpreted by any bytecode interpreter (`java` in the figure, or, alternatively, an interpreter that is embedded in a browser or appletviewer). Since filenames are essential in Java, the transformed program is stored in the file `MyClass.java` after a copy of the original program has been saved in a file `MyClass.orig`. In case changes to the original program are required, this latter file must be renamed into `MyClass.java` again before `javar` can be re-applied to the program.

The rest of this paper is organized as follows. First, in section 2, we give some preliminaries on the threading mechanism of the Java programming language. In section 3, we show how a restructuring compiler can exploit implicit loop parallelism in stride-1 loops. Thereafter, in section 4 we discuss how implicit parallelism in multi-way recursive methods can be made explicit. In section 5, we presents the results of a series of experiments that show the potential of the transformations presented in this paper. Finally, we state conclusions in section 6.

2 Multi-Threading in Java

The Java programming language supports so-called multi-threading (see e.g. [2, 10, 12, 13, 18, 21, 22, 26, 27, 28, 29, 38] for a more detailed presentation), where several threads that share a single address space may run concurrently. The most straightforward way to obtain a new thread in Java is by calling the method `start()` on an instance of a subclass of `java.lang.Thread` that overrides the `run()`-method of this latter class with appropriate code. This causes a new thread to execute this `run()`-method, while the original thread continues asynchronously with the code that follows the invocation of `start()`. A thread can wait for completion of another thread by calling the method `join()` on this thread. The following fragment, for instance, effectively implements the FORK/JOIN-construct illustrated in figure 2:

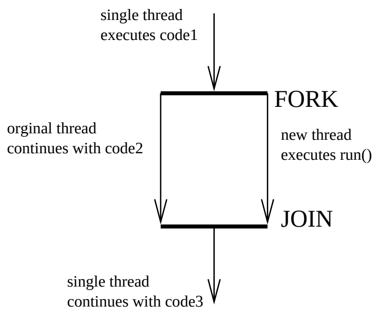

Figure 2: A FORK/JOIN-construct

```
code1;
MyThread t = new MyThread();
t.start();
code2;
t.join();   // may throw InterruptedException
code3;
```

```
class MyThread extends Thread {
  public void run() {
     ...
  }
}
```

The Java programming language supports a synchronization mechanism that is based on the concept of monitors [15]. Mutual exclusion amongst different threads that call methods with the qualifier '`synchronized`' on the same object is guaranteed. Once a lock of a particular object has been acquired, a thread can suspends itself by calling the `wait()`-method on this object. This will release the lock of this object, allowing other threads to obtain the lock. If a thread that has acquired the lock of a particular object calls the `notify()`-method, an arbitrary thread that has formerly been suspended by calling the `wait()`-method on this object will awake again, although it must re-acquire the lock of the object before it may continue. Likewise, calling the `notifyAll()`-method will awake all suspended threads, each of which must re-acquire the lock before it may continue.

An implementation of the JVM *generally* uses pre-emptive, priority based scheduling for threads, where the currently running thread always has highest priority among all *runnable* threads, and as soon as a thread with higher priority becomes runnable, the running thread is pre-empted in favor of the latter. However, the specification does not guarantee fairness amongst runnable threads with equal priority. Furthermore, the specification requires that any variable that is read or written to *synchronously* is explicitly loaded and stored back at acquiring and releasing locks, so that changes to shared data become visible to other threads that access these variables by means of synchronized methods. *Asynchronously* accessed variables, on the other hand, must be declared volatile to prevent the compiler from performing optimizations that disable the visibility of such changes (e.g. by keeping a variable that is only read in a register).

3 Loop Parallelization

In this section, we first briefly review some issues related to parallel loops. Thereafter, we discuss how a restructuring compiler can exploit implicit DOALL- and DOACROSS-like parallelism in Java by means of multi-threading. Because data dependence theory and analysis are discussed extensively in the literature (see e.g. [3, 4, 5, 7, 16, 20, 25, 30, 31, 33, 34, 39, 40, 41]), we do not further elaborate on these issues. Instead, we simply assume that the restructuring compiler has some mechanism to identify parallel loops in a program.

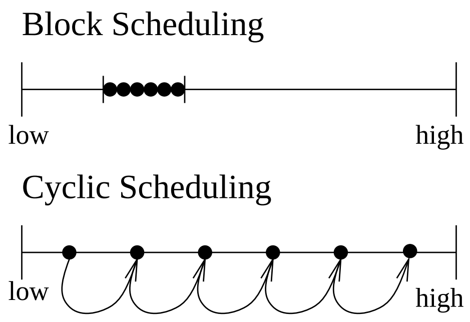

Figure 3: Pre-Scheduling Policies

3.1 Parallel Loops

If all iterations of a loop are independent, i.e. *if no data dependence is carried by the loop*, then this loop can be converted into a DOALL-loop. Even if data dependences are carried by a loop, however, some parallelism may result from executing the loop as a DOACROSS-loop, where a partial execution of some (parts) of the iterations is enforced to satisfy the loop-carried data dependences.

Although there are several methods to enforce synchronization in a DOACROSS-loop (see e.g. [8, 9, 23][24, 25, 40]), in this paper we focus on **random synchronization** [39, p75-83][41, p289-295], where synchronization variables are implemented as bit-arrays that provide one bit for each iteration. Synchronization is enforced using non-blocking **post**-statements to set particular bits of synchronization variables and **wait**-statements to block until certain bits of these synchronization variables become set.

On shared-address space architectures, parallel loops can be executed using fork/join-like parallelism to start a number of threads that will execute the different iterations of the loop in parallel [40, 385-387]. The way in which iterations are assigned to the threads is dependent on the scheduling policy [33, ch4][39, p73-74][40, p387-392][41, 296-298]. In a **pre-scheduling policy**, iterations are assigned statically to threads, for instance, in a block-wise or cyclic fashion (viz. figure 3). In a **self-scheduling policy**, threads enter a critical section to obtain a next chunk of iterations dynamically. Here, there is a clear trade-off between using small chunks to reduce the potential of load imbalancing, and using large chunks to limit synchronization overhead. A good comprise is to vary the chunk size dynamically. In guided self-scheduling, for example, $1/t$ of the remaining iterations are assigned to each next thread, where t denotes the total number of threads that are used to execute the parallel loop.

3.2 Class Hierarchy for Parallel Loops

In figure 4, we present a class hierarchy for implementing parallel loops in Java. Class `Thread` is provided in package `java.lang` of the Java API. The second layer of this hierarchy is also independent of the source program, and can be provided in another immutable package. For each particular parallel loop in a program, a new class `LoopWorker_x` is constructed explicitly by the restructuring compiler and added to the transformed program.

3.2.1 Schedules

The interface `Schedules` is used to provide the class `LoopPool` with symbolic constants that represent different scheduling policies:

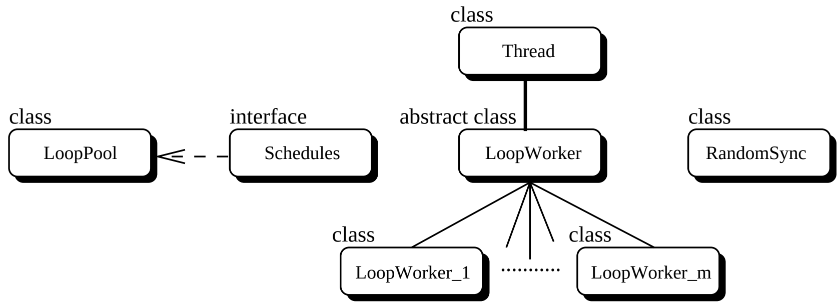

Figure 4: Class Hierarchy

```
interface Schedules {
  static final int SCHED_BLOCK  = 0;
  static final int SCHED_CYCLIC = 1;
  static final int SCHED_GUIDED = 2;
}
```

In the remaining of this paper, we focus on the implementation of three scheduling policies: block scheduling, cyclic scheduling, and guided-self scheduling. Other scheduling policies, however, can be easily incorporated in the framework.

3.2.2 LoopWorker

The abstract class `LoopWorker` provides an abstraction of a loop-worker that can be used to execute several iterations of a parallel loop. The instance variables `low`, `high`, and `stride` represent an execution set `[low,high)` and stride. Instance variables `pool` and `sync` provide a hook to a pool of iterations and a set of synchronization variables, respectively, shared amongst all loop-workers that are executing iterations of the same parallel loop:

```
abstract class LoopWorker extends Thread {
  int          low, high, stride;
  LoopPool     pool;
  RandomSync[] sync;
  ...
}
```

Because the class for loop-workers of a particular parallel loop must always be obtained by sub-classing this class to provide an appropriate `run()` method, instantiation of `LoopWorker` itself is prevented by making the class abstract.

Class `LoopWorker` also provides a class method `parloop()` that can be used to start the parallel execution of a loop. This method expects the lower and upper bound of a parallel stride-1 loop in two integer parameters `l` and `h`, some loop-workers in an array `w` (in fact, these loop-workers will always be workers of a specific `LoopWorker_x` class), the number of synchronization variables required in parameter `numS`, and, finally, the representation of a scheduling policy in parameter `sched`. First, new instantiations of a pool and the appropriate number of synchronization variables are obtained. Thereafter, these objects are made available to all loop-workers and a fork is performed. Finally, the method performs a join by waiting for all threads to finish:

```
static void parloop(int l, int h, LoopWorker[] w, int numS, int sched) {
  LoopPool     p = new LoopPool(l, h, w.length, sched);
  RandomSync[] s = new RandomSync[numS];

  for (int i = 0; i < numS; i++)
    s[i] = new RandomSync(l, h);

  // FORK
  for (int i = 0; i < w.length; i++) {
     w[i].pool = p;
     w[i].sync = s;
     w[i].start();
  }
  // JOIN
  for (int i = 0; i < w.length; i++) {
     try { w[i].join(); }
        catch (InterruptedException e) {}
  }
}
```

3.2.3 LoopPool

During execution of a parallel loop, loop-workers compete for work by accessing a shared pool of iterations, as is illustrated in figure 5. This structure of such a pool is defined by the class `LoopPool`. Two instance variables `low` and `high` are used to represent the execution set [`low`,`high`) of the parallel stride-1 loop, while the instance variables `numW` and `sched` are used to record the number of loop-workers and a kind of scheduling policy. Furthermore, the class provides two administration variables `blocksize` and `count`:

```
class LoopPool implements Schedules {
  int low, high, sched, numW, blocksize, count;
  ...
}
```

In the only constructor of this class, initial values are assigned to all instance variables:

```
LoopPool(int l, int h, int n, int s) {
  low   = l;
  high  = h;
  numW  = n;
  sched = s;

  blocksize = (int) Math.ceil(((double) (high-low)) / numW);
  count     = numW;
}
```

The next amount of work is obtained by calling instance method `nextWork()` on the pool. Depending on the kind of scheduling policy used, new values are assigned to

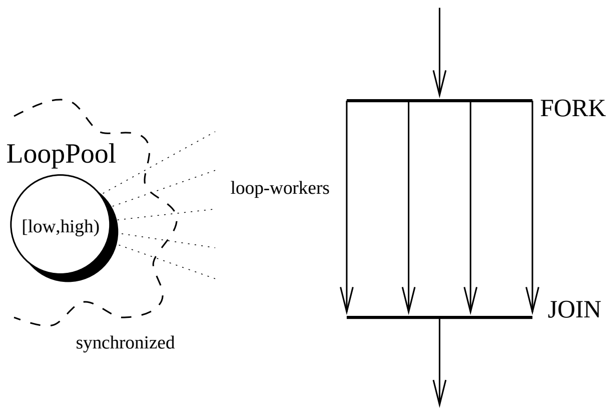

Figure 5: Execution of a Parallel Loop

the instance variables `low`, `high`, and `stride` of a loop-worker that is supplied in the parameter `worker`. If the pool has been exhausted, the method returns the value 'false'. Because a pool is shared amongst all loop-workers of a particular parallel loop, mutual exclusion while updating shared data is enforced by making the method synchronized:

```
synchronized boolean nextWork(LoopWorker worker) {
  boolean more = false;

  switch (sched) {

    case SCHED_CYCLIC:
      more          = (count-- > 0);
      worker.low    = low++;
      worker.high   = high;
      worker.stride = numW;
      break;

    case SCHED_GUIDED:
      blocksize = (int) Math.ceil(((double) (high-low)) / numW);

    case SCHED_BLOCK:                   // FALL THROUGH
      more          = (low < high);
      worker.low    = low;
      worker.high   = Math.min(low + blocksize, high);
      worker.stride = 1;
      low          += blocksize;
      break;
  }
  return more;
}
```

To obtain a uniform interface between loop-workers and a pool, in our framework the pre-scheduled policies are implemented as special versions of a self-scheduled policy, where each loop-worker directly obtains all work in the first call to `nextWork()` and terminates after the second call.

3.2.4 RandomSync

The class `RandomSync` defines an implementation of synchronization variables for random synchronization in DOACROSS-loops. A boolean arrays `bits` is used to implement the bit-array, while an integer `low` is used to record the lower bound of the execution set of the parallel loop:

```
class RandomSync {
  boolean[] bits;
  int       low;
  ...
}
```

The following constructor can be used to obtain a synchronization variable for a parallel stride-1 loop with execution set $[\mathtt{l},\mathtt{h})$. In this constructor, a new bit-array that has one bit for each iteration is obtained (viz. bit `(i-low)` belongs to iteration `i`), and the lower bound `l` is recorded:

```
RandomSync(int l, int h) {
  low = l;
  if (l < h)
    bits = new boolean[h-l];
}
```

Setting the bit of iteration `i` of a synchronization variable is done by calling the following synchronized instance method `doPost()` on this variable. Likewise, calling the synchronized instance method `doWait()` shown below on a synchronization variable blocks until the bit of iteration `j` of this variable becomes set. The test `low <= j`, however, makes this method non-blocking for out-of-bounds tests (which only may go backwards in the iteration space of the DOACROSS-loop):

```
synchronized void doPost(int i) {
  bits[i-low] = true;
  notifyAll();
}
```

```
synchronized void doWait(int j) {
  if (low <= j)
    while (! bits[j-low])
       try { wait(); }
         catch(InterruptedException e) {}
}
```

The call to `notifyAll()` will eventually wake up all threads that are blocked on some synchronization variable (although, of course, they only re-acquire the lock of the monitor one at the time). Because being notified does not necessarily mean that the appropriate bit actually has been set, however, the `wait()`-statement appears in a while-statement (rather than in an if-statement).

