Automatically Exploiting Implicit Parallelism in Multi-way Recursive Methods in Java*

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

Abstract

In this paper we show how implicit parallelism in multi-way recursive methods, typically used to implement tree traversal or divide-and-conquer algorithms, can be made explicit by a restructuring compiler using the multi-threading mechanism of Java. Expressing parallelism in Java itself clearly has the advantage that the transformed program remains portable. After compilation of the transformed Java program into byte-code, speedup can be obtained on any platform on which the Java byte-code interpreter support the true parallel execution of threads, whereas only a slight overhead is induced on uni-processors.

1 Introduction

To obtain true portability, a Java program is compiled into the architectural neutral instructions (byte-code) of an abstract machine (the Java Virtual Machine), rather than into native machine code. In this manner, a compiled Java program can run on any platform on which a Java byte-code interpreter is available. Although the interpretation of byte-code 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 byte-code 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 previous work [2] we have shown how a restructuring compiler can exploit implicit loop parallelism in Java programs. In this paper we discuss a method that can be used by a restructuring compiler to make implicit parallelism in multi-way recursive methods explicit by means of multi-threading (see e.g. [5, 1, 3, 4, 8, 10, 11, 12, 16] for a detailed presentation of multi-threading in Java). Since threads are lightweight processes that share an address-space, true parallel execution of threads is usually only supported on shared-address space architectures [7]. Hence, the focus of this paper is on obtaining speedup on such architectures.

The rest of this paper is organized as follows. In section 2, 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. Subsequently, in section 3, we present the results of a series of experiments. Finally, in section 4, conclusions are stated.

*This project is supported by DARPA under contract ARPA F19628-94-C-0057 through a subcontract from Syracuse University.

2 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.

2.1 Parallel Multi-way Recursive Methods

We refer to a method 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                                                          (1)
       r1 = target1.myMethod(a11, ..., a1k)
       ...
       rn = targetn.myMethod(an1, ..., ank)
       post_code
     }
  }
}
```

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.

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 [7, 9, 13, 14]. 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 1.

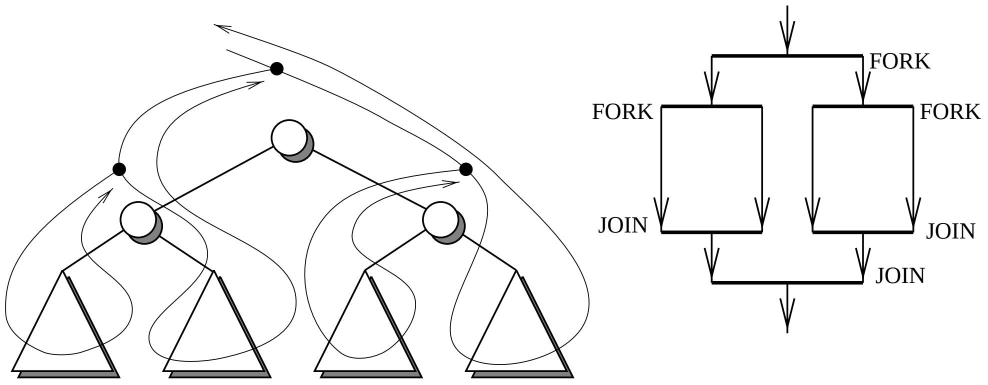

Figure 1: Static Allocation

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 im-balancing can be alleviated by starting additional threads.

2.2 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 have a suffix of the form '_x', although in reality the compiler is responsible for avoiding conflicts with other identifiers,

2.2.1 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 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 2.

Instance variable `d_x` will be used to record the current depth in the method invocation tree. Moreover, for each formal argument of `myMethod()`, there is an instance variable `fi` of the appropriate type.

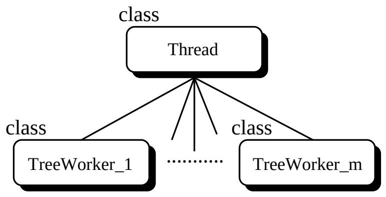

Figure 2: Class Hierarchy

An instance variable `target_x` is possibly used to store a target, while an instance variable `result_x` can be used to transfer the result of a method invocation. 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 {
  int     d_x;
  MyClass target_x;
  type1   f1;
  ...
  typek   fk;
  type    result_x;
  TreeWorker_x(int d_x, MyClass target_x, type1 f1, ..., typem fk) {
    this.d_x      = d_x;
    this.target_x = target_x;
    this.f1       = f1;
    ...
    this.fk       = fk;
    start();
  }
  public void run() {
    result_x = target_x.myMethod_par_x(d_x, f1, ..., fk);
  }
}
```

Constructs involving `target_x` 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_x` are omitted.

2.2.2 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) {
  ...
}
```

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_x()` 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(Expection e) {}
  ri = wi_x.result_x;
}
```

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 [5], 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, we may reduce overhead by executing an *unaltered* copy of the original method, as illustrated in section 3.1.2). 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 im-balancing problems (for some large problems, performance can be improved by keeping the number of threads that are *actually running* equal to p, as discussed in section 3.2). Threads eventually join with their originating threads, until the single thread that invoked the parallel recursive method remains. 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.

