Online Computation of Critical Paths for Multithreaded Languages

Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa

Department of Information Science, Faculty of Science, University of Tokyo
E-mail: {oyama,tau,yonezawa}@is.s.u-tokyo.ac.jp

Abstract. We have developed an instrumentation scheme that enables programs written in multithreaded languages to compute a critical path at runtime. Our scheme gives not only the length (execution time) of the critical path but also the lengths and locations of all the subpaths making up the critical path. Although the scheme is like Cilk's algorithm in that it uses a "longest path" computation, it allows more flexible synchronization. We implemented our scheme on top of the concurrent object-oriented language Schematic and confirmed its effectiveness through experiments on a 64-processor symmetric multiprocessor.

1 Introduction

The scalability expected in parallel programming is often not obtained in the first run, and then performance tuning is necessary. In the early stages of this tuning it is very useful to know what the *critical path* and how long it is. The length of an execution path is defined as the amount of time needed to execute it, and the critical path is the longest one from the beginning of a program to the end. One of the advantages of getting critical path information is that we can use it to identify the cause of low scalability and to pinpoint the "bottleneck" program parts in need of tuning.

This paper describes a simple scheme computing critical paths on-the-fly, one that when a program is compiled inserts instrumentation code for computing the critical paths. An instrumented program computes its critical path while it is running, and just after termination it displays critical-path information on a terminal as shown in Fig. 1. Our instrumentation scheme is constructed on top of a high-level parallel language supporting *thread creation* and *synchronization via first-class data structures (communication channels)*. Because the instrumentation is expressed as rules for source-to-source translation from the parallel language to the same language, our scheme is independent of the implementation details of parallel languages such as the management of activation frames. Although our primary targets are shared-memory programs, our scheme can also be applied to message-passing programs. We implemented the scheme on top of the concurrent object-oriented language Schematic [7, 8, 12] and confirmed its effectiveness experimentally.

This work makes the following contributions.

```
frame entry point                      frame exit point              elapsed time    share
=============================================================================================
main()                              ---  move_molecules(mols,100)          741 usec     9.2%
spawn                                                                      10 usec     0.1%
move_molecules(mols, n)             ---  spawn move_one_mol(mol[i])         39 usec     0.5%
spawn                                                                      10 usec     0.1%
move_one_mol(molp)                  ---  calc_force(molp, &f)             366 usec     4.6%
spawn                                                                      10 usec     0.1%
calc_force(molp, fp)                ---  return                          4982 usec    61.9%
communication                                                              15 usec     0.2%
sumf += f  (in move_one_mol)        ---  send(r, sumf)                    504 usec     6.3%
communication                                                              15 usec     0.2%
v = recv(r)  (in move_molecules)    ---  send(s, v*2)                     128 usec     1.6%
communication                                                              15 usec     0.2%
u = recv(s)  (in main)              ---  die                            1207 usec    15.0%
=============================================================================================
Critical path length:                                                     8042 usec   100.0%
```

Fig. 1. An example of critical-path information displayed on a terminal. All subpaths in a critical path are shown.

- It provides an instrumentation scheme for languages where threads synchronize via first-class synchronization data. As far as we know, no scheme for computing critical paths for this kind of parallel languages has yet been developed.
- Our instrumentation scheme also gives the length of each subpath in a critical path. As far as we know, previous schemes either provide the length of only the critical path [1, 6] or provide a list of procedures and the amount of time each contributes to the critical path [3].
- The usefulness of our scheme has been demonstrated through realistic applications running on symmetric multiprocessors.

Instrumentation code has usually been inserted into the synchronization parts and entry/exit points of procedures by programmers, but our scheme is implemented by the compiler of a high-level multithreaded language. The compiler inserts instrumentation code automatically, thus freeing programmers from the need to rewrite any source code. In an approach using low-level languages such as C and parallel libraries such as MPI or Pthread, on the other hand, a programmer has to modify a program manually. The source modification approach [10] may require much effort by programmers and result in human errors.