Although in the implementation shown above, waiting threads do not compete for processor time, the overhead of frequently accessing synchronized methods may be substantial. In this particular case, mutual exclusion while accessing the bit-array is not really required, and we can also implement the methods shown above using **busy-waiting** by eliminating the qualifiers `synchronized` and and replacing the `wait()`-statement by the call '`Thread.currentThread().yield()`'. In section 5.3 we will see that this approach can substantially improve the performance of a DOACROSS-loop. However, because the Java language specification does not guarantee fairness between runnable threads with equal priority, and in Java there is no way to declare the *elements* of an array as `volatile` (and since the elements of `bits` are accessed asynchronously, other processors may fail to see changes [13, ch.17]), this approach may be unsuited for particular implementations of the Java Virtual Machine.

3.3 Actual Loop Parallelization

Now, we are ready to discuss the steps that can be taken by a compiler to exploit implicit loop parallelism in a stride-1 loop of the following form, where we assume that `low` and `high` denote two arbitrary *loop-invariant* expressions of type `int`:

```
    class MyClass {
      ...
      [qualifiers] type myMethod(...) {
        ...
Li:     for (int i = low; i < high; i++)
           body(i);
        ...
      }
    }
```

Furthermore, we assume that the compiler has ascertained that loop `Li` can be executed in either a DOALL- or DOACROSS-like manner, possibly after standard compiler optimizations, loop normalization, and possibly other loop transformations (see e.g. [1, 11, 32, 33, 39, 40, 41] are used to enhance the opportunities of loop parallelization. The parallelization itself proceeds as follows, where, for simplicity, new identifiers that may conflict with other identifiers are denoted with a suffix '_x'. In reality, however, an appropriate suffix must be generated by the compiler.

3.3.1 Construction of run_x()

First, depending on whether `myMethod()` is a *class method* or an *instance method*, the restructuring compiler adds the following class or instance method `run_x()` to `MyClass`, where `body(i)` denotes a literal copy of the original loop-body. The `l1...lp` (with corresponding types `typei`) denotes the set of local variables and parameters of `myMethod()` that are referred to *within* `body(i)`, but that are declared *outside* the loop `Li`:

```
[static] void run_x(int l_x, int h_x, int s_x, RandomSync[] sync_x, type1 l1, ..., typep lp) {
for (int i = l_x; i < h_x; i += s_x)
   body(i);
}
```

Class and instance variables, the loop index `i`, and all other variables declared within the loop remain visible in `run_x()`. Hence, if the value of every `li` remains unaltered in the loop-body, the original loop can be executed in parallel by letting different threads invoke this method for different subsets of iterations and providing the appropriate value of each `li` as argument.

If, on the other hand, the value of any `li` *itself* may change (not counting changes to elements or fields of an array or object reference `li`), there is not likely much parallelism in the loop and the compiler simply resorts to disabling the loop parallelization to prevent the requirement to propagate such changes back to the main thread.

For DOACROSS-like execution, the compiler adds the appropriate synchronization primitives to the loop-body according to methods described in the literature [20, 23, 24, 25, 41]). Setting the bit of iteration `i` of a synchronization with number `k` is implemented as '`sync_x[k].doPost(i)`'. Likewise, waiting for the bit of iteration `j` of the synchronization variable with number `k` to become set is implemented as '`sync_x[k].doWait(j)`'.

3.3.2 Construction of LoopWorker_x

Subsequently, the restructuring compiler constructs the following class `LoopWorker_x`, and adds this new sub-class of `LoopWorker` to the transformed program:

```
class LoopWorker_x extends LoopWorker {
  MyClass target;
  type1   l1_x;
  ...
  typek   lp_x;

  LoopWorker_x(MyClass target, type1 l1_x, ..., typep lp_x) {
    this.target = target;
    this.l1_x   = l1_x;
    ...
    this.lp_x   = lp_x;
  }
  public void run() {
    while ( pool.nextWork() )
      target.run_x(low, high, stride, sync, l1_x, ..., lp_x);
  }
}
```

Since `target` is only required if `myMethod()` is an *instance method*, all constructs involving this instance variable are omitted if `myMethod()` is a *class method*. In this case, '`MyClass`' is used in the invocation of `run_x()` instead.

3.3.3 Loop Replacement

In the third and final step, the original loop `Li` is replaced by the following block of code, where `low` and `high` denote the lower and upper bound expression that are used in the original loop, and each `li` denotes a local variable or parameter that is declared outside this loop, and referred to (but unaltered) within the loop-body. If `myMethod()` is a class method, construct `this` is omitted from the constructor invocation:

```
{  LoopWorker_x[] worker_x = new LoopWorker_x[NUM];
   for (int i_x = 0; i_x < NUM; i_x++)
      worker_x[i_x] = new LoopWorker_x(this, l1, ..., lp);
   LoopWorker.parloop(low, high, worker_x, SVARS, SCHED);
}
```

Here, `NUM`, `SVARS`, and `SCHED` denote compiler selected literal constants that represent the number of loop-workers, the number of synchronization variables, and the kind of scheduling policy for this parallel loop, respectively.

3.3.4 Exception Handling

In principle, any exception that may be thrown during execution of the parallel loop and that is explicitly handled thereafter can be dealt with by catching such an exception in the `run()`-method of a loop-worker, storing this exception in an additional field `e_x` of the loop-worker, and re-throwing a caught exception after the invocation of `parloop()` generated in the third step (see section 3.3.3) as follows:

```
for (int i_x = 0; i_x < NUM; i_x++)
  if (worker_x[i_x].e_x != null)
    throw worker_x[i_x].e_x;
```

A single field of type `java.lang.Exception` can be used if all kinds of exceptions are handled explicitly after the loop, or some different fields in the loop-worker can be used to deal specifically with various types of exceptions. In any case, however, because the order in which iterations of a parallel loop are executed is completely unpredictable, the programmer must be aware that after loop parallelization, no assumptions about which iterations have or have not been executed successfully can be made in any subsequent explicit exception handling. In particular, several exceptions may occur in the different threads of a parallel loop, whereas only one of these exceptions will be re-thrown. Consequently, if the order in which exceptions may be thrown in a loop is essential, parallelization of this loop should be disabled. Otherwise, the mechanism sketched above can be used to transfer an exception that is thrown by a look-worker back to the main thread.

4 Multi-way Recursive Method Parallelization

In this section, we introduce the concept of a parallel multi-way recursive method and show how a restructuring compiler can exploit implicit parallelism in such methods in Java by means of multi-threading. Because automatically detecting parallel multi-way recursive methods can be very hard, we simply assume that the programmer uses annotations to identify all parallel multi-way recursive methods in a program.

4.1 Parallel Multi-way Recursive Methods

We refer to a *non*-private, *un*-synchronized, and *final* method (i.e. overriding or hiding is not possible) of the following form, where the different recursive method invocations in between executing `pre_code` and `post_code` can be done in parallel as a **parallel multi-way recursive method**:

```
class MyClass {
  ...
  [qualifiers] type myMethod(type1 f1, ..., typek fk) {
     if (cond)
       alt_code
     else {
       pre_code
       r1 = target1.myMethod(a11, ..., a1k)
       ...
       rn = targetn.myMethod(an1, ..., ank)
       post_code
     }
  }
}
```
(1)

Here, each `targeti` either denotes '`MyClass`' if `myMethod()` is a class method, or an arbitrary variable of type `MyClass` (including `this`) otherwise. If `myMethod()` is a `void`-method, there are no assignments to the different `ri`. Algorithms that traverse an explicit tree-like data structure or divide-and-conquer algorithms can usually be expressed in this form. Because in both cases, a virtual tree of method invocations is traversed, we will visualize the parallelization of such methods using trees.

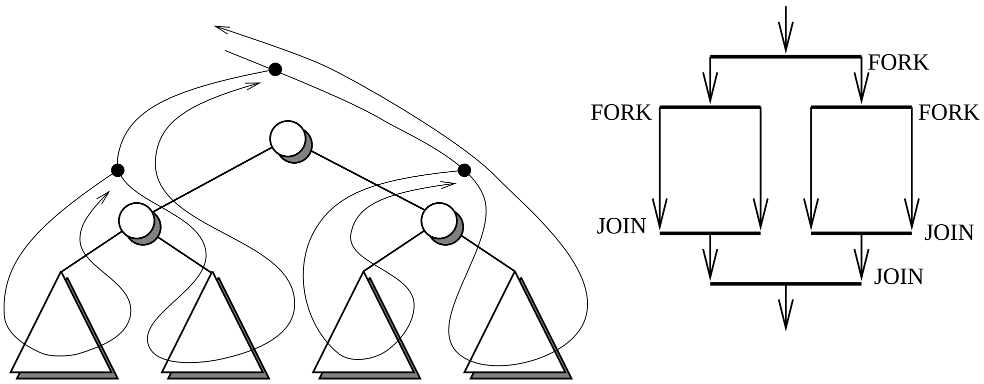

Figure 6: Static Allocation

The most straightforward way to exploit implicit parallelism in a parallel n-way recursive method is to let a running thread assign all but one of the recursive method invocations to other threads [17, 19, 35, 36]. Although our method is based on this simple approach, the analysis shown below reveals the limitation on the corresponding speed-up, and better ways of parallelizing an algorithm may exist. If $n = 2$, for example, and each invocation divides the input into two sets of (roughly) the same size in time proportional to the remaining input size, then this approach changes the serial execution time $T_s(N) = \Theta(N \cdot \log N)$ into the parallel execution time $T_p(N) = \Theta(N)$ using $p = N$ processors, as implied by the following recurrence relations (with $T_s(1) = T_p(1) = \Theta(1)$ for handling the base-case):

$$\begin{cases} T_s(N) & = & \Theta(N) & + & 2 \cdot T_s(N/2) \\ T_p(N) & = & \Theta(N) & + & T_p(N/2) \end{cases}$$

Hence, in this case the best possible speedup is $S = T_s(N)/T_p(N) = \Theta(\log N)$. Similar analysis reveals that for 2-way recursive methods in which executing `pre_code` and `post_code` takes constant time, the best possible speedup using the simple parallelization method sketched above is $S = \Theta(N/\log N)$.

The easiest way to assign work to a *limited number* of processors is to let running threads assign method invocations to other threads only in the top levels of the method invocation tree. Forking and eventually joining new threads in only the top two levels of the method invocation tree, for example, assigns the method invocations of a 2-way recursive to four processors, as illustrated in figure 6.

Although such a **static allocation scheme** may yield poor performance if the subtrees assigned to the different processors substantially vary in size, in this paper we simply rely on the fact that most multi-way recursive methods try to keep the method invocation tree reasonably balanced. Moreover, we will show that some load imbalancing can be alleviated by starting additional threads.

4.2 Actual Parallelization

In this section, we describe the steps that can be taken by a restructuring compiler to make implicit parallelism in a parallel multi-way recursive method of the form (1) explicit by means of multi-threading. Again, for simplicity, all new identifiers that may conflict with other identifiers are denoted with a suffix '`_x`'.

4.2.1 Construction of the Tree-Worker Class

In the first step, a sub-class `TreeWorker_x` of `java.lang.Thread` is constructed that provides an implementation of a tree-worker that can be used specifically to execute invocations of the final method `myMethod()` in parallel. Consequently, if a Java program contains m parallel multi-way methods, then m classes are constructed and added to the transformed Java program, as is illustrated in figure 7.

An instance variable `result` can be used to transfer the result of a method invocation. Instance variable `depth` will be used to record the current depth in the method invocation tree.

For each formal argument `fi` of `myMethod()`, there is an instance variable `fi_x` of the appropriate type. An instance variable `target` is possibly used to store a target. Finally, a constructor that initializes a new tree-worker and a `run()`-method that calls a new method `myMethod_par_x()` with the appropriate parameters are provided:

```
class TreeWorker_x extends Thread {
  type    result;
  int     depth;
  MyClass target;
  type1   f1_x;
  ...
  typek   fk_x;

  TreeWorker_x(int depth, MyClass target, type1 f1_x, ..., typek fk_x) {
    this.depth  = depth;
    this.target = target;
    this.f1_x   = f1_x;
    ...
    this.fk_x   = fk_x;
    start();
  }
  public void run() {
    result = target.myMethod_par_x(depth, f1_x, ..., fk_x);
  }
}
```

Constructs involving `target` are only required if `myMethod()` is an *instance method*. For a *class method*, these constructs are omitted and '`MyClass`' is used in the call to `myMethod_par_x()` instead. If `myMethod()` is a `void`-method, all constructs involving `result` are omitted.