2.2.3 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:

```
[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 *indirect 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.

2.2.4 Exception Handling

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 2.2.2) 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, because invocations of a parallel multi-way recursive method are executed is 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.

3 Experiments

In this section, we discuss the parallelization of tree traversal and some typical divide-and-conquer algorithms. Experiments have been conducted on an IBM RISC System/6000 G30 with four 604 processors using the IBM V1.0.2.B Java programming environment. All programs are compiled into byte-code 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.

3.1 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. [3] 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.

3.1.1 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()`:

```
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);
  }
}
```

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 2.1, 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 2.2.1):

```
class TreeWorker_a extends Thread {
  private int  d_a;
  private Tree t;
  int     result_a;
  TreeWorker_a(int d_a, Tree t) {
    this.d_a = d_a;
    this.t   = t;
  }
  public void run() {
    result_a = Tree.compLevel1_par_a(d_a, t);
  }
}
```

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 2.2.2):

```
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);
    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(Expection e) {}
      l = w1_a.result_a;
    }
    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 2.2.3):

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

In figure 5, we show the serial execution time T_s and the parallel execution time T_p for cut-depths $c = 1$ (4 threads) and $c = 3$ (16 threads). All versions are applied to *full* binary trees with a varying number N of nodes, where the number of levels ranges 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 4, 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:

```
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_b` field of type `Tree` on which the parallel method is called in the `run()`-method:

```
class TreeWorker_b extends Thread {
  private int  d_b;
  private Tree target_b;
  private Tree t;
  int     result_b;
  TreeWorker_b(int d_b, Tree target_b, Tree t) {
    this.d_b      = d_b;
    this.target_b = target_b;
    this.t        = t;
  }
  public void run() {
    result_b = target_b.compLevel2_par_b(d_b, t);
  }
}
```

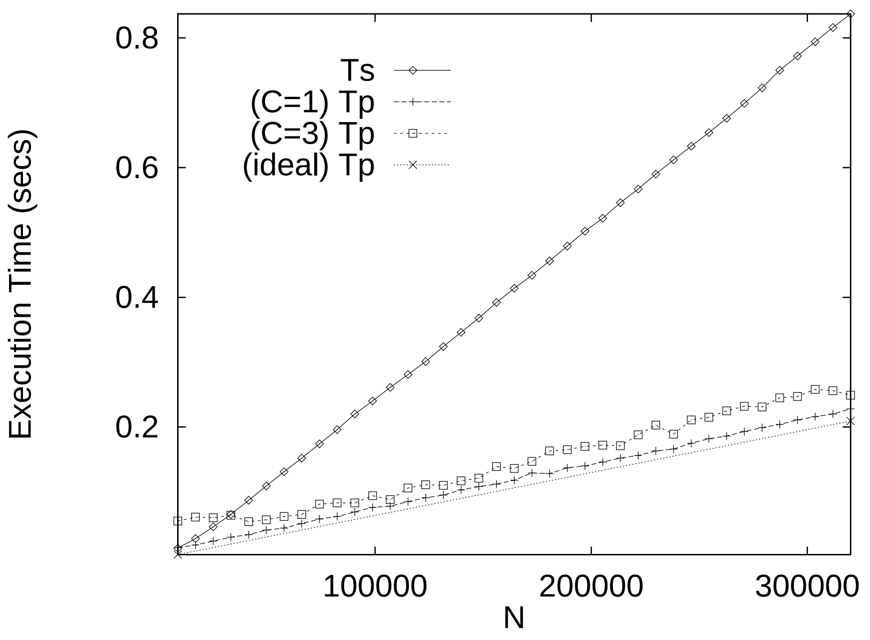

Figure 3: Tree Traversal (class method)

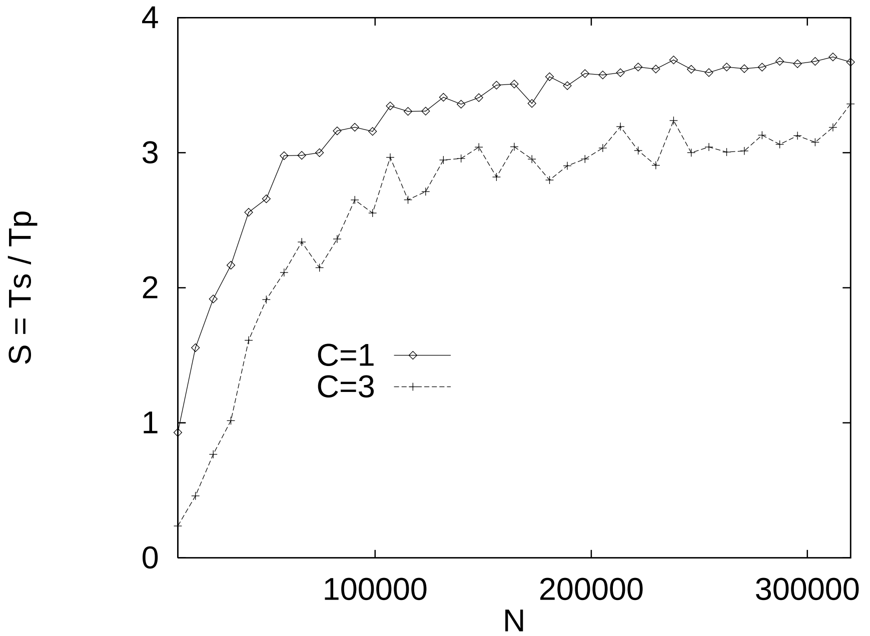

Figure 4: Speedup of Tree Traversal (class method)

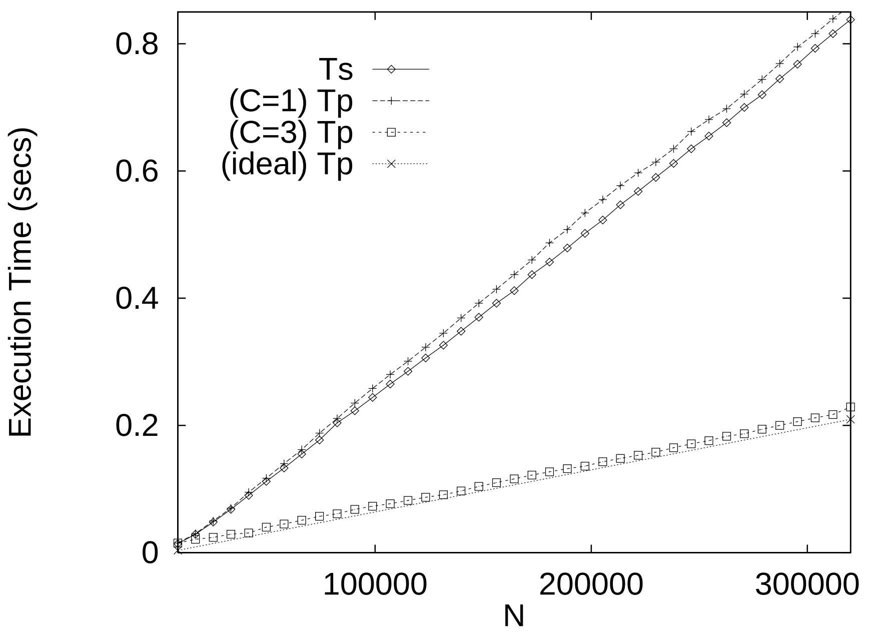

Figure 5: Unbalanced Tree Traversal (instance method)

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()`:

```
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;
    PNodeWorker_n w1_b = null;
    if (d_b <= CUT_DEPTH)
      w1_b = new PNodeWorker_n(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) {}
      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`:

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

In figure 5, we show the results of applying this method to the trees of the previous section with two extra top nodes to introduce some load im-balancing, as illustrated in figure 6.

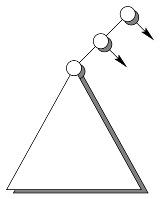

Figure 6: Unbalanced Tree

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. figure 3 and 5).

3.1.2 Overhead Reduction

As alluded to in section 2.2.2, 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 as follows:

```
int compLevel2_par_b(int d_b) {
    if (d_b > CUT_DEPTH)
      return compLevel2_ser();
    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;
        PNode2Worker_n w1_b = null;
        w1_b = new PNode2Worker_n(d_b+1, left);
        r = right.traverse_par_b(d_b+1);
        try { w1_b.join(); }
           catch(InterruptedException e) {}
        l = w1_b.result;
        return (l > r) ? (l+1) : (r+1);
      }
   }
}
```

In figure 7, 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 im-balancing.

3.2 Quick Sorting

As an example of a typical divide-and-conquer algorithm, consider the following implementation of quick-sorting [6] in which, as advocated in [15] small sub-arrays are sorted by insertion sorting to prevent further recursive method invocations for small sub-arrays:[1]

[1]Alternatively, we could ignore small sub-arrays during the recursion and apply a single insertion sort to the whole array afterwards. This approach, however, is less amenable to parallelization.

```
public class Sort {
  ...
  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  );
    }
  }
}
```

3.2.1 Straightforward Parallelization

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 8, 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 9, we show the corresponding speedup. The reason that even for well-balanced method invocation trees the speedup is never optimal becomes immediately apparent from the discussion in section 2.1. 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 10, 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 im-balancing due to im-balanced invocation trees can be resolved by starting additional threads.

3.2.2 Shared Tree-Control

So far, we have seen that there is a clear trade-off between starting many threads to improve load-balancing, and limiting the number of threads to avoid the overhead that is due to contention between running threads. Therefore, in this section we explore whether keeping the number of threads that are actually running equal to the number of processors allows for larger cut-depths in an attempt to improve load-balancing.

As illustrated in figure 11, the parallel execution of a method will be coordinated by an instance of the class `TreeControl`. In this class, an instance variable `running` records the number of actually running threads. This variable is initialized to 1, because initially only one main thread is executing a multi-way recursive method. Instance variable `numCPU` must be set to the number of available processors.