The rest of this paper is organized as follows. Section 2 clarifies the advantages of obtaining critical path information. Section 3 describes our instrumentation scheme and Sect. 4 gives our experimental results. Section 5 describes related work and Sect. 6 gives our conclusion and mentions future work.

2 Benefits of Getting Critical Path Information

Computing critical path information brings us the following benefits because it helps us understand parallel performance.[1]

[1] The usefulness of critical paths for understanding performance is described in [1].

- A critical path length indicates an *upper bound* on the degree to which performance can be improved by increasing the number of processors. When the execution time is already close to critical path length, the use of more processors is probably futile and may even be harmful.
- Critical path information is essential for performance prediction [1].
- Critical path information helps identify the cause of low scalability. If the critical path is very short, for example, low scalability is likely to result from the increase of overhead or workload. If it is close to the actual execution time, it should be shortened.
- Programmers can pinpoint the program parts whose improvement will affect the overall performance and thus avoid tuning unimportant parts.
- A compiler may be able to optimize a program adaptively by using critical path information. This topic is revisited in Sect. 6.

3 Our Scheme: Online Longest Path Computation

3.1 Target Language

Our target is the C language extended by the addition of thread creation and channel communication.[2] *Channels* are data structures through which threads can communicate values. Channels can contain multiple values. If a channel has multiple values and multiple receiving threads at the same time, which pair communicates is unspecified. Channels can express a wide range of synchronization data including locks, barriers, and monitors. The target language has the following primitives.

spawn $f(x_1, \ldots, x_n)$**:** It creates a thread to calculate $f(x_1, \ldots, x_n)$. A spawned function f must have a `void` type.

send(r, v)**:** It sends a value v to a channel r.

recv(r)**:** It attempts to receive a value from a channel r. If r has a value, this expression receives that value and returns it. Otherwise, the execution of the current thread is suspended until a value arrives at r.

die: It terminates the whole program after displaying the critical path from the beginning of the program up to this statement.

Since we can encode sequential function calls and return statements straightforwardly with thread creation and channel communication, we do not consider these calls and statements separately.

Figure 2 shows a sample program in the target language. The function `fib` creates threads for computing the values of recursive calls. The values are communicated through channels. l's are labels, which are used later for representing program points. Labels are added automatically by the compiler in a preprocessing phase.

[2] Our scheme was originally designed for process calculus languages. For readability, however, this paper describes the scheme in an extension to C.

```
fib(r, x) {
  l1 : if (x < 2) {
         l2 : send(r, 1);
       } else {
             x1 = x - 1;
             r1 = newChannel();
         l3 : spawn fib(r1, x1);
             x2 = x - 2;
             r2 = newChannel();
         l4 : spawn fib(r2, x2);
         l5 : v2 = recv(r2);
         l6 : v1 = recv(r1);
             v = v1 + v2;
         l7 : send(r, v);
      }
}

main() {
  s = newChannel();
  spawn fib(s, 2);
  u = recv(s);
  print(u);
  die;
}
```

Fig. 2. A multithreaded program to compute the second Fibonacci number.

Data communicated by threads must be contained in channels and accessed through send and receive operations even in shared-memory programs because our scheme ignores the producer-consumer dependency of the data not communicated through channels.

3.2 Computed Critical Paths

A *dynamic* structure of a parallel program can be represented by a directed acyclic graph (DAG). Figure 3 shows a DAG for the program in Fig. 2. A node in a DAG represents the beginning of a thread or one of the parallel primitives spawn, send, recv, or die. A DAG has three kinds of directed edges:

Arithmetic edges They represent intraframe dependency between nodes. An arithmetic edge from a node X to a node Y is weighted with a value representing the amount of time that elapses between the leaving of X and the reaching of Y.