4.2.2 Modification of the Method

Subsequently, `myMethod()` is converted into another method `myMethod_par_x()` that takes an additional integer parameter `d_x`:

```
[qualifiers] type myMethod_par_x(int d_x, type1 f1, ..., typek fk) {
  ...
}
```

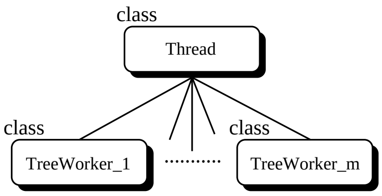

Figure 7: Class Hierarchy

The method-body is modified as follows. First, *all* invocations that appear in the method-body are renamed accordingly, and the expression 'd_x+1' is added as an initial parameter. Moreover, the first n-1 recursive method invocations of the n subsequent invocations that can be executed in parallel are rewritten into the following form, where CUT_DEPTH denotes some literal integer constant selected by the compiler:

```
TreeWorker_x wi_x = null;
if (d_x <= CUT_DEPTH)
  wi_x = new TreeWorker_x(d_x+1, targeti, ai1, ..., aik);
else
  ri = targeti.myMethod_par_x(d_x+1, ai1, ..., aik);
```

Here, the actual parameter targeti is omitted in case myMethod() is a class method, and 'ri =' is omitted in the else-branch if myMethod() is a void-method.

For each such rewriting, the following construct is generated before the post_code fragment to implement the appropriate synchronization. The assignment statement is omitted for a void-method:

```
if (wi_x != null) {
  try { wi_x.join(); }
     catch(InterruptedException e_x) {}
  ri = wi_x.result;
}
```

If ri is a local variable, it may be necessary to add a dummy assignment to the declaration of this variable to preserve the **definite assignment property** of Java [13], because the original assignment has been moved into two conditional statements.

After these transformations have been applied to an n-way recursive method, n-way forks will be performed in the top levels $0 \dots c$ of the method invocation tree in case CUT_DEPTH$= c$. In the other levels, each separate thread continues to execute method invocations in a serial fashion (at this stage, the overhead of passing d_x can be reduced by executing an *unaltered* copy of the original method). Hence, if p processors are available to execute a parallel n-way recursive method, then c should be at least $\lceil {}^{n}\log p \rceil$ to obtain sufficient threads. Using a slightly larger cut-depth, however, may be useful to alleviate load imbalancing problems. Threads eventually join with their originating threads, until the single thread that invoked the parallel recursive method remains. Note that the join is also required in case the code fragment post_code is empty to enforce the appropriate synchronization before the method as a whole terminates.

4.2.3 Construction of a new Method

In the last step, a new method myMethod() with the same qualifiers as the original method and of the form shown below is added to the class in which the original method appears, where the return is omitted for a void-method:

```
[qualifiers] type myMethod(type1 f1, ..., typek fk) {
  return myMethod_par_x(0, f1, ..., fk);
}
```

After these transformations, all method invocations of myMethod_par_x() have access to the appropriate depth within the method invocation tree.

Because the interface of myMethod() itself remains unaffected, all calls to the original method remain completely unaware of the transformations. Hence, the parallelization of myMethod() only involves some local transformations of the class MyClass. However, this also implies that the method is less suited to deal with *mutual recursion*, i.e. situations in which myMethod() calls another method that, in turn, invokes myMethod() again. In such cases a parallel execution will be re-initiated for each invocation of myMethod() in the other method.

4.2.4 Exception Handling

In principle, any exception that may be thrown during execution of the method and that is explicitly handled thereafter can be dealt with by catching such an exception in the `run()`-method of a tree-worker, storing this exception in an additional field `e_x` of the tree-worker, and re-throwing a caught exception in the join-construct generated during the second step (see section 4.2.2) as follows:

```
if (wi_x != null) {
  ...
  if (wi_x.e_x != null)
    throw wi_x.e_x
}
```

In this manner, the exception is caught and re-thrown by tree-workers until eventually the exception is explicitly handled in the body of the method or the exception reaches the main thread that initiated the parallel executed and is handled thereafter. A single field of type `java.lang.Exception` can be used if explicit handling for all kinds of exceptions is provided, or some different fields in the loop-worker can be used to deal specifically with various types of exceptions. In any case, however, because invocations of a parallel multi-way recursive method are executed in an unpredictable order, the programmer must be aware that after parallelization, no assumptions about which invocations have or have not been executed successfully can be made in any subsequent explicit exception handling. Hence, if the order in which exceptions in the method may be thrown is essential, parallelization should be disabled.

5 Experiments

In this section, we discuss the results of a number of experiments that have been conducted on an IBM RS/6000 G30 with four PowerPC 604 processors using the AIX4.2 JDK1.0.2b programming environment. Unless stated otherwise, programs are compiled into bytecode using the flag '-O', and subsequently interpreted using the flag '-noasyncgc' and with both just-in-time compilation and the parallel execution of threads enabled.

5.1 Some Loop Examples

We first discuss loop parallelization in full detail using some simple examples of single and double loops. In the examples, we assume that exceptions do not have to be dealt with.

5.1.1 Single Loops

Consider, for example, the following class `SimpleLoop1` in which an instance method `instanceMethod()` is used to assign the inverse of the first `N` elements in a floating-point array `b` to corresponding elements in another floating-point array `a`:

```
class SimpleLoop1 {
   double[] a;
   double[] b;
   void instanceMethod(int N) {
L1:  for (int i = 0; i < N; i++)
        a[i] = 1.0D / b[i];
   }
}
```

Obviously, because no data dependences are carried by loop `L1`, the loop can be executed in a DOALL-like manner. Moreover, because the parameter `N` is only used as upper bound, the compiler does not have to deal with any variables that are declared outside the loop, but that are refered to within the loop-body. Exploiting implicit loop parallelism proceeds as follows.

First, the following instance method `run_1()` is constructed and added to the class `SimpleLoop1` (see section 3.3.1):

```
void run_1(int l_1, int h_1, int s_1, RandomSync[] sync_1) {
  for (int i = l_1; i < h_1; i += s_1)
     a[i] = 1.0D / b[i];
}
```

Thereafter, the restructuring compiler constructs the following class `LoopWorker_1` and adds this class to the transformed Java program (see section 3.3.2):

```
class LoopWorker_1 extends LoopWorker {
  SimpleLoop1 target;
  LoopWorker_1(SimpleLoop1 target)  {
    this.target = target;
  }
  public void run() {
    while ( pool.nextWork(this) ) {
      target.run_1(low, high, stride, sync);
    }
  }
}
```

Finally, loop `L1` in `instanceMethod()` is replaced by the following construct, where `NUM=4` and `SVARS=0`. Because the same amount of work is done in each iteration, block-scheduling has been selected (see section 3.3.3, where adding '`implements Schedules`' to the transformed class enables the compiler to use the symbolic representation of a scheduling policy):

```
void instanceMethod(int N) {
  LoopWorker_1[] worker_1 = new LoopWorker_1[4];
  for (int i_1 = 0; i_1 < 4; i_1++)
    worker_1[i_1] = new LoopWorker_1(this);
  LoopWorker.parloop(0, N, worker_1, 0, SCHED_BLOCK);
}
```

Now, suppose that a similar loop that operates on class variables appears in a class method `classMethod()` of a class `SimpleLoop2`:

```
class SimpleLoop2 {
   static double[] a;
   static double[] b;
   static void classMethod(int N) {
L2:  for (int i = 0; i < N; i++)
       a[i] = 1.0D / b[i];
   }
}
```

In this case, a class method `run_2()` similar to the one shown above (but with a qualifier `static`) is added to the class `SimpleLoop2`, and the whole loop is replaced by a construct that is similar to the construct shown above (but without an argument `this` in the constructor invocation of `LoopWorker_2`). The loop-worker for the parallel loop `L2` merely consists of a `run()`-method:

```
class LoopWorker_2 extends LoopWorker {
  public void run() {
    while ( pool.nextWork(this) )
      SimpleLoop2.run_2(low, high, stride, sync);
  }
}
```

In figures 8 and 9, we show the serial execution time T_s and the parallel execution time T_p of the methods `instanceMethod()` and `classMethod()` for varying values of `N`. The execution time of the parallel versions run with the true parallel execution of threads *disabled* is also shown. These experiments indicate that on a uni-processor, the overhead introduced by the loop parallelization method presented in this paper is very small. In figure 10, we show the speedup of the true parallel versions. Here we see that on the four processor IBM RS/6000 G30, parallelization of this particular single loop becomes useful if `N` exceeds $20,000$. An efficiency ranging from 75% up to over 90% is obtained once `N` exceeds $220,000$.

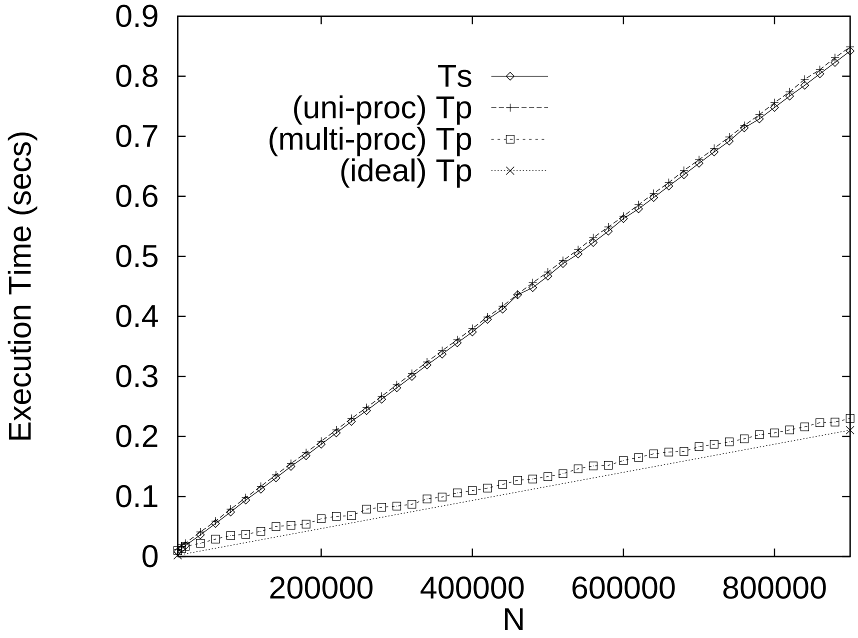

Figure 8: Single Loop in Instance Method

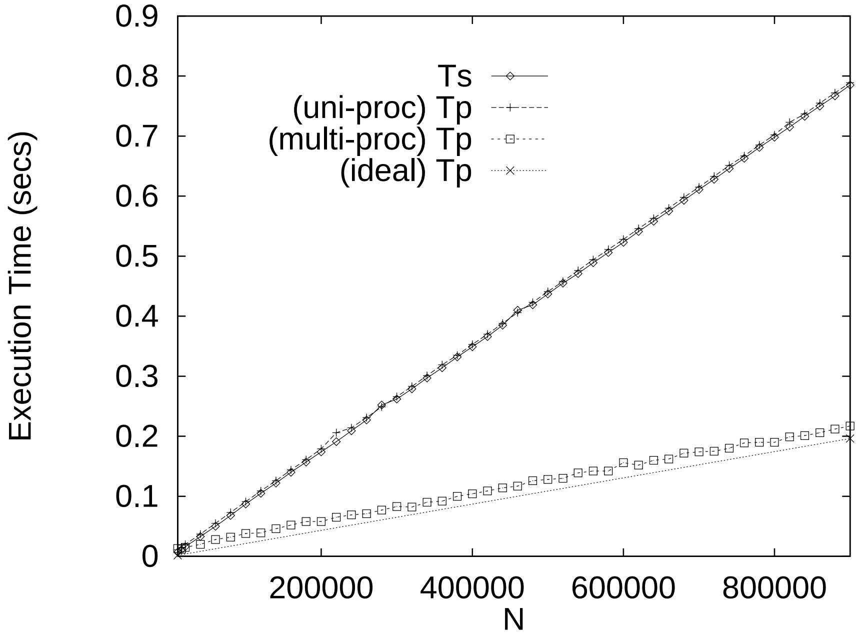

Figure 9: Single Loop in Class Method

5.1.2 Double Loops

The following double loop is based on an applet example found in [27]:

```
class DoubleLoop {
  void computePixels(int N) {
    int[] pixels = new int[N * N];
    for (int y = 0; y < N; y++)
      for (int x = 0; x < N; x++) {
        int r = (x^y)     & 0xff;
        int g = (x*2^y*2) & 0xff;
        int b = (x*4^y*4) & 0xff;
        pixels[y*N + x] = (255 << 24) | (r << 16) | (g << 8) | b;
      }
    ... more operations ...
  }
}
```

Data dependence analysis reveals that neither the y- nor the x-loop carries any data dependence. Parallelization of the outermost loop is more desirable to amortize startup overhead over more iterations. In contrast with the previous examples, however, in this case the loop-body of the y-loop refers to the parameter N of `computePixels()` (used as upper bound in the x-loop) as well as the local array `pixels` that is declared outside this loop. Because the value of N and `pixels` *itself* is not altered in the loop, however, the loop can still be parallelized by supplying the value of these variables to the method that is constructed in the first step (see section 3.3.1):

```
void run_3(int l_3, int h_3, int s_3, RandomSync[] sync_3, int N, int[] pixels) {
  for (int y = l_3; y < h_3; y += s_3)
    for (int x = 0; x < N; x++) {
      int r = (x^y)     & 0xff;
      int g = (x*2^y*2) & 0xff;
      int b = (x*4^y*4) & 0xff;
      pixels[y*N + x] = (255 << 24) | (r << 16) | (g << 8) | b;
    }
}
```

Loop-workers for this loop are described by the following class, having two additional fields to store the value of the integer N and the integer array `pixels` (see section 3.3.2):

```
class LoopWorker_3 extends LoopWorker {
  DoubleLoop target;
  int        N_3;
  int[]      pixels_3;

  LoopWorker_3(DoubleLoop target, int N_3, int[] pixels_3)  {
    this.target   = target;
    this.N_3      = N_3;
    this.pixels_3 = pixels_3;
  }
  public void run() {
    while ( pool.nextWork(this) ) {
      target.run_3(low, high, stride, sync, N_3, pixels_3);
    }
  }
}
```