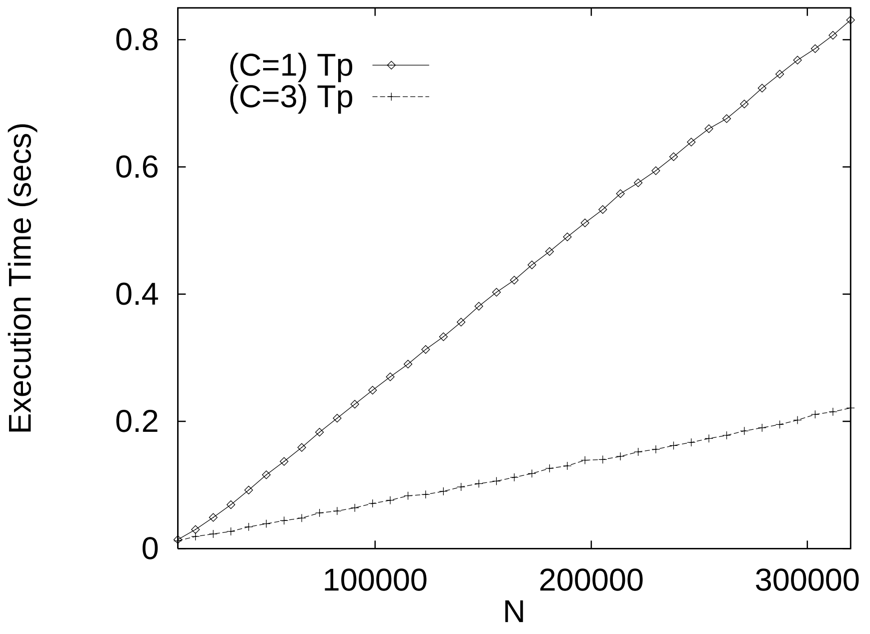

Figure 7: Improved Unbalanced Tree Traversal (instance method)