Spawn edges They represent dependency from a spawn node to a node for the beginning of a spawned thread. Spawn edges are weighted with a predefined constant value representing the difference between the time the spawn operation starts and the time a newly-created thread becomes runnable.

Communication edges They represent dependency from a sender node to its receiver node. Communication edges are weighted with a predefined constant value representing communication delay between the time the send operation starts and the time the sent value becomes available to the receiver.

A path is weighted with the sum of the weights of included edges, and the critical path we compute is the one from the node for the beginning of a program to the die node with the largest weight. The critical path *length* we compute is represented in a DAG as the weight of a critical path.

Unfortunately, the shape of the DAG representing program execution may vary between different runs. For example, communication edges may connect different pairs of nodes in different runs because of the nondeterministic behavior

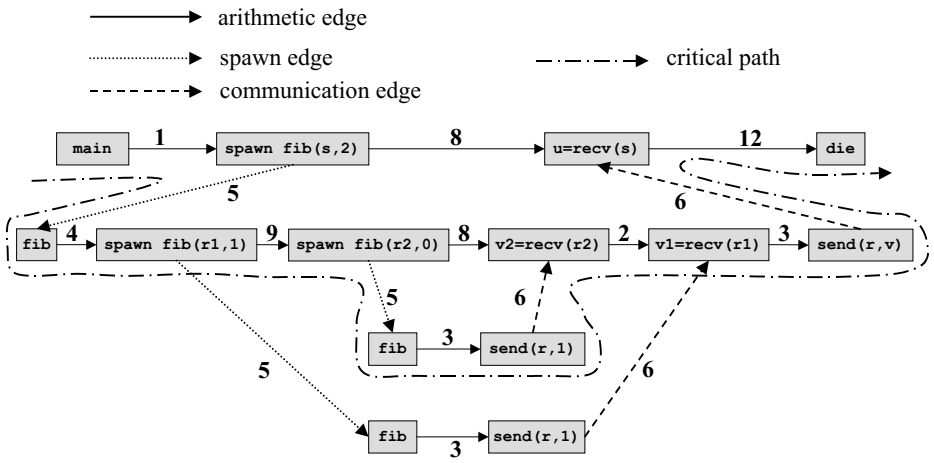

Fig. 3. A DAG representing a dynamic structure of the program in Fig. 2.

of communication. Furthermore, the execution time for each program part will change for various reasons (e.g., cache effects). Therefore, we compute the critical path of *a DAG that is created in an actual run*.

The DAG model described above assumes the following.

- Thread creation cost is constant (all the spawn edges in Fig. 3 have the same weight).
- Communication cost is constant (all the communication edges in Fig. 3 have the same weight).
- Sends, receives, and spawns themselves are completed instantaneously (the nodes in Fig. 3 have no weight).

The first and second assumptions do not take data locality into account. The cost of creating a thread, for example, depends on the processor the thread is placed on; the cost will be small when the thread gets placed on the processor its creator is on. Similarly, interprocessor communication is more costly than intraprocessor communication; two threads on the same processor can communicate through the cache but those on different processors cannot. Our current implementation assumes that a newly created thread and its creator are *not* on the same processor and that all communication is interprocessor communication. Improving our scheme by taking data locality into account would be an interesting thing to do.

The third assumption is reasonable if sends, receives, and spawns are cheap enough compared with the other program parts represented by DAG edges. Otherwise the cost of the operations should be taken into account by giving DAG nodes nonzero weights.

A critical path is shown to programmers in the form of a sequence of *subpaths*, of which there are the following three kinds:

Intraframe subpaths Each is a sequence of arithmetic edges representing the part of execution that begins when control enters a function frame and ends when control leaves the frame. The information about an intraframe subpath consists of its location in the source code and its execution time. The location is expressed by specifying the entry and exit points of a function frame. The entry points are the beginnings of a function body and receives. The exit points are sends and spawns.