Finally, the original loop is replaced by the construct shown below, in which the value of N and `pixels` is supplied to every loop-worker (see section 3.3.3):

```
void computePixels(int N) {
  int[] pixels = new int[N * N];
  {  LoopWorker_3[] worker_3 = new LoopWorker_3[4];
     for (int i_3 = 0; i_3 < 4; i_3++)
       worker_3[i_3] = new LoopWorker_3(this, N, pixels);
     LoopWorker.parloop(0, N, worker_3, 0, SCHED_BLOCK);
  }
  ... more operations ...
}
```

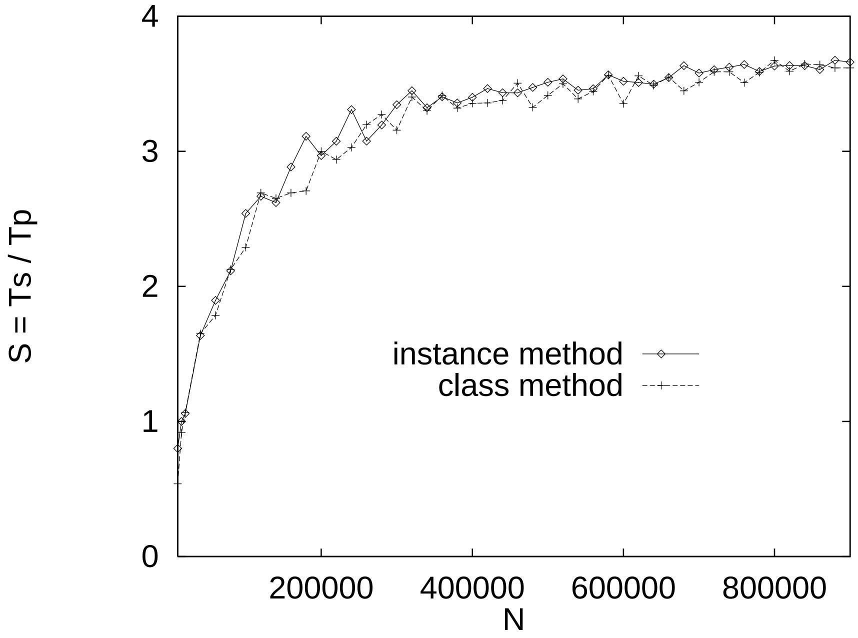

Figure 10: Speedup of Single Loops

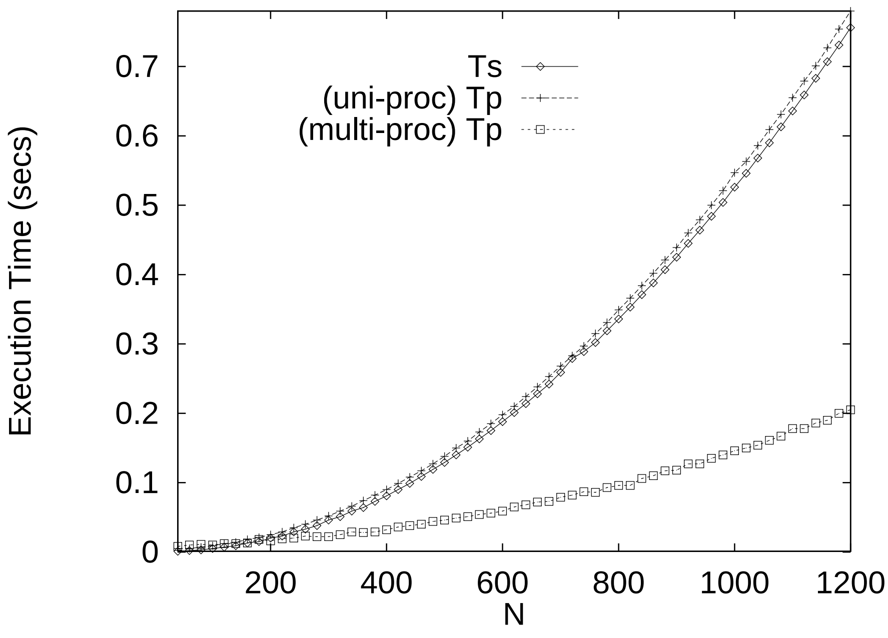

Figure 11: Double Loop

In figure 11, we show the execution time of the serial and parallel version of the previous loop (excluding the time required for explicit memory allocation) for varying values of `N`. For this particular double loop, parallelization becomes useful if `N` exceeds 180, and an efficiency ranging from 75% up to more than 90% is obtained once `N` exceeds 500.

5.2 Matrix Multiplication

To demonstrate the usefulness of the different scheduling policies, we have conducted some experiments with loop parallelization in the following class `Matmat`:

```
 class Matmat {

    final  static int M = 120, N = 120, K = 100;

    static double a[][] = new double[M][N];
    static double b[][] = new double[N][K];
    static double c[][] = new double[M][K];

   public static void main(String args[]) {
     ...
L1:   for (int i = 0; i < M; i++)
         for (int j = 0; j < K; j++)
            for (int k = 0; k < N; k++)
               c[i][j] += a[i][k] * b[k][j];
     ...
   }
 }
```

The most straightforward way to parallelize this implementation on a shared-address-space architecture is to convert the outermost `i`-loop into a parallel loop. In figure 12, we show the execution time of the original serial loop and the parallel loop for a varying number of threads and the three different scheduling policies discussed in this paper. Because work is spread evenly over the iterations, the scheduling policies have similar performance.

Now, suppose that the array `a` is used to store a lower triangular matrix, so that the the whole loop can be expressed as follows:

```
L1:   for (int i = 0; i < M; i++)
         for (int j = 0; j < K; j++)
            for (int k = 0; k < i; k++)
               c[i][j] += a[i][k] * b[k][j];
```

Because in this loop the amount of work is not spread evenly over the iterations, block scheduling suffers from some load imbalancing, as can be seen in figure 13, This load imbalancing, however, can be alleviated by allocating a few additional loop-workers.

Now, suppose that only a few rows of the matrix stored `c` have to be computed. As illustrated below, this can be accomplished by means of a boolean array `filter`:

```
L1:   for (int i = 0; i < M; i++)
        if (filter[i])
          for (int j = 0; j < K; j++)
            for (int k = 0; k < N; k++)
               c[i][j] += a[i][k] * b[k][j];
```

If, for example, only the even rows of `c` have to be computed, a severe load imbalancing may result using cyclic scheduling, as illustrated in figure 14. If, as another example, only the first $M/2$ elements are set, guided self-scheduling suffers from a similar load imbalancing, as can be seen in figure 15. These experiments indicate that, in general, no decisive statement about which scheduling policy is the best can be made.

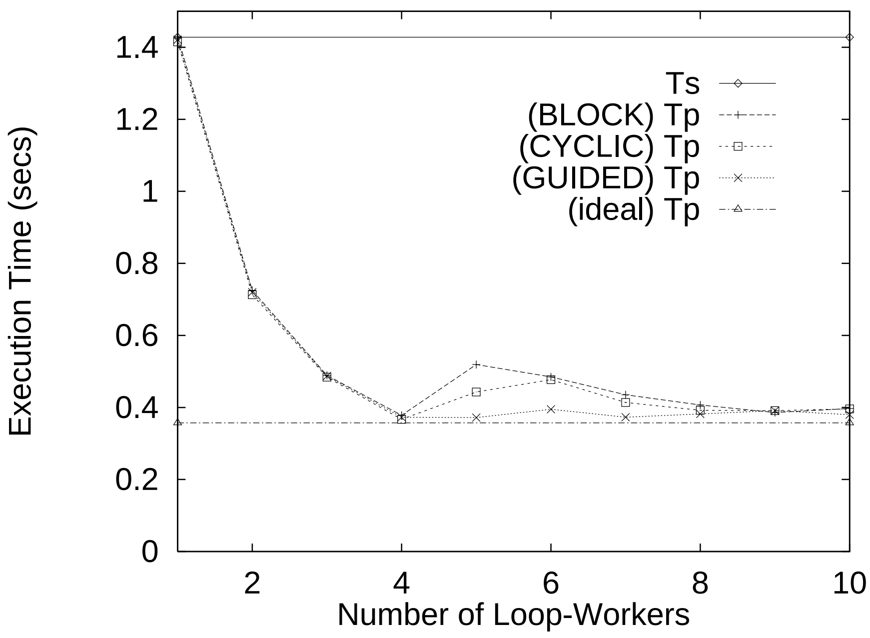

Figure 12: Matrix Multiplication

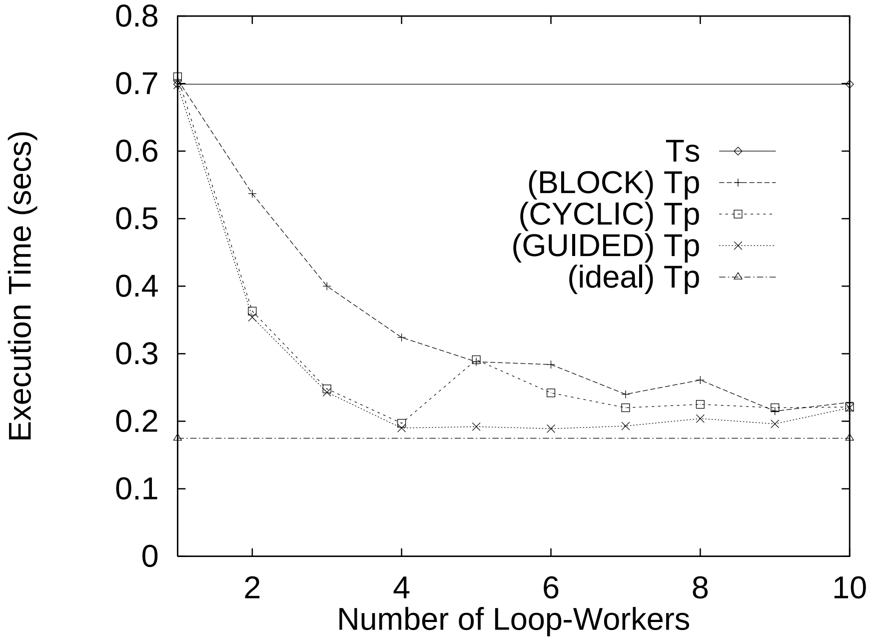

Figure 13: Triangular Matrix Multiplication

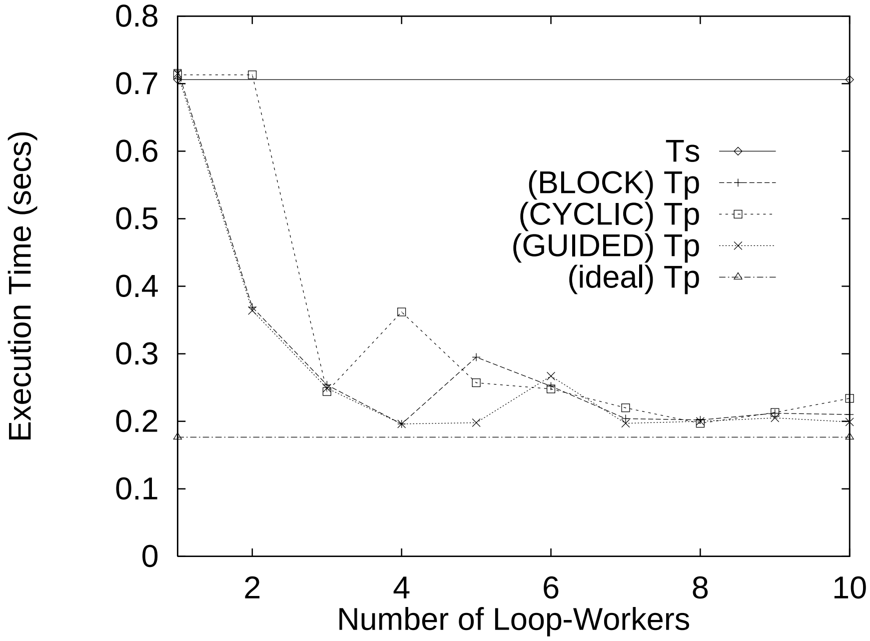

Figure 14: Filtered Matrix Multiplication (even rows)

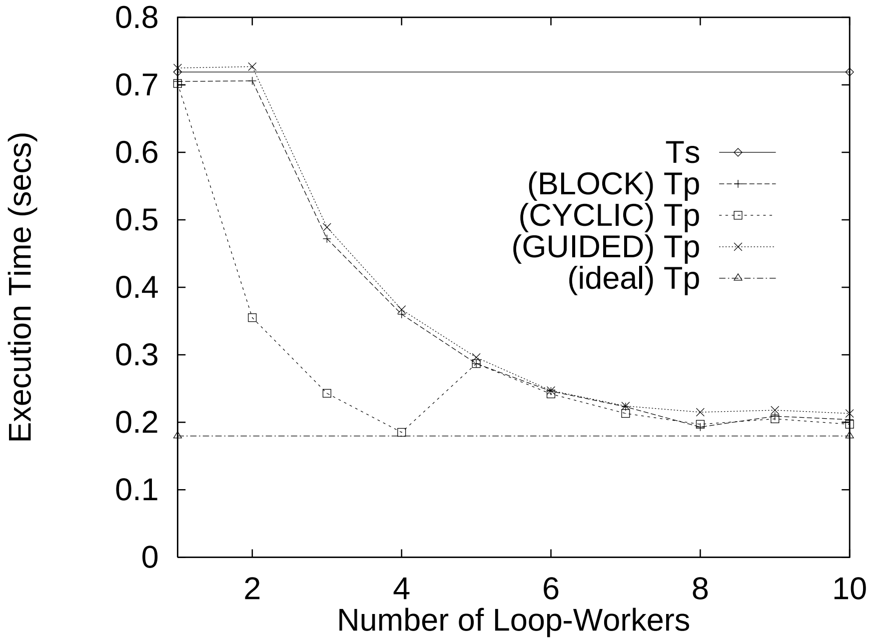

Figure 15: Filtered Matrix Multiplication (first M/2 rows)