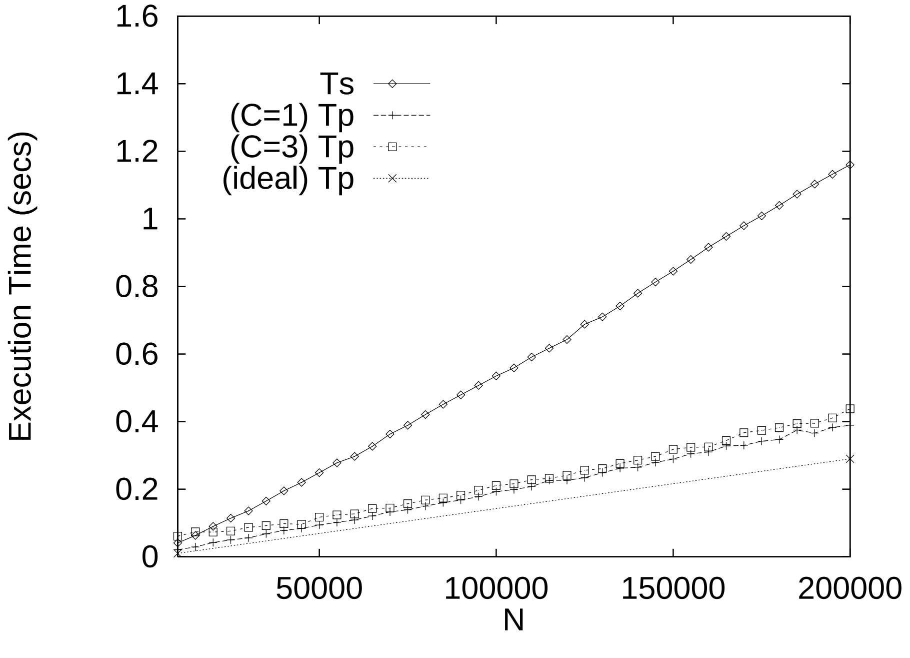

Figure 8: Quick/Insertion Sorting of Reversed Integer Arrays

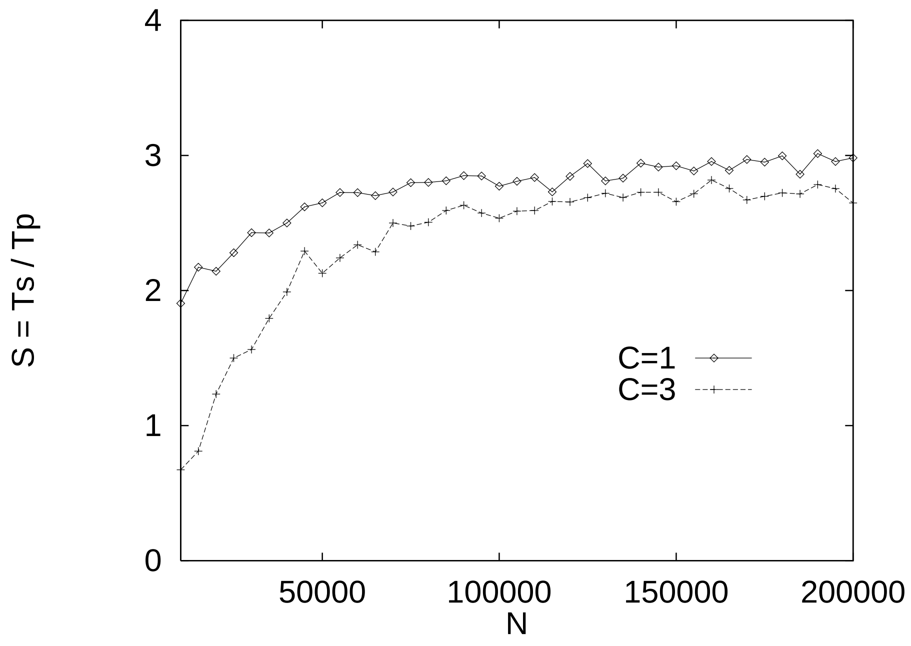

Figure 9: Speedup of Quick/Insertion Sorting of Reversed Integer Arrays

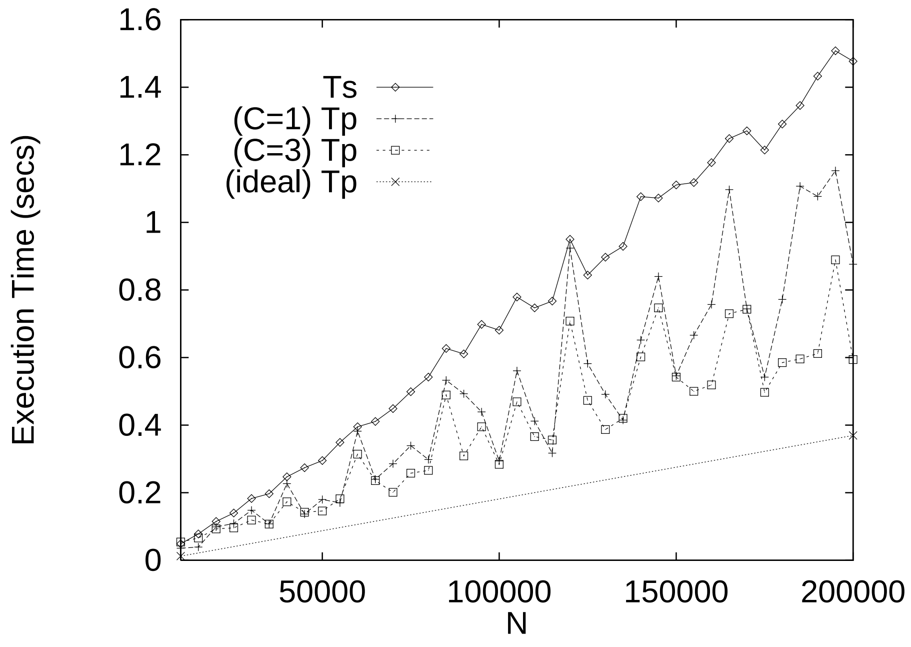

Figure 10: Quick/Insertion Sorting of Random Integer Arrays

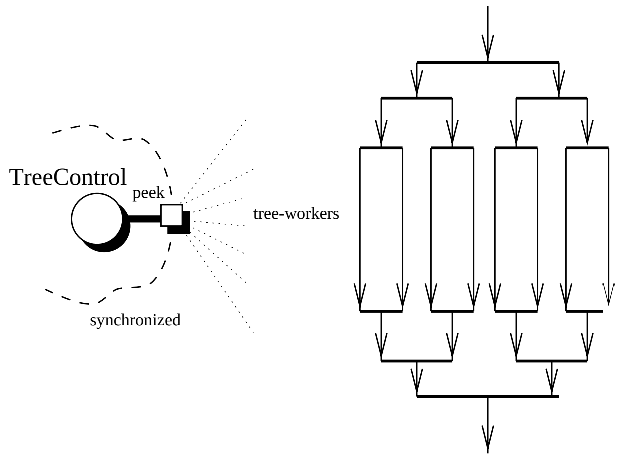

Figure 11: Shared Tree-Control

A volatile boolean variable `peek` will be used for efficiency purposes to determine *asynchronously* whether `running` is less than `numCPU`:

```
class TreeControl {
  private  int     running  = 1, numCPU
  volatile boolean peek;

  TreeControl(int nC) {
    numCPU = nC;
    peek   = (running < numCPU);
  }
  ...
}
```

Instance methods `incRunning()` and `decRunning()` are used to increment and decrement the number of running threads. Because an instance of `TreeControl` is shared among all threads that are executing a particular parallel multi-way recursive method, mutual exclusion during updating the shared variables is enforced by making both method synchronized:

```
synchronized boolean incRunning() {
  if (running < numCPU) {
    if (++running == numCPU)
      peek = false;
    return true;
  }
  return false;
}
```

```
synchronized void decRunning() {
  if (running-- == numCPU)
    peek = true;
}
```

The parallelization of a parallel multi-way recursive method now proceeds similar to the method explained in section 2.2 with the following changes.