Spawn subpaths Each consists of one spawn edge. The information about a spawn subpath consists of the constant weight only, so the information about any spawn subpath is the same as that about any other.

Communication subpaths Each consists of one communication edge. The information about a communication subpath also consists only of the constant weight, so the information about any communication subpath is also the same as that about any other.

3.3 Instrumentation

Our scheme computes critical paths through a *longest path computation* (Fig. 4). Programs instrumented with our scheme maintain the information on the critical path from the beginning of a program to the currently executing program point. In a spawn operation the information about the critical path to the spawn operation is passed to the newly created thread. In a send operation the information about the critical path to the operation is attached to a sent value. A receive operation compares the length of the critical path to the receive with the length of the critical path to the sender of the received value *plus* the weight of a communication edge. The larger of the compared values is used as the length of the critical path to the program part following the receive.

Figure 5 shows an instrumented version of the function `fib` in Fig. 2. An instrumented program maintains in the three local variables (represented in Fig. 5 as *el*, *et*, and *cp*) the following three values:

Entry label (*el*): It represents a program point that indicates the beginning of the currently executing subpath.

Entry time (*et*): It represents the time when the currently executing subpath started.

Critical path (*cp*): It represents a sequence of subpaths that makes up the longest path up to the beginning of the currently executing subpath.

The function `currTime()` returns the current value of a timer provided by the operating system.[3] [*v*,*cp*] is a data structure comprising a communicated value v and the information on a critical path *cp*. `length`(*cp*) returns the length of a critical path *cp*. $\{b, e, t\}$ is a data structure representing an intraframe subpath that begins at a label b, ends at a label e, and takes the amount t of time to execute. `addInFrSubPath`(*cp*, $\{b, e, c\}$) appends an intraframe subpath $\{b, e, c\}$ to

[3] Our scheme does not assume a *global* timer shared among processors. Threads communicate critical path information only, not the value of a timer.

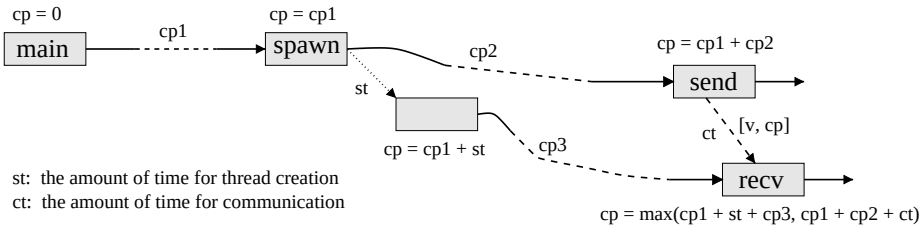

Fig. 4. Computation of the longest path.

a critical path *cp* and returns the extended critical path. `addSpawnSubPath`(*cp*) and `addCommSubPath`(*cp*) respectively append a spawn subpath and a communication subpath to a critical path *cp* and return the extended critical path.

The length of the critical path to the current program point is equal to

`length`(*cp*) + (the current time − *et*)

while the control is in arithmetic edges. The value (the current time − *et*) is called a *lapse* and is the time between the entry label and the current program point. The relations between *et*, *el*, and *cp* are illustrated in Fig. 6, and the lapses kept by the variables t_1, t_2, t_3, t_4, and t_5 in the program in Fig 5 are illustrated in Fig. 7.

As noted earlier, our DAG model assumes that sends, receives, and spawns take no time and thus that there is no time difference between the end of one edge and the start of the edge executed next. In actual execution, however, there are delays between two DAG edges. The delays are descheduling time and suspension time. An underlying thread system may delay the start of a runnable thread because it is not always possible to assign a processor to all runnable threads. A thread may be suspended from the time it starts trying to receive a value until the time it receives the value.

Our instrumentation excludes these kinds of delays from a critical path. We "stop the timer" when a thread reaches a node and "restart the timer" when it leaves the node. When reaching a node, a thread calculates the current lapse and sets it to a variable *t*. When leaving the node, the thread "adjusts" the value of the variable *et* to make the current lapse still equal to *t*.[4]

Figure 8 shows source-to-source translation rules for our instrumentation.