5.3 Random Synchronization

Consider the following imperfectly nested loop, in which a static loop-carried flow dependence $S_1\delta_< S_2$ with distance 8 holds (i.e. for $i \geq 8$, statement instance $S_2(i)$ depends on statement instance $S_1(i-8)$):

```
     class SingleDependence {
       static final int N = ...;

       static double[]   a = new double[N];
       static double[][] b = new double[N][N];

       static void compute() {
         for (int i = 0; i < N-8; i++) {
S1:        a[i+8] /= 3.0D;
           for (int j = 0; j < N; j++)
S2:           b[i][j]  = 20.0D * a[i];
         }
       }
       ...
     }
```

Straightforward parallelization of the i-loop is only valid if the loop is converted into a DOACROSS-loop and one synchronization variable is used to enforce all data dependences, i.e. `SVARS = 1`. Parallelization of the i-loop proceeds as explained earlier, where the class method `run_x()` that is added to class `SingleDependence` is shown below:

```
       static void run_x(int l_x, int h_x, int s_x, RandomSync[] sync_x) {
         for (int i = l_x; i < h_x; i += s_x) {
S1:        a[i+8] /= 3.0D;
           sync_x[0].doPost(i);               // post(ASYNC, i)
           sync_x[0].doWait(i-8);             // wait(ASYNC, i-8)
           for (int j = 0; j < N; j++)
S2:          b[i][j] = 20.0D * a[i];
         }
       }
```

However, another way to obtain parallelism in the loop shown above is to first apply loop-distribution [33, 39, 40, 41] to the i-loop, which is valid because all data dependences are lexically forward. Thereafter, the second resulting i-loop can be converted into a DOALL-loop. For small values of N, parallelization of the other i-loop or the j-loop is not likely to be useful:

```
       static void compute() {
         for (int i = 0; i < N-8; i++)
S1:        a[i+8] /= 3.0D;
         for (int i = 0; i < N-8; i++) {
           for (int j = 0; j < N; j++)
S2:          b[i][j]  = 20.0D * a[i];
         }
       }
```

In figure 16, we show the execution time of the DOACROSS- and the DOALL-loop for varying values of N. For the former loop, we present the execution time for block-scheduling and cyclic-scheduling in case the `wait()/notifyAll()`-implementation of random synchronization described in section 3.2.4 is used. Here we see that the overhead of random synchronization is substantial, and that the loop becomes effectively serialized if block-scheduling is used. In this case, it is more efficient to use loop distribution to enable operations on array b to be executed in a DOALL-like manner.

Unfortunately, data dependences cannot always be dealt with so easily. For example, in the example shown above, there is a data dependence cycle, caused by the static flow dependences $S_1\delta_< S_2$ and $S_3\delta_< S_2$ having distance 8 and 9 respectively:

```
  class DependenceCycle {
    static final int N = ...;

    static double[]   a = new double[N];
    static double[][] b = new double[N][N];
    static double[]   c = new double[N];

    static void compute() {
      for (int i = 9; i < N-8; i++) {
S1:     a[i+8] /= 3.0D;
        for (int j = 0; j < N; j++)
S2:       b[i][j]  = (20.0D * c[i-9]) / a[i];
S3:     c[i] /= 8.0D;
      }
   }
    ...
  }
```

In this case, the only way to exploit parallelism in the outermost loop is to execute the `i`-loop in a DOACROSS-like manner using two synchronization variables, i.e. `SVARS = 2`. The `run_y()` method that can be used for this purpose is shown below:

```
  static void run_y(int l_y, int h_y, int s_y, RandomSync[] sync_y) {
    for (int i = l_y; i < h_y; i += s_y) {
S1:   a[i+8] /= 3.0D;
      sync_y[0].doPost(i);                  // post(ASYNC, i)
      sync_y[0].doWait(i-8);                // wait(ASYNC, i-8)
      sync_y[1].doWait(i-9);                // wait(CSYNC, i-9)
      for (int j = 0; j < N; j++)
S2:      b[i][j] = (20.0D * c[i-9]) / a[i];
S3:   c[i] /= 8.0D;
      sync_y[1].doPost(i);                  // post(CSYNC, i)
    }
  }
```

In figure 17, we show the execution time of the serial and parallel version of the previous `i`-loop using a `wait()/notifyAll()`- and a busy-waiting-implementation of random synchronization. Here we see that the latter is clearly superior. Starting two additional threads (viz. $t = 6$) for the former implementation in an attempt to exploit available processors time while threads are waiting fails due to the overhead involved.

5.4 Tree Traversals

In this section, we illustrate the parallelization of multi-way recursive methods in full detail with two very simple tree traversal methods for trees containing integer data items that are implemented as follows (see e.g. [10] for discussion of how some typical data structures can be implemented in Java):

```
class Tree {
  int  val;
  Tree left, right
  ...
}
```

In the examples, we assume that exceptions do not have to be dealt with.

5.4.1 Straightforward Parallelization

The number of levels in a tree, for example, can be computed by passing the root of this tree to the following class method `compLevel1()`:

```
final static int compLevel1(Tree t) {
  if (t == null)
    return 0;
  else {
    int l, r;
    l = compLevel1(t.left);
    r = compLevel1(t.right);
    return (l > r) ? (l+1) : (r+1);
  }
}
```

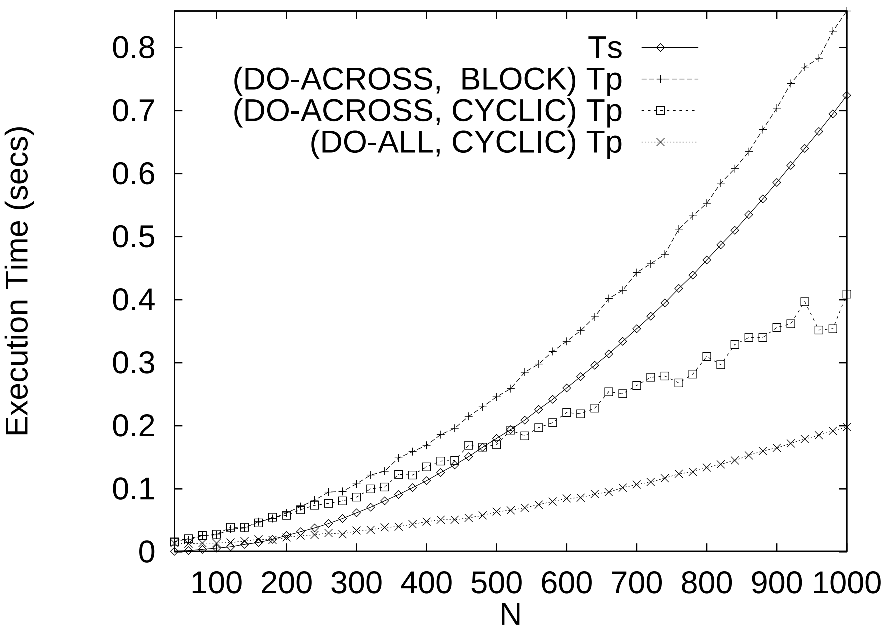

Figure 16: Random Synchronization

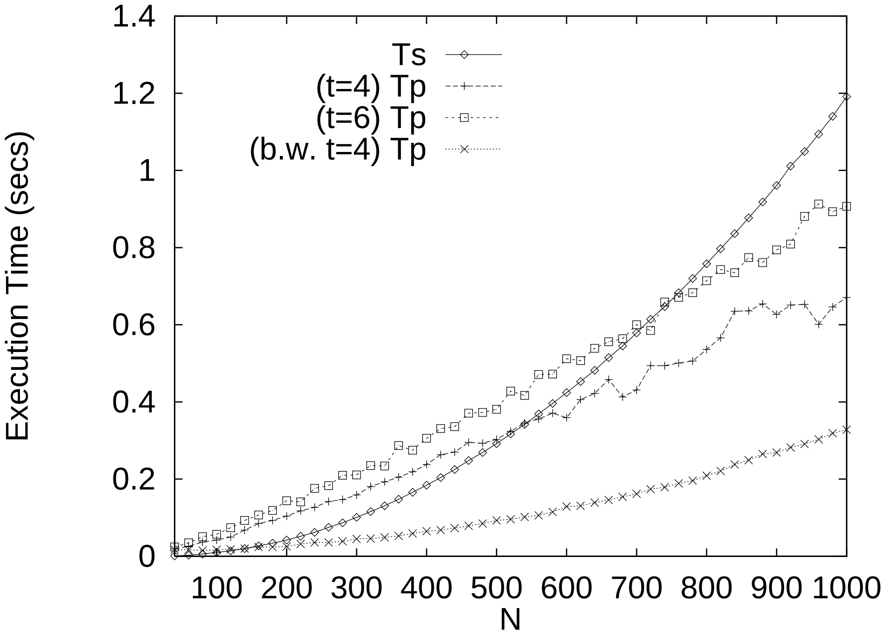

Figure 17: Random Synchronization

Obviously, the number of levels in the two sub-trees rooted at `t` can be computed in parallel and `compLevel1()` has the form (1) given in section 4.1, where `target1` and `target2` are implicitly defined as `Tree`. Hence, the programmer can use annotations to identify `compLevel1()` as a parallel 2-way recursive method.

In the first step, the compiler constructs the following class (see section 4.2.1):

```
class TreeWorker_a extends Thread {
  int  result;
  int  depth;
  Tree t_a;
  TreeWorker_a(int depth, Tree t_a) {
    this.depth = depth;
    this.t_a   = t_a;
  }
  public void run() {
    result = Tree.compLevel1_par_a(depth, t_a);
  }
}
```

Subsequently, the original method `compLevel1()` is rewritten into the method shown below, where a dummy assignment to `l` has been added to preserve the definite assignment property (see section 4.2.2):

```
final static int compLevel1_par_a(int d_a, Tree t) {
  if (t == null)
    return 0;
  else {
    int l = 0, r;
    TreeWorker_a w1_a = null;
    if (d_a <= CUT_DEPTH)
      w1_a = new TreeWorker_a(d_a+1, t.left);
    else
       l = compLevel1_par_a(d_a+1, t.left);
    r = compLevel1_par_a(d_a+1, t.right);
    if (w1_a != null) {
      try { w1_a.join(); }
         catch(InterruptedException e_a) {}
      l = w1_a.result;
    }
    return (l > r) ? (l+1) : (r+1);
  }
}
```

Finally, the following method that invokes the static method `compLevel1_par_a()` is added to the class `Tree` (see section 4.2.3):

```
final static int compLevel1(Tree t) {
  return compLevel1_par_a(0, t);
}
```

In figure 18, we show the serial execution time T_s and the parallel execution time T_p for two cut-depths $c = 1$ (4 threads) and $c = 3$ (16 threads). All versions are applied to *full* binary trees with a varying number of nodes N, letting the number of levels range from 14 to 19. Since only a constant amount of work is done for each node, a reasonable speedup may be expected for $p = 4$, since the serial execution time $T_s = \Theta(N)$ can be changed into $T_p(N) = \Theta(1) + \Theta(1) + T_1(N/4)$. In figure 19, we present the obtained speedup. Because the trees are well-balanced, increasing the number of threads only decreases performance due to the contention between running threads.

Alternatively, the previous computation can be done by calling the following instance method `compLevel2()`, expressed specifically in the form (1), on the root of the tree (since `compLevel2()` is final, we avoid the situation where another method is called in case `left` or `right` contains an instance of a sub-class of `Tree` that overrides `compLevel2()`):

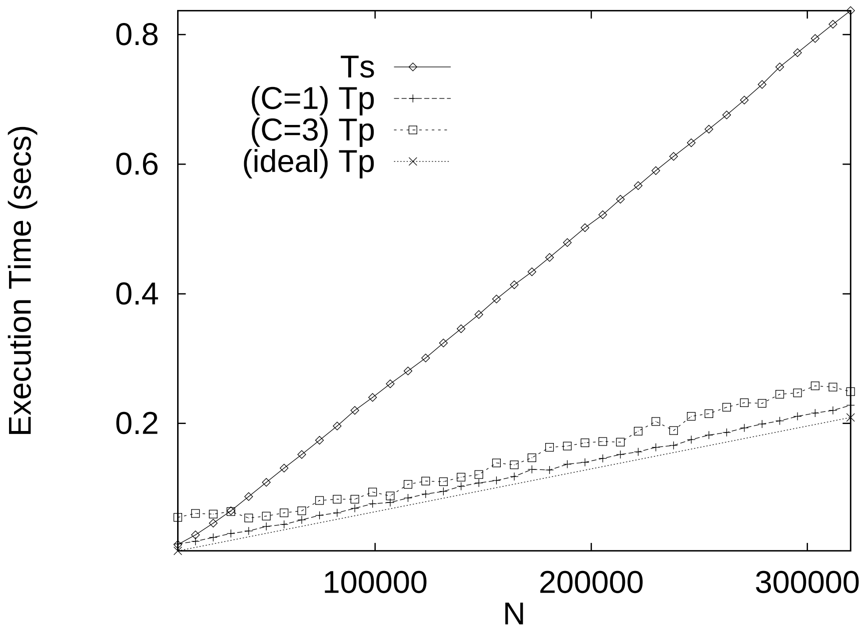

Figure 18: Tree Traversal (class method)