Each tree-worker that is constructed in the first step (see section 2.2.1) now has two additional instance variables `c_x` and `first`. Moreover, an additional method `free()` is supplied, which is called at end of the `run()` method. Because the thread that initiated the tree-worker will also call `free()`, mutual exclusion is enforced by making this method synchronized:

```
class TreeWorker_x extends Thread {
  private TreeControl c_x;
  private boolean     first = true;
  ...
  TreeWorker_x(TreeControl c_x, ...) {
    this.c_x = c_x;
    ...
    start();
  }
  synchronized void free() {
    if (first) {
      c_x.decRunning();
      first = false;
    }
  }
  public void run() {
    ...
    free();
  }
}
```

In the second step (see section 2.2.2), an additional parameter `c_x` of type `TreeControl` is added to the method `myMethod_par_x()` in order to make the tree-control accessible in all invocations of this method. Moreover, the fork is now implemented as follows:

```
TreeWorker_x wi_x = null;
if ((d_x <= CUT_DEPTH) && (c_x.peek) && (c_x.lockWorker()) )
  wi_x = new TreeWorker_x(c_x, d_x+1, ...);
else
  ...
```

Now, a running thread will only start another thread it can successfully invoke method `incRunning()` on the shared tree-control. For efficiency purposes, however, first the boolean `peek` is consulted *asynchronously.* This variable is declared volatile to prevent the compiler from performing optimizations that disable the visibility of any changes to this variable made by other threads (e.g. by keeping the variable in a register). Moreover, although several threads may see that `peek` becomes true if only one thread can successfully execute `incRunning()` thereafter, true race conditions are prevented by enforcing mutual exclusion in `incRunning()` itself.

The join with each thread is now implemented as follows:

```
if (wi_x != null) {
  wi_x.free();
  try { wi_x.join(); }
     catch(InterruptedException e) {}
  ri = wi_x.result_x;
}
```

In this manner, the number of actually running threads is decreased as soon as either the initially running executes `join()` and becomes idle, or the new thread finishes its `run()`-method first. Because in any case, the initial thread will continue to run after the join has been executed, the shared variable `first` is used to ensure that the second call to `free()` has no effect.

Finally, in the last step (see section 2.2.3) a shared-tree control is made available to all method invocations as follows, where `NUM_CPU` denotes the number of processors that are available:

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

In figure 12, we show the execution time of parallel quick/insertion-sorting without a shared tree-control for cut-depths $c = 3$ and $c = 9$, and with a shared-tree control for cut-depth $c = 9$. Obviously, increasing the cut-depth without limiting the number of threads that are actually running has a dramatic impact on the performance. Adding a shared tree-control, however, allows for larger cut-depths. Unfortunately, due to the overhead of synchronously accessing a shared-tree control, the effects of the corresponding improved load-balancing become only clear for sorting arrays with a length that exceeds $1,000,000$.