Function Definition A function obtains an additional argument *cp*, through which a critical path is passed between function frames. S' comes up out of S by applying all other transformation rules.

Spawn A critical path *cp* is extended with an intraframe subpath and a spawn subpath. The resulting extended critical path is passed to a spawned function.

[4] The processing enables our scheme to be used in languages with garbage collection (GC). The GC time can be excluded from computed critical paths by allowing threads to jump to a GC routine only when they are in a DAG node.

```
fib(r, x, cp) {
  el = l1;
  et = currTime();
  if (x < 2) {
    t0 = currTime() - et;
    cp' = addInFrSubPath(cp, {el, l2, t0});
    send(r, [1, cp']);
    et = currTime() - t0;
  } else {
    x1 = x - 1;
    r1 = newChannel();

    t1 = currTime() - et;
    cp' = addInFrSubPath(cp, {el, l3, t1});
    cp'' = addSpawnSubPath(cp');
    spawn fib(r1, x1, cp'');
    et = currTime() - t1;

    x2 = x - 2;
    r2 = newChannel();

    t2 = currTime() - et;
    cp' = addInFrSubPath(cp, {el, l4, t2});
    cp'' = addSpawnSubPath(cp');
    spawn fib(r2, x2, cp'');
    et = currTime() - t2;

    t3 = currTime() - et;
    [v2, cp'] = recv(r2);
    cp'' = addCommSubPath(cp');
    if (t3 + length(cp) < length(cp'')) {
      cp = cp'';
      el = l5;
      et = currTime();
    } else {
      et = currTime() - t3;
    }

    t4 = currTime() - et;
    [v1, cp'] = recv(r1);
    cp'' = addCommSubPath(cp');
    if (t4 + length(cp) < length(cp'')) {
      cp = cp'';
      el = l6;
      et = currTime();
    } else {
      et = currTime() - t4;
    }

    v = v1 + v2;

    t5 = currTime() - et;
    cp' = addInFrSubPath(cp, {el, l7, t5});
    send(r, [v, cp']);
    et = currTime() - t5;
  }
}
```

Fig. 5. Instrumented version of the function `fib` in Fig. 2.

Send A critical path to a send statement is computed and attached to a value.

Receive The length of the critical path to a receive is compared with the length of the received critical path extended with a communication subpath. If the former is shorter, the extended critical path is set to the variable *cp* and a new subpath starts. Otherwise the thread simply adjusts the entry time.

If critical paths are to be computed accurately, the underlying system should have the following characteristics.

- Fast and scalable execution of `currTime()` (i.e., negligible overhead for its execution)
- Threads that are not preempted during the execution of an arithmetic edge. In other words, once a thread acquires a processor and starts executing an arithmetic edge, it will not lose the processor until it reaches a node. Otherwise, the amount of time that elapses while the thread is descheduled would be included in the weight of the arithmetic edge.

In our experimental platform the first assumption holds but the second does not. The experimental results were therefore affected by the measurement perturbation. It seems difficult to solve the perturbation problem without the cooperation of a thread scheduler.

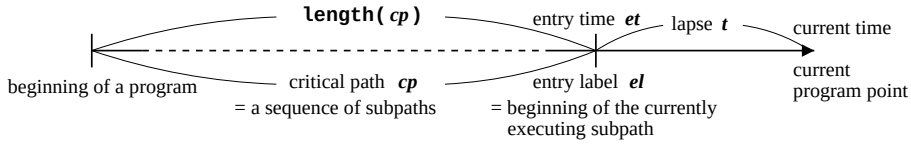

Fig. 6. The relation between the times kept by programs instrumented with our scheme.