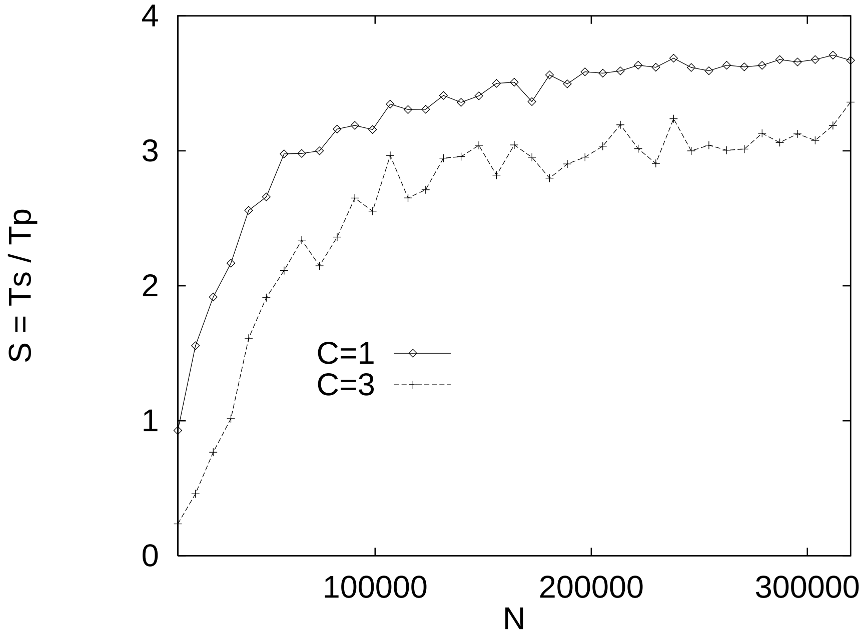

Figure 19: Speedup of Tree Traversal (class method)

```
final int compLevel2() {
  if ( (left == null) || (right == null) )
    return (left != null)
     ? (1 + left.compLevel2())
     : ((right != null) ? (1 + right.compLevel2()) : 1);
  else {
   int l, r;
   l = left.compLevel2();
   r = right.compLevel2();
   return (l > r) ? (l+1) : (r+1);
 }
}
```

Because `compLevel2()` is an instance method, the corresponding class `TreeWorker_b` has an additional `target` field of type `Tree` on which the parallel method is called in the `run()`-method:

```
class TreeWorker_b extends Thread {
  int  result;
  int  depth;
  Tree target;
  TreeWorker_b(int depth, Tree target) {
    this.depth  = depth;
    this.target = target;
  }
  public void run() {
    result = target.compLevel2_par_b(depth);
  }
}
```

The transformations applied to the parallel method `compLevel2_par_b()` are similar to the transformations presented in the previous section. However, now an additional parameter is passed to the constructor of `TreeWorker_b` to record the target on which the method must be called. Moreover, note that the recursive method invocations in the `alt_code` fragment also have been replaced by invocations of `compLevel2_par_b()`:

```
final int compLevel2_par_b(int d_b) {
  if ( (left == null) || (right == null) )
    return (left != null)
      ? (1+left.compLevel2_par_b(d_b+1))
      : ((right != null) ? (1+right.compLevel2_par_b(d_b+1)) : 1);
  else {
    int l = 0, r;
    TreeWorker_b w1_b = null;
    if (d_b <= CUT_DEPTH)
      w1_b = new TreeWorker_b(d_b+1, left);
    else
      l = left.compLevel2_par_b(d_b+1);
    r = right.compLevel2_par_b(d_b+1);
    if (w1_b != null) {
      try { w1_b.join(); }
         catch(InterruptedException e_b) {}
      l = w1_b.result;
    }
    return (l > r) ? (l+1) : (r+1);
  }
}
```

Finally, the following method that implicitly calls `compLevel2_par_b()` on `this`, i.e. the object on which method `compLevel2()` itself was called, is added to the class `Tree`:

```
final int compLevel2(Tree t) {
  return compLevel2_par_b(0);
}
```

In figure 20, we show the results of applying this method to the trees of the previous section with two extra top nodes to introduce some load imbalancing, as illustrated in figure 21. Since for cut-depth $c = 1$, all work is done by only one thread, in this case the parallel execution time is equal to the serial execution time with some slight overhead that is mainly due to passing the additional parameter `d_x`. For cut-depth $c = 3$, however, speedup is obtained again. Moreover, because threads that have completed their work no longer compete for processor time, in this case less overhead due to contention arises (cf. the parallel execution time for $c = 3$ in figures 18 and 20).

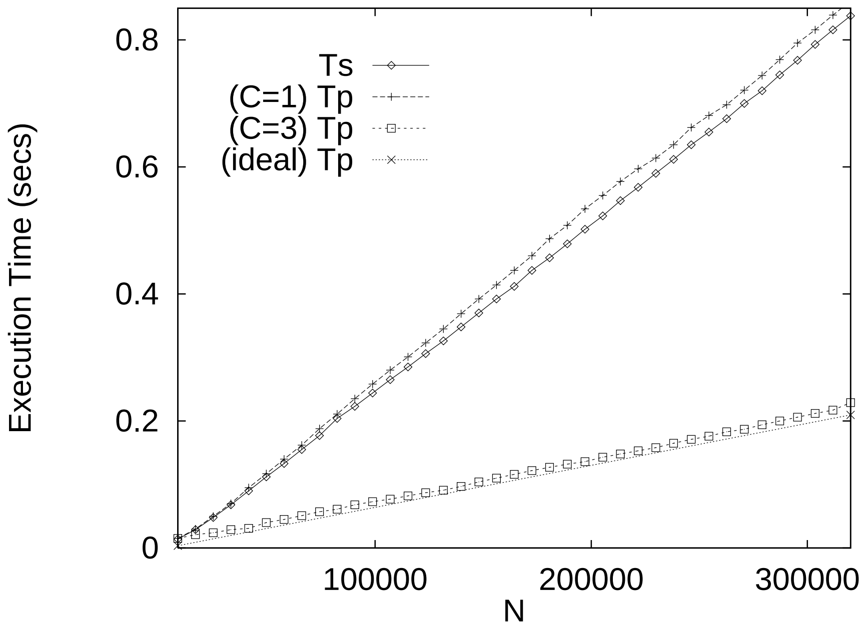

Figure 20: Unbalanced Tree Traversal (instance method)

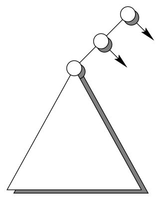

Figure 21: Unbalanced Tree

5.4.2 Overhead Reduction

As alluded to in section 4.2.2, the overhead of passing an additional parameter during serial execution in the bottom levels of the method invocation tree can be eliminated at the expense of some code duplication by using an unaltered copy of the original method at this stage. For example, if we denote this copy by `compLevel2_ser_b()` and all recursive method invocations in this copy are renamed accordingly, then `compLevel2_par_b()` can be expressed, for instance, as follows:

```
final int compLevel2_par_b(int d_b) {
    if (d_b > CUT_DEPTH)
      return compLevel2_ser_b();
    else {
      if ((left == null) || (right == null))
        return (left != null)
         ? (1+left.compLevel2_par_b(d_b+1))
         : ((right != null) ? (1+right.compLevel2_par_b(d_b+1)) : 1);
      else {
        int l = 0, r;
        TreeWorker_b w1_b = new TreeWorker_b(d_b+1, left);
        r = right.compLevel2_par_b(d_b+1);
        try { w1_b.join(); }
           catch(InterruptedException e_b) {}
        l = w1_b.result;
        return (l > r) ? (l+1) : (r+1);
      }
   }
}
```

In figure 22, we present the parallel execution time for cut-depths $c = 1$ and $c = 3$. The execution time for cut-depth $c = 1$ reveals that after this improvement, the overhead of the parallelization method is almost negligible. As expected, speedup is obtained again for cut-depth $c = 3$. In the next section we will see that starting some additional threads can also be useful to overcome less trivial load imbalancing.

5.5 Quick Sorting

As an example of a typical divide-and-conquer algorithm, consider the following implementation of quick-sorting [14] in which, as advocated in [37], small sub-arrays are sorted by insertion sorting to prevent further recursive method invocations for small sub-arrays:

```
public class Sort {
  ...
  final static void quicksort(int[] a, int q, int r) {
    if ((r - q) <= 20) {
      // Insertion Sorting of a[q..r]
      int i, j;
      for (i = q+1; i <= r; i++) {
         int v = a[i];
         for (j = i; q < j; j--)
           if (a[j-1] > v)
             a[j] = a[j-1];
           else
             break;
         a[j] = v;
      }
    }
    else {
      // Quick Sorting
      int t, s = q;
      t = a[q]; a[q] = a[(q+r)/2]; a[(q+r)/2] = t;
      for (int i = q+1; i <= r; i++)
        if (a[i] <= a[q]) {
          t = a[++s]; a[s] = a[i]; a[i] = t;
        }
      t = a[q]; a[q] = a[s]; a[s] = t;
      quicksort(a, q,   s-1);
      quicksort(a, s+1, r  );
    }
  }
}
```

After the programmer has indicated that the recursive method invocations can be done in parallel, parallelization of this method proceeds as explained in the previous sections. In figure 23, we show the serial execution time T_s and parallel execution time T_p with cut-depths $c = 1$ and $c = 3$ for integer arrays of varying length N. In each array of length N, value $N - i + 1$ is assigned to every element i, so that with the pivoting method shown above, the method invocation trees are well-balanced. In figure 24, we show the corresponding speedup.

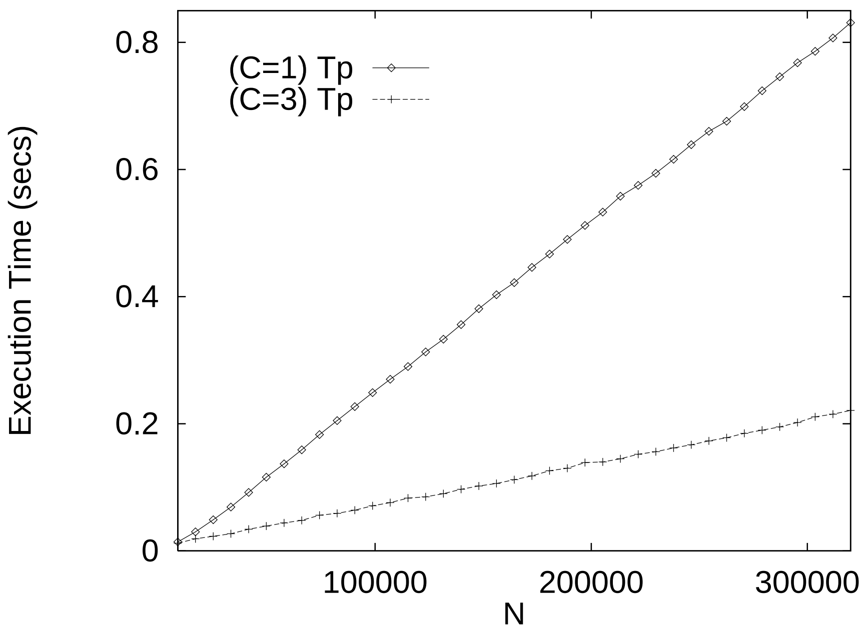

Figure 22: Improved Unbalanced Tree Traversal (instance method)

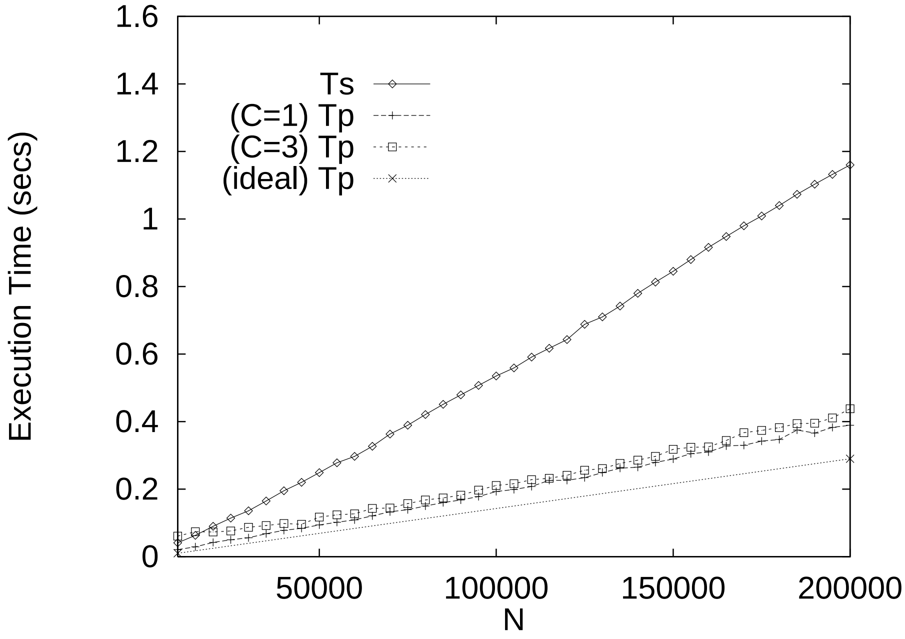

Figure 23: Quick/Insertion Sorting of Reversed Integer Arrays

The reason that even for well-balanced method invocation trees the speedup is never optimal becomes immediately apparent from the discussion in section 4.1. The initial linear terms in $T_p(N) = \Theta(N) + \Theta(N/2) + T_1(N/4)$ contribute substantially to the best possible parallel execution time for $p = 4$. For $N = 200,000$, for instance, the time required to partition an array of size N and $N/2$ takes 0.08 sec. and 0.04 sec., respectively, which is about the distance between the measured parallel execution time and the ideal parallel execution time (i.e. simply the serial execution time divided by four). In figure 25, we present the execution time for random integer arrays (the same pseudo-random sequence of a particular length was used in the serial and parallel experiments). Here we see that some of the load imbalancing due to imbalanced invocation trees can be resolved by starting additional threads.