3.3 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. [15]). An implementation that handles small sub-arrays differently to improve performance is shown below, where we assume that BIT$[i] = 2^i$.

```
class Sort {
 ...
  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--l
          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 13, 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 in figure 12. 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 kept more balanced.

3.4 Merge Sorting

In this section, we discuss the results of the parallelization of an array and linked-lists version of merge-sorting (see e.g. [15]).

3.4.1 Merge Sorting of Arrays

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

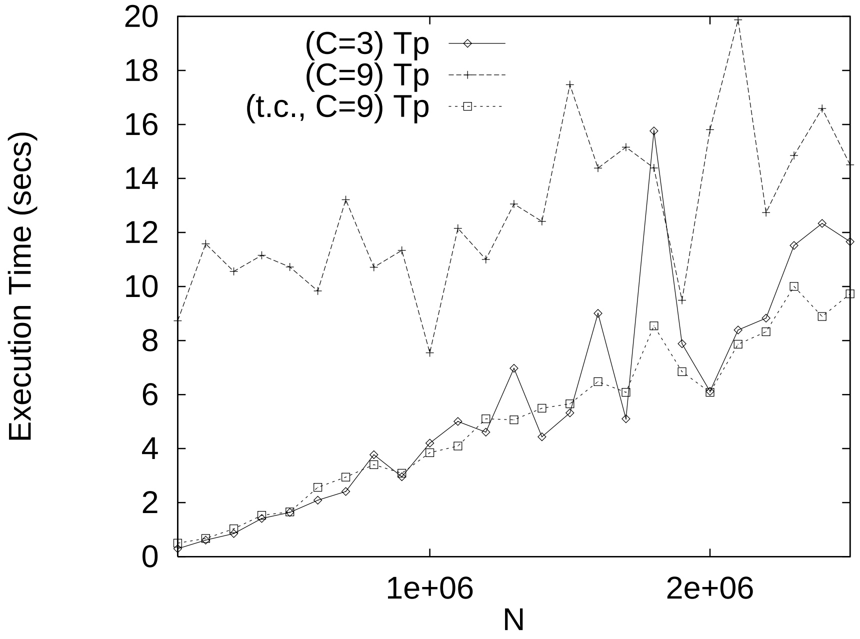

Figure 12: Quick/Insertion Sorting of Large Random Integer Arrays

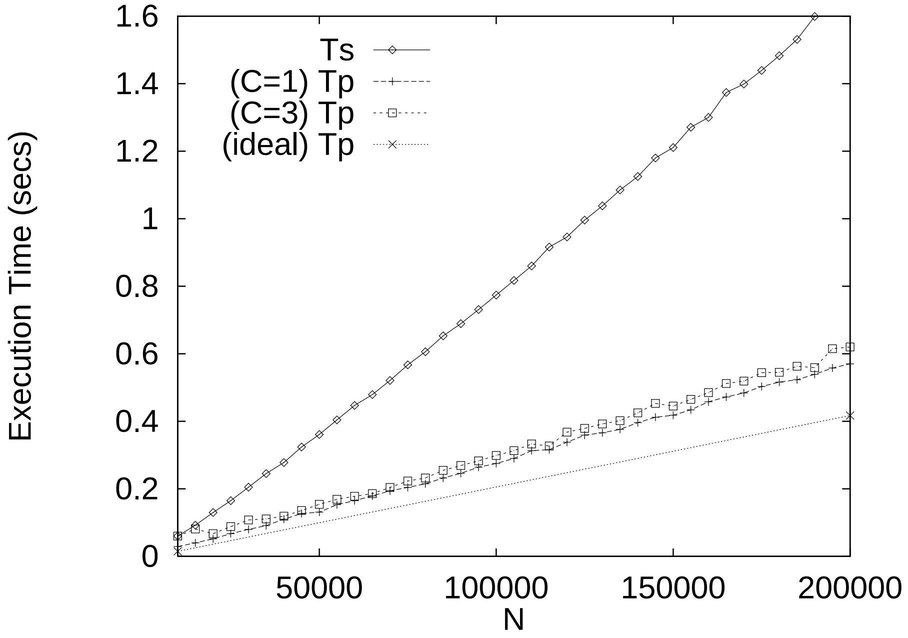

Figure 13: Radix-Exchange/Insertion Sorting of Random Integer Arrays

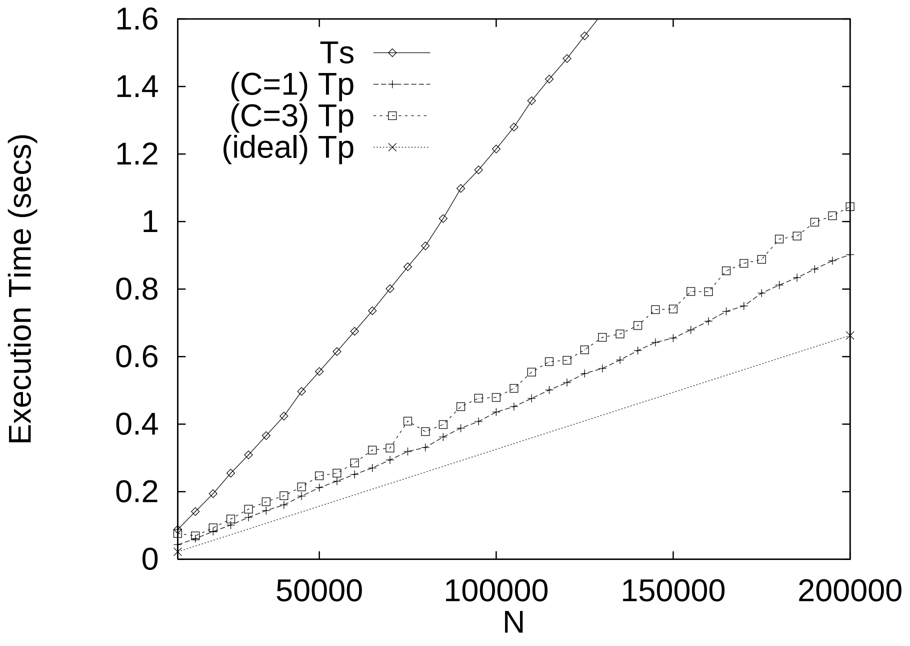

Figure 14: Merge/Insertion Sorting of Random Integer Arrays

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

  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 14, we show execution time of the serial and parallel version applied to the random integer arrays used in figure 12. 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.

3.4.2 Merge Sorting of Linked Lists

Consider a linked-list of integers that are implemented using the following class `List` (cf. [3]):

```
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:

```
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 15. 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:

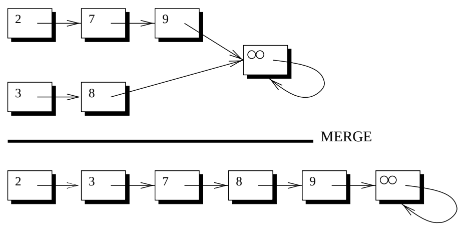

Figure 15: Merging two Linked Lists

```
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 16, 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 17 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.