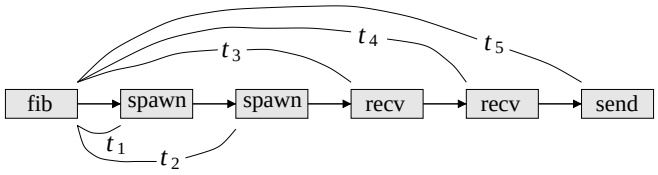

Fig. 7. The lapses kept by the variables t_1, t_2, t_3, t_4, and t_5 in the program in Fig. 5.

3.4 Potential Problems and Possible Solutions

One potential problem with our scheme is the instrumentation overhead, which increases the overall execution time. We can reduce overall execution time by not instrumenting selected functions. Calls to those functions would then be regarded as arithmetic operations. On the other hand, the critical path length computed by our scheme does not include most of the instrumentation overhead: it does not include the amount of time needed for appending a subpath, for attaching critical path information to sent values, or for comparing two critical paths in receives. We thus expect the critical path length computed with an instrumented program is close to that of the uninstrumented program.

Another potential problem is that storing critical path information may require a large amount of memory, even though our scheme requires less memory than do the tracefile-based schemes that record the times of all inter-thread events. Memory usage can be reduced with the technique that *compresses* path information by finding repeated patterns [5]. It would also be possible to, when the memory usage exceeds a threshold, discard the information on subpaths and begin keeping only critical path length.

4 Experiments

We implemented our scheme on top of the concurrent object-oriented language Schematic [7, 8, 12] and tested its usefulness through experiments on a symmetric multiprocessor, the Sun Ultra Enterprise 10000 (UltraSPARC 250MHz $\times$ 64, Solaris 2.6). To get the current time we used the `gethrtime` function in the Solaris OS. We tested our scheme on the following application programs.

Function Definition

```
f(x1,...,xn) { l: S } ⟶
  f(x1,...,xn,cp) {
    el = l;
    et = currTime();
    S'
  }
```

Spawn

```
l: spawn f(x1,...,xn); ⟶
  t = currTime() - et;
  cp' = addInFrSubPath(cp,{el,l,t});
  cp'' = addSpawnSubPath(cp');
  spawn f(x1,...,xn,cp'');
  et = currTime() - t;
```

Termination

```
l: die; ⟶
  t = currTime() - et;
  cp' = addInFrSubPath(cp,{el,l,t});
  printCriticalPath(cp');
  exit();
```

Send

```
l: send(r,v); ⟶
  t = currTime() - et;
  cp' = addInFrSubPath(cp,{el,l,t});
  send(r,[v,cp']);
  et = currTime() - t;
```

Receive

```
l: v = recv(r); ⟶
  t = currTime() - et;
  [v,cp'] = recv(r);
  cp'' = addCommSubPath(cp');
  if (t + length(cp) < length(cp'')) {
    /* Sender is later */
    cp = cp'';
    el = l;
    et = currTime();
  } else {
    /* Receiver is later */
    et = currTime() - t;
  }
```

Fig. 8. Translation rules for instrumentation to compute critical paths.

Prime A prime generator using the Sieve of Eratosthenes. This program consists of two parts, one that divides prime candidates by smaller primes and another that gathers primes into a list. It is unclear in what ratio the two parts contribute to a critical path.

CKY A parallel context-free grammar parser. It is an irregular program in which a number of fine-grain threads synchronize via channels. Like **Prime**, it consists of two parts. One part for lexical rule application and another for production rule application. It is not obvious in what ratio the two parts contribute to a critical path.

Raytrace A raytracer that calculates RGB values of pixels in parallel. After calculating a pixel value, a processor sends it to a buffer object which manages file I/O. The object caches pixel values and flushes them when 64 values are accumulated. Accesses to the buffer object are mutually excluded and hence the object becomes a bottleneck.