5.6 Radix-Exchange Sorting

An alternative divide-and-conquer sorting algorithm that can better adapt to integers with truly random bits is so-called radix-exchange sorting (see e.g. [37]). An implementation that handles small sub-arrays differently to improve performance is shown below, where we assume that $\texttt{BIT}[i] = 2^i$.

```
class Sort {
 ...
  final static void radixsort(int a[], int l, int r, int b) {
    if (b >= 0) {
      if ((r-l) <= 20) {
        // Insertion Sorting of a[l..r]
        ...
      }
      else {
        int i = l, j = r, t;

        do {
          while (((a[i] & BIT[b]) == 0) && (i < j)) i++;
          while (((a[j] & BIT[b]) != 0) && (i < j)) j--;
          t = a[i]; a[i] = a[j]; a[j] = t;
        } while (j != i);

        if ((a[r] & BIT[b]) == 0) j++;
        radixsort(a,l,j-1,b-1);
        radixsort(a,j,r,  b-1);
      }
    }
  }
}
```

The static initializer show below can be used to initialize array `BIT`:

```
static int [] BIT = new int[32];
static {
  int k = 1;
  for (int i = 0; i < 32; i++) {
     BIT[i] = k;
     k *= 2;
  }
}
```

An array of positive 32-bit integers (with a zero most significant bit), for example, can be sorted by calling `radixsort()` with `b = 30`. Because the two recursive method invocations can be done in parallel, `radixsort()` is a parallel 2-way recursive method, and implicit parallelism can be made explicit using the method discussed in this paper.

In figure 26, we show the serial and parallel execution time of radix-exchange/insertion sorting when applied to the *absolute* values of the same random integer arrays used for quick/insertion sorting. Although serial radix-exchange/insertion sorting is more expensive than serial quick/insertion sorting, the parallel versions perform better for these random integer arrays, because the method invocation trees are more balanced.

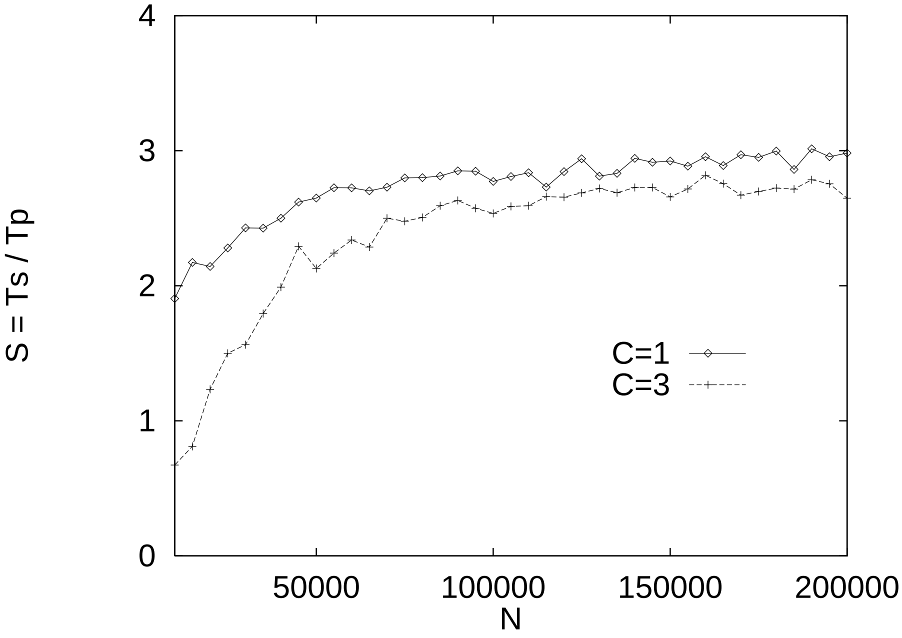

Figure 24: Speedup of Quick/Insertion Sorting of Reversed Integer Arrays

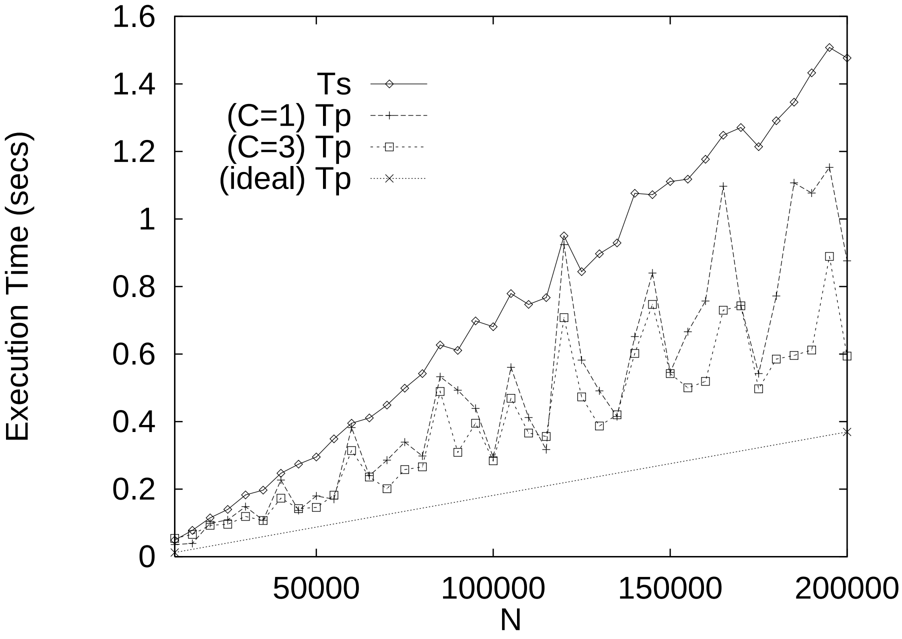

Figure 25: Quick/Insertion Sorting of Random Integer Arrays

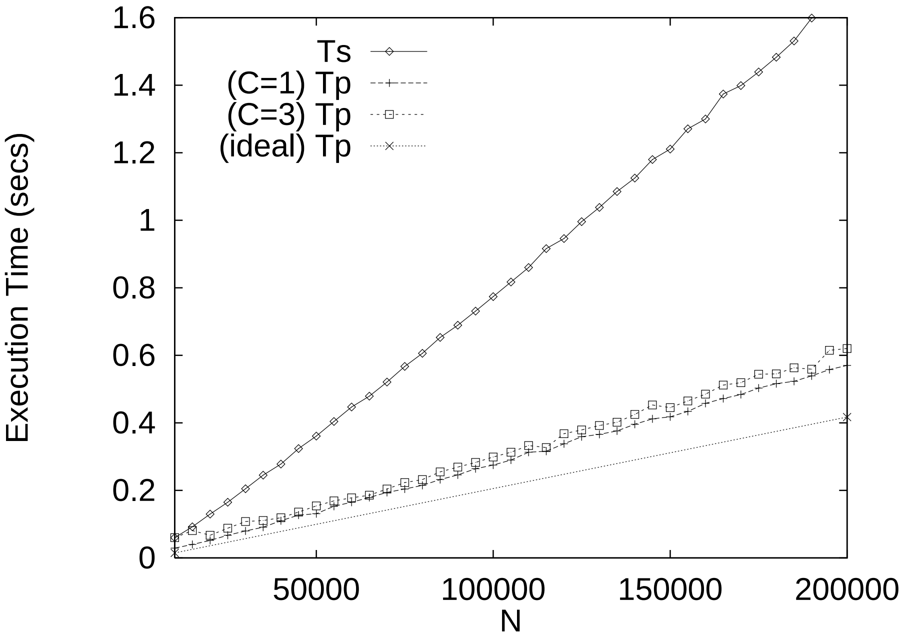

Figure 26: Radix-Exchange/Insertion Sorting of Random Integer Arrays

5.7 Merge Sorting

In this section, we discuss the results of some experiments that have been conducted with array version and linked-lists version of merge-sorting.

5.7.1 Merge Sorting of Arrays

An implementation of merge-sorting for arrays that uses insertion sorting for small sub-arrays is shown below (see e.g. [37]). Here, we assume that a sufficiently large temporary integer array `tmp` is is available to support the merge step:

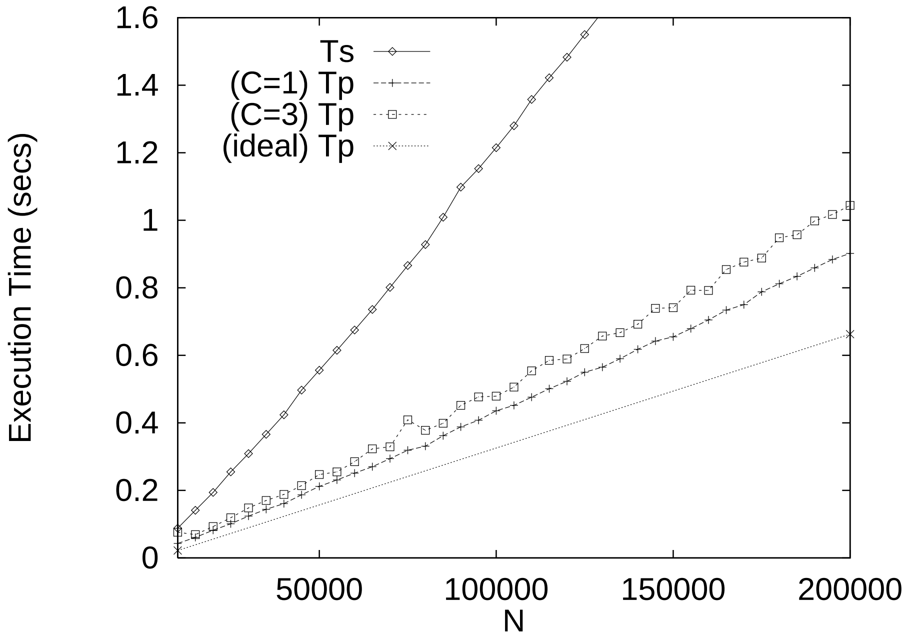

Figure 27: Merge/Insertion Sorting of Random Integer Arrays

```
class Sort {
  ...
  static int[] tmp = new int[SIZE];

  final static void arrayMergesort(int[] a, int l, int r) {
    if ((r - l) <= 20) {
     // Insertion Sorting of a[l..r]
     ...
    }
    else {
      int m = (r+l) / 2;

      arrayMergesort(a, l,m);
      arrayMergesort(a, m+1,r);

     for (int i = m; i >= l; i--)
       tmp[i] = a[i];
     for (int j = m+1; j <= r; j++)
       tmp[r+m+1-j] = a[j];
     int i = l, j = r;
     for (int k = l; k <= r; k++)
       if (tmp[i] < tmp[j])
         a[k] = tmp[i++];
       else
         a[k] = tmp[j--];
     }
  }
}
```

Again, because the recursive method invocations can be done in parallel, the method of this paper can be used to exploit implicit parallelism in `arrayMergesort()`. In figure 27, we show execution time of the serial and parallel version applied to the same random integer arrays used earlier. Although, in contrast with the previous sorting method, the method invocation trees are always well-balanced (independent of the actual values of the elements), the overhead of data movement has a clear impact on the performance of this sorting method.

5.7.2 Merge Sorting of Linked Lists

Consider a linked-list of integers that is implemented using the following class `List`:

```
class List {
  int  val;
  List next;

  List(int val, next) {
    this.val  = val;
    this.next = next;
  }
  ...
}
```

Under the assumption that each list is terminated with a special node `SENTINEL` that points to itself and with a data item larger than all elements in the list, merge-sorting can be implemented as follows (see e.g. [37]):

```
final static List listMergesort(List l) {
  if (l.next == SENTINEL)
    return l;
  else {
    List a = l, b = l.next.next.next;
    while (b != SENTINEL) {
      l = l.next;
      b = b.next.next;
    }
    b      = l.next;
    l.next = SENTINEL;

   a = listMergesort(a);
   b = listMergesort(b);

   return merge(a, b);
  }
}
```

Obviously, `listMergesort()` is a parallel 2-way recursive class method and can be parallelized using the method of this paper. In this case, however, the performance can depend substantially on the implementation of the method `merge()`. This method must merge two sorted linked lists into one completely sorted linked list, as illustrated in figure 28. Consider, for example, the following simple implementation of `merge()`, in which an auxiliary node `l` is explicitly allocated to obtain a hook to the new list:

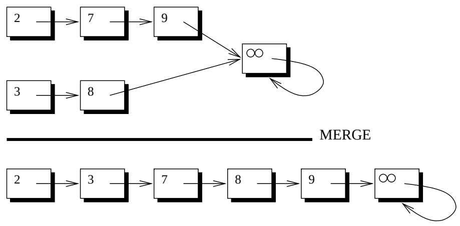

Figure 28: Merging two Linked Lists

```
static List merge(List l1, List l2) {
  List l = new List(0,null), prev = l;
  while (prev != SENTINEL)
    if (l1.key <= l2.key) {
      prev = prev.next = l1;
      l1 = l1.next;
     }
    else {
      prev = prev.next = l2;
      l2 = l2.next;
    }
  return l.next;
}
```

In figure 29, we show the execution time of serial and parallel merge-sorting on integer lists varying in length up to $N = 100,000$. Rather surprisingly, parallelizing the previous implementation of merge-sorting dramatically decreases the performance. This performance decrease is due to the explicit memory allocation in `merge()`. If we implement `merge()` without the need for an auxiliary node, an example of which is shown below, then the execution time shown in figure 30 result:

```
static List merge(List l1, List l2) {
  List l;
  if (l1.key <= l2.key) {
    l = l1;
    l1 = l1.next;
  }
  else {
    l = l2;
    l2 = l2.next;
  }
  List prev = l;
  while (prev != SENTINEL)
  if (l1.key <= l2.key) {
    prev = prev.next = l1;
    l1 = l1.next;
  }
  else {
    prev = prev.next = l2;
    l2 = l2.next;
  }
  return l;
}
```

This experiment clearly reveals that the parallelization method of this paper should only be applied to parallel multi-way recursive methods in which no explicit memory allocation is performed.