3.5 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:

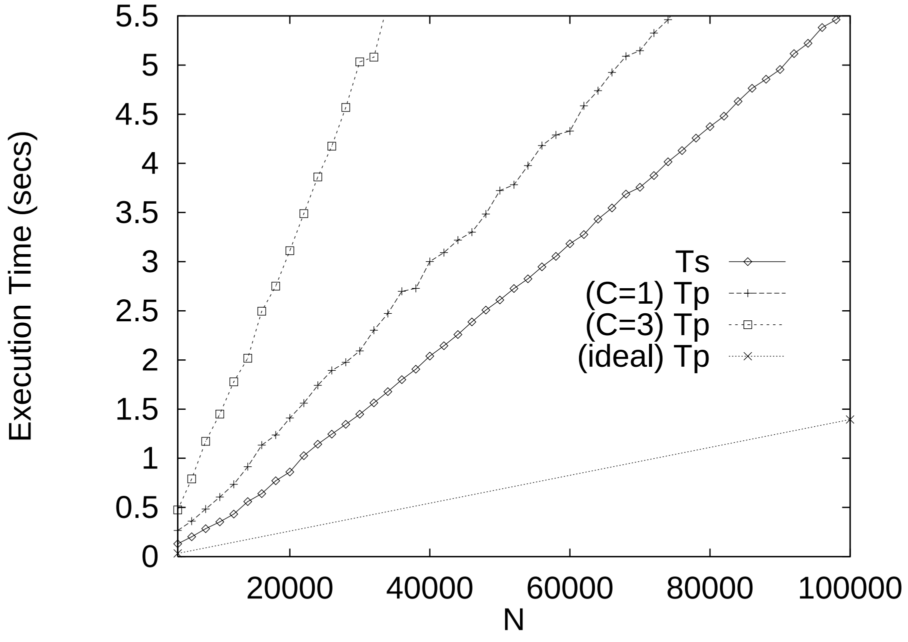

Figure 16: Linked List Merge Sorting (memory allocation)

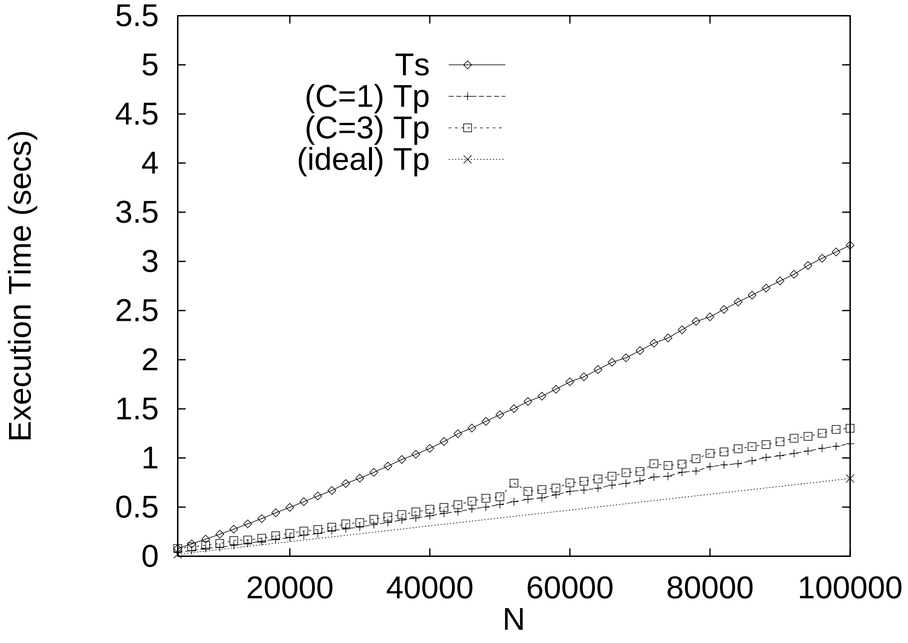

Figure 17: Linked List Merge Sorting (no memory allocation)

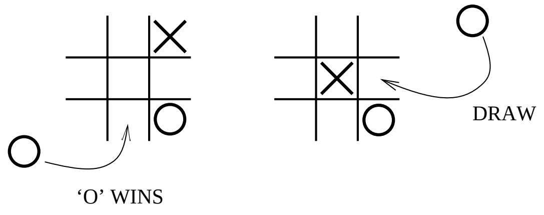

Figure 18: Tic-Tac-Toe

```
class TicTacToe {
   ...
  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;
  }
}
```

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 18, 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 {
  private int     d_x;
  private int[][] board;
  private boolean b;
  int     result_x;
  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_x = TicTacToe.minMax_par_x(d_x, 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:

```
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_x;
           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.

4 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 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 Risc System/600 G30 to show the potential of the parallelization of multi-way recursive methods. 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 invocations requires time proportional to the remain input size (e.g. quick-sorting), the best parallel execution time that can be obtained using the simple parallelization method of this paper suffers from some initial linear terms. Some load im-balancing that is inherent to using a static allocation scheme can be alleviated by increasing the cut-depth to start a few additional threads. In this case, keeping the number of threads that are actually running equal to the number of available processors may improve the performance for long running algorithms.

References

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

[2] Aart J.C. Bik and Dennis B. Gannon. A strategy for exploiting implicit loop parallelism in java programs. Technical Report TR-465, Computer Science Department, Indiana University, 1996.

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

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

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

[6] C.A.R. Hoare. Quick-sorting. *Communications of the ACM.*

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

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

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

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

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

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

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

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

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

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