The experimental results are shown in Fig. 9, where each line graph has one line showing the execution time of the instrumented program (Inst'ed), one showing the execution time of the uninstrumented program (Original), and one showing the critical path length computed by the instrumented program (CPL). The bar chart compares the execution time, on a single processor, between uninstrumented and instrumented programs. Garbage collection did not happen in any of the experiments.

In all the applications, as the number of processors increased, the execution time of uninstrumented programs got very close to the computed critical path

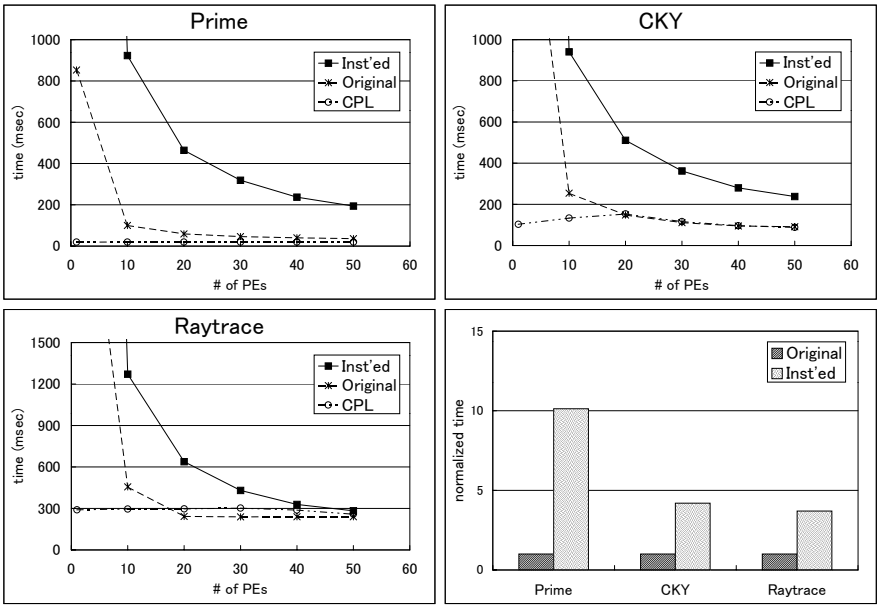

Fig. 9. Measured execution times and computed critical path lengths.

length. The critical path lengths computed with the instrumented programs were extremely close to the best runtimes for the uninstrumented programs. On a single processor the uninstrumented programs were from four to ten times faster than the instrumented ones.

The critical path length we compute can also be used for performance prediction: the performance on large-scale multiprocessors can be predicted from that on a single processor. Cilk's work [1] showed that a program's execution time on P processors is approximately $T_1/P + T_\infty$, where T_1 is the execution time on a single processor and T_∞ is the critical path length.[5] For **Prime** there was almost no difference (5% or less) between the actual execution time and the predicted execution time.

The execution time for each part of **CKY** depends on thread scheduling order. This order changed as the number of processors was increased, and thus the length of the critical path also changed. The critical path length computed for **Raytrace** was greater than the execution time. We do not know why.

We acquired information useful for future tuning. More than 95 percent of the critical path length for **Prime** was due to the time needed for gathering primes into a list. The application of lexical rules made up only four percent of the critical path length for **CKY**.

[5] $T_1/P + T_\infty$ is the simplest expression of all the ones they proposed for performance prediction. More sophisticated expressions can be found in their papers.

The experimental results show that our scheme gives a good estimate of the upper bound on performance improvement, that it provides information useful in performance prediction, and that it provides information useful for tuning programs. They also show, however, that it makes programs much slower.

5 Related Work

Cilk. As far as we know, Cilk [1, 6] is the only high-level parallel language that provides an online computation of the critical path. It also computes the critical path of a DAG created in an actual run. Since Cilk is based on a fully-strict computation, it deals only with implicit synchronization associated with fork-join primitives. It should also be noted that in the Cilk scheme, the shape of a DAG does not vary in different runs and the spawn and communication edges have zero weight.