5.8 State Space Searching

Finally, we discuss some possible future extensions of our method by means of the following simple implementation of a min-max search algorithm for the game tic-tac-toe. In this implementation, we assume that the method `evalBoard()` yields one of the values -1, 0, or $+1$ if 'O' wins, the game is a draw, or 'X' wins, respectively, or the value `UNDECIDED` otherwise:

```
class TicTacToe {
   ...
  final static int minMax(int[][] board, boolean b) {

    int e = evalBoard(board);  // -1/0/+1 or UNDECIDED

    if (e == UNDECIDED) {

      e = (b) ? -1 : +1;

      for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
          if (board[i][j] == EMPTY) {

            board[i][j] = (b) ? X_SYMBOL : O_SYMBOL;   // PLAY
            int e1 = minMax(board, ! b);
            board[i][j] = EMPTY;                      // UNPLAY

            if ( (b) && (e1 >= e) )
              e = e1;
            else if ((! b) && (e1 <= e) )
              e = e1;
          }
    }

    return  e;
  }
}
```

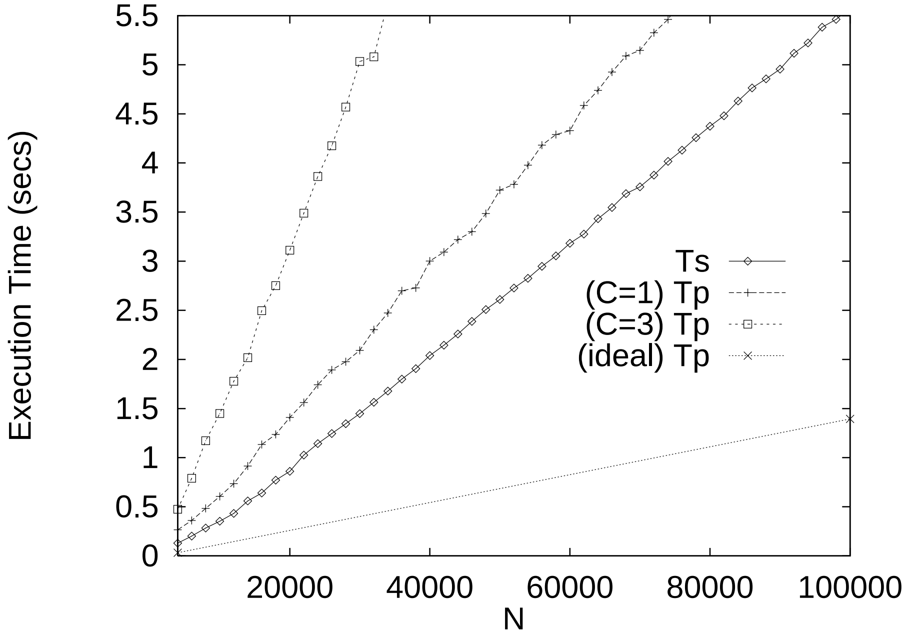

Figure 29: Linked List Merge Sorting (memory allocation)

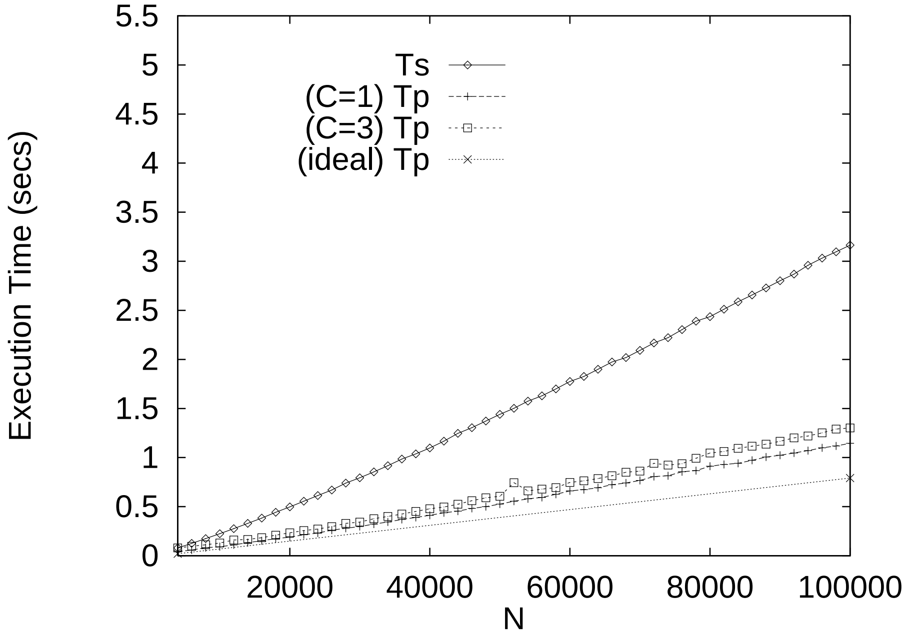

Figure 30: Linked List Merge Sorting (no memory allocation)

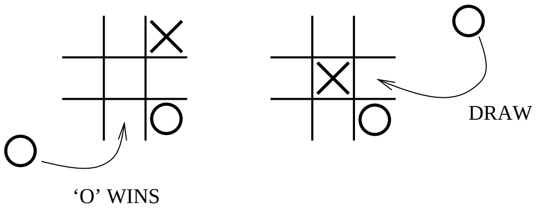

Figure 31: Tic-Tac-Toe

The method minMax() can be called with an arbitrary board position and a boolean value that indicates whose turn it is (if b holds, then player 'X' must place the next symbol). Clearly, this method can be easily modified to yield the best move as side-effect of the evaluation. For example, given the board positions shown in figure 31, this algorithm evaluates to the value -1 and 0, respectively, yielding the moves shown in the same figure.

Although searching different part of the state space can clearly be performed in parallel, for several reasons our method is not directly applicable. First, the different recursive method invocations are controlled by a for-loop, rather than having some fixed recursive method invocations that appear statically in the program text. Second, although the different method invocations could operate on different copies of the board, for efficiency purposes, the state space searching is implemented by doing and undoing moves on a single board. Finally, to avoid the need for additional storage, the best evaluation seen so far is updated immediately after each recursive method invocations, rather than computing the best evaluation after all recursive method invocations are done.

Fortunately, these problems can be easily dealt with. The first problem can be solved using an array of appropriate tree-workers, rather than a fixed number of scalar tree-worker. In such cases, it is more convenient to start a thread for each method invocations that can be done in parallel, rather than treating the last invocation differently. Furthermore, the second problem can be resolved by implementing by-value parameter passing for board. However, because in the previous section we have seen that explicit memory allocation may have a dramatic impact on the performance, this copying is only done in the top levels of the method invocation tree. The last problem is solved by simply distributing the for-loops around the recursive method invocations and the computation of the best evaluation, which is possibly because the intermediate result of each method invocation is stored in field result_x of the corresponding thread.

The class TreeWorker_x for the parallel version has the following form:

```
class TreeWorker_x extends Thread {
  int     result;
  int     depth;
  int[][] board;
  boolean b;
  TreeWorker_x(int d_x, int[][] board, boolean b) {
    this.d_x   = d_x;
    this.board = board;
    this.b     = b;
    start();
  }
  public void run() {
    result = TicTacToe.minMax_par_x(depth, board, b, set);
  }
}
```

The previous observations give rise to the following method minMax_par_x(), where minMax_ser_x() is an unaltered copy of the original method, and copyBoard() is an auxiliary method that yields a new copy of a board:

```
final static int minMax_par_x(int d_x, int[][] board, boolean b) {
  if (d_x > CUT_DEPTH)
    return minMax_ser_x(board, b);
  else {
    int e = evalBoard(board);
    if (e == UNDECIDED) {
      e = (! b) ? +1 : -1;

      TreeWorker_x[][] worker_x = new TreeWorker_x[3][3];
      for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
          if (board[i][j] == EMPTY) {
            int[][] newboard = copyBoard(board);
            newboard[i][j] = (b) ? X_SYMBOL : O_SYMBOL;   // MOVE
            worker_x[i][j] = new TreeWorker_x(d_x + 1, newboard, ! b);
      }
     else
      worker_x[i][j] = null;

     for (int i = 0; i < 3; i++)
       for (int j = 0; j < 3; j++)
         if (worker_x[i][j] != null) {
           try { worker_x[i][j].join(); }
            catch (Exception ex) {}
           int e1 = worker_x[i][j].result;
           if ((b) && (e1 >= e))
             e = e1;
           else if ((! b) && (e1 <= e) )
             e = e1;
       }
     }
   return  e;
  }
}
```

For an empty board, for example, the serial and parallel version using cut-depth $c = 0$ evaluate to a draw in about 3.32 and 0.93 seconds respectively, yielding a speedup of about 3.6.

6 Conclusions

In this paper, we have presented a number of transformations that can be used by a restructuring compiler to exploit some forms of implicit parallelism in Java programs. In particular, we have shown how implicit parallelism in loops and multi-way recursive methods can be made explicit by means of the multi-threading mechanism of the language. Automatically exploiting implicit parallelism simplifies the task of the programmer and makes the parallelization less error-prone. Moreover, because parallelism is expressed in Java itself, the transformed program remains portable. Speedup can be obtained on any platform that supports the true parallel execution of threads, whereas only a slight overhead is induced on uni-processors.

A series of experiments have been conducted on an IBM RS/6000 G30 to show the potential of this approach. We have illustrated the loop parallelization method in detail with some simple loop examples, and we have shown that speedups with an efficiency over 90% can be obtained. Different scheduling policies and a `wait()/notifyAll()`- and busy-waiting-implementation of random synchronization were explored. Moreover, the parallelization of multi-way recursive methods has been illustrated in detail. Good speedup can be obtained for problems in which the method invocation trees are well-balanced, provided that no explicit memory allocation is performed. For methods in which each invocation requires time proportional to the remain input size (e.g. quick-sorting), speedup is limited by the initial linear terms that appear in the best parallel execution time that can be obtained using the simple parallelization method of this paper. Some load imbalancing that is inherent to using a static allocation scheme can be alleviated by increasing the cut-depth to start a few additional threads.

A prototype Java restructuring compiler `javar` that implements the transformations of this paper (and a prototype bytecode parallelization tool `javab`) are made available at the HP-Java page at the Indiana University: `http://www.extreme.indiana.edu/hpjava/`

Acknowledgments

The authors would like to thank Ronald van Loon and Juan E. Villacis for carefully proofreading this paper.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[3] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer, Boston, 1988.

[4] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer, Boston, 1993. A Book Series on Loop Transformations for Restructuring Compilers.

[5] Utpal Banerjee. *Loop Parallelization*. Kluwer, Boston, 1994. A Book Series on Loop Transformations for Restructuring Compilers.

[6] Aart J.C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, 1997.

[7] David Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Department of Computer Science, Rice University, 1987.

[8] Ron G. Cytron. DOACROSS, beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–844, 1986.

[9] Ron G. Cytron. Limited processor scheduling of DOACROSS loops. In *Proceedings of the International Conference on Parallel Processing*, pages 226–234, 1987.

[10] H.M. Deitel and P.J. Deitel. *Java, How to Program*. Prentice-Hall, 1997.

[11] C.N. Fischer and R.J. LeBlanc. *Crafting a Compiler*. Benjamin-Cummings, Menlo Park, California, 1988.

[12] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Sebastopol, CA, 1996.

[13] James Gosling, Bill Joy, and Guy Steele. *Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[14] C.A.R. Hoare. Quick sort. *Computer Journal*, 1962.

[15] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[16] David J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, New York, 1978. Volume 1.

[17] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Programming*. The Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[18] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts, 1997.

[19] Ted G. Lewis. *Foundations of Parallel Programming*. IEEE Computer Society Press, Washington, 1994.

[20] Zhiyuan Li and Walid Abu-Sufah. On reducing data synchronization in multiprocessed loops. *IEEE Transactions on Computers*, C-36:105–109, 1987.

[21] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[22] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly & Associates, Sebastopol, CA, 1997.

[23] Samuel P. Midkiff. *The Dependence Analysis and Synchronization of Parallel Programs*. PhD thesis, C.S.R.D., 1993.

[24] Samuel P. Midkiff and David A. Padua. Compiler generated synchronization for DO loops. In *Proceedings of the International Conference on Parallel Processing*, pages 544–551, 1986.

[25] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36:1485–1495, 1987.

[26] Michael Morrison. *Java Unleashed*. Samsnet, Indianapolis, Indiana, 1996.

[27] Patrick Naughton. *The Java Handbook*. McGraw-Hill, New York, 1996.

[28] Patrick Niemeyer and Joshua Peck. *Exploring Java*. O'Reilly & Associates, Sebastopol, CA, 1996.

[29] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associates, Sebastopol, CA, 1997.

[30] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29:763–776, 1980.

[31] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, 1986.

[32] Thomas W. Parsons. *Introduction to Compiler Construction*. Computer Science Press, New York, 1992.

[33] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer, Boston, 1988.

[34] Constantine D. Polychronopoulos, David J. Kuck, and David A. Padua. Execution of parallel loops on parallel processor systems. In *Proceedings of the International Conference on Parallel Processing*, pages 519–527, 1986.

[35] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York, 1987.

[36] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, New York, 1994.

[37] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, 1988. Second Edition.

[38] Glenn L. Vanderburg et al. *Tricks of the Java Programming Gurus*. Samsnet, Indianapolis, Indiana, 1996.

[39] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London, 1989.

[40] Michael J. Wolfe. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, California, 1996.

[41] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.