Paradyn. Hollingsworth's work [3] computes at runtime a critical path profile that gives us a list of procedures whose "contributions" to the critical path length are large. The language model Hollingsworth uses is essentially the same as ours, although the communication in his target language is performed not through channels but in the form of message-passing designating a peer. His scheme, like ours, attaches critical path information to messages. But our scheme, unlike his, displays information about the subpaths in a critical path.

Tracefile-Based Offline Schemes. In Dimemas [10] parallel programs call the functions in the instrumented communication library and generate a tracefile after execution. The tracefile contains the parameters and timings of *all* communication operations. A critical path is constructed a posteriori in a tracefile-based *simulation* to which a programmer gives architecture parameters and task-mapping directions. ParaGraph [2] is a tool for visualization of the tracefiles generated by the instrumented communication library PICL. ParaGraph can visualize critical path information. In our scheme the instrumented programs maintain information only on the subpaths that may be included in a critical path. Our on-the-fly computation therefore requires much less memory than do tracefile-based computations.

6 Conclusion and Future Work

We developed an instrumentation scheme that computes critical paths on-the-fly for multithreaded languages supporting thread creation and channel communication. We implemented the scheme and confirmed that the critical path information computed makes it easier to understand the behavior of parallel programs and gives a useful guide to improving the performance of the programs. If our scheme is incorporated into parallel languages, performance tuning in those languages will become more effective.

One area for future work is adaptive optimization utilizing critical path information. A critical path can be shortened if a compiler recompiles with powerful time-consuming optimizations a set of performance-bottleneck procedures whose share of the critical path length is large. We may be able to extend the framework of HotSpot [11] and Self [4], which compiles frequently executed bytecode dynamically, by giving higher compilation priority to bottleneck procedures. Another adaptive optimization exploiting critical path information would be to have a runtime give higher scheduling priority to threads executing a critical path. Our previous work [9] gives heuristics for efficient thread scheduling in synchronization bottlenecks, on which multiple operations are blocked and are likely to be in a critical path. Critical path information makes more adaptive and more reasonable scheduling possible without the help of heuristics. The use of this information might thus become an essential part of adaptive computation.

References

1. M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 212–223, 1998. See also The Cilk Project Home Page `http://supertech.lcs.mit.edu/cilk/`
2. M. T. Heath and J. E. Finger. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, 1991.
3. J. K. Hollingsworth. Critical Path Profiling of Message Passing and Shared-memory Programs. *IEEE Transactions on Parallel and Distributed Systems*, pages 1029–1040, 1998.
4. U. Hölzle and D. Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, pages 229–243, 1994.
5. J. R. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*, pages 259–269, 1999.
6. C. E. Leiserson, 1999. personal communication.
7. Y. Oyama, K. Taura, T. Endo, and A. Yonezawa. An Implementation and Performance Evaluation of Language with Fine-Grain Thread Creation on Shared Memory Parallel Computer. In *Proceedings of 1998 International Conference on Parallel and Distributed Computing and Systems (PDCS '98)*, pages 672–675, 1998.
8. Y. Oyama, K. Taura, and A. Yonezawa. An Efficient Compilation Framework for Languages Based on a Concurrent Process Calculus. In *Proceedings of Euro-Par '97 Parallel Processing*, volume 1300 of *LNCS*, pages 546–553, 1997.
9. Y. Oyama, K. Taura, and A. Yonezawa. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99)*. World Scientific, 1999.
10. Pallas GmbH. *Dimemas.* `http://www.pallas.de/`.
11. Sun Microsystems. *The Java HotSpot^{TM} Performance Engine.*
12. K. Taura and A. Yonezawa. Schematic: A Concurrent Object-Oriented Extension to Scheme. In *Proceedings of Workshop on Object-Based Parallel and Distributed Computation (OBPDC '95)*, volume 1107 of *LNCS*, pages 59–82, 1996.