The Cilkview Scalability Analyzer

Yuxiong He[*] Charles E. Leiserson[†] William M. Leiserson[‡]
Intel Corporation
Nashua, New Hampshire

ABSTRACT

The Cilkview scalability analyzer is a software tool for profiling, estimating scalability, and benchmarking multithreaded Cilk++ applications. Cilkview monitors logical parallelism during an instrumented execution of the Cilk++ application on a single processing core. As Cilkview executes, it analyzes logical dependencies within the computation to determine its work and span (critical-path length). These metrics allow Cilkview to estimate parallelism and predict how the application will scale with the number of processing cores. In addition, Cilkview analyzes scheduling overhead using the concept of a "burdened dag," which allows it to diagnose performance problems in the application due to an insufficient grain size of parallel subcomputations.

Cilkview employs the Pin dynamic-instrumentation framework to collect metrics during a serial execution of the application code. It operates directly on the optimized code rather than on a debug version. Metadata embedded by the Cilk++ compiler in the binary executable identifies the parallel control constructs in the executing application. This approach introduces little or no overhead to the program binary in normal runs.

Cilkview can perform real-time scalability benchmarking automatically, producing gnuplot-compatible output that allows developers to compare an application's performance with the tool's predictions. If the program performs beneath the range of expectation, the programmer can be confident in seeking a cause such as insufficient memory bandwidth, false sharing, or contention, rather than inadequate parallelism or insufficient grain size.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; I.6.6 [**Simulation and Modeling**]: Simulation Output Analysis

General Terms

Measurement, Performance, Theory.

Keywords

Burdened parallelism, Cilk++, Cilkview, dag model, multicore programming, multithreading, parallelism, parallel programming, performance, scalability, software tools, span, speedup, work.

This work was supported in part by the National Science Foundation under Grants 0615215, 0712243, and 0822896 and in part by the Defense Advanced Research Projects Agency under Contract W31P4Q-08-C-0156. Cilk, Cilk++, and Cilkview are registered trademarks of Intel Corporation.

[*]Author's current address: Microsoft Research, Redmond, Washington. Email address: `yuxhe@microsoft.com`.

[†]Author is Professor of Computer Science and Engineering at MIT and an Intel consultant. Email address: `cel@mit.edu`

[‡]Email address: `william.m.leiserson@intel.com.`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.
Copyright 2010 ACM 978-1-4503-0079-7/10/06 ...$10.00.

1. INTRODUCTION

Although the performance of serial application programs can be measured by execution time, multithreaded applications exhibit one additional dimension: ***scalability***. How does execution time scale as the number of processing cores increases. Although performance tools to analyze serial applications are widely available, few tools exist that effectively address scalability issues for multithreaded programs on multicore machines. When a serial application fails to run as quickly as one expects, various tools, such as gprof [26], Intel® VTune™ [44], Intel® Performance Tuning Utility [1], etc., can be used to analyze the program execution and identify performance bottlenecks. Parallel-performance tools, such as Intel® Thread Profiler [30] of VTune and Intel® Parallel Amplifier [29] of Intel® Parallel Studio, can provide important data about a multithreaded application, but since the information gathered is specific to how the computation was scheduled for the particular run, these tools do not address how performance scales.

The Cilkview scalability analyzer,[1] which runs under the Pin [35] dynamic instrumentation framework, gathers statistics during a single instrumented run of a multithreaded Cilk++ [28, 33] application, analyzes the logical parallelism within the computation, and estimates the application's scalability over various numbers of cores. It also provides a framework for benchmarking actual runs of the application so that the actual behavior can be compared with the scalability predictions.

To illustrate Cilkview, consider the problem of analyzing the simple quicksort program shown in Figure 1. First, it is helpful to understand the Cilk++ extensions to C++, which consist of three keywords: `cilk_spawn`, `cilk_sync`, and `cilk_for`. Parallel work is created when the keyword `cilk_spawn` precedes the invocation of a function, as in line 15 of the figure. A function cannot safely use the values returned by its spawned children until it

[1]The examples in this article were produced with an as-yet-unreleased version of Cilkview, but the latest released version (available at `http://software.intel.com/en-us/articles/intel-cilk/`) differs insubstantially.

```
1   // Parallel quicksort

3   #include <algorithm>
4   #include <iterator>
5   #include <functional>
6   #include <cmath>
7   #include <cilkview>

9   using namespace std;

11  template <typename T>
12  void qsort(T begin, T end) {
13    if (begin != end) {
14      T middle = partition(begin, end,
             bind2nd(less<typename
             iterator_traits<T>::value_type>()
             ,*begin));
15        cilk_spawn qsort(begin, middle);
16        qsort(max(begin + 1, middle), end);
17        cilk_sync;
18    }
19  }

21  // Simple test code:
22  int cilk_main() {
23    int n = 100;
24    double a[n];
25    cilk::cilkview cv;
26    cilk_for (int i=0; i<n; ++i) {
27      a[i] = sin((double) i);
28    }

30    cv.start();
31    qsort(a, a + n);
32    cv.stop();
33    cv.dump(``qsort'');

35    return 0;
36  }
```

Figure 1: Parallel quicksort implemented in Cilk++.

executes a `cilk_sync` statement, which acts as a local "barrier." In the quicksort example, the `cilk_sync` statement on line 17 avoids the anomaly that would occur if the recursively spawned function in line 15 did not complete before the return, thus leaving the vector to be sorted in an intermediate and inconsistent state. The keyword `cilk_for`, as in line 26, indicates a parallel `for` loop, which allows all iterations of the loop to operate in parallel.

The Cilkview API allows users to control which portions of their Cilk++ program are analyzed. For example, the program in Figure 1 restricts the analysis to the sorting function by designating "start" and "stop" points in the code on lines 30 and 32, respectively. Without these start and stop points, Cilkview defaults to reporting the full run of the application from beginning to end.

Figure 2 shows the scalability profile produced by Cilkview that results from running the quicksort application in Figure 1 on 10 million numbers. Included in the graphical output are curves indicating estimated upper and lower bounds of performance over different numbers of cores. The area between the upper and lower bounds indicates an estimation of the program's speedup with the given input. Cilkview produces this output from a single instrumented serial run of the application. Section 4 describes the Cilkview output in more detail.

In addition to scalability estimation, Cilkview supports a framework for automatically benchmarking an application across a range of processor core counts. For each benchmark run, Cilkview measures the total elapsed time and plots the results. If scalability analysis is enabled, the benchmark data — shown in Figure 2 as crosses — is overlaid on the scalability profile. In the figure, the benchmark runs represent the actual speedup of the quicksort program using from 1 to 8 cores of an 8-core Intel® Core$^{\text{TM}}$ i7 machine.

To produce the upper-bound estimate for a program's speedup, Cilkview employs the "dag" (directed acyclic graph) model of multithreading [4,5], which is founded on earlier theoretical work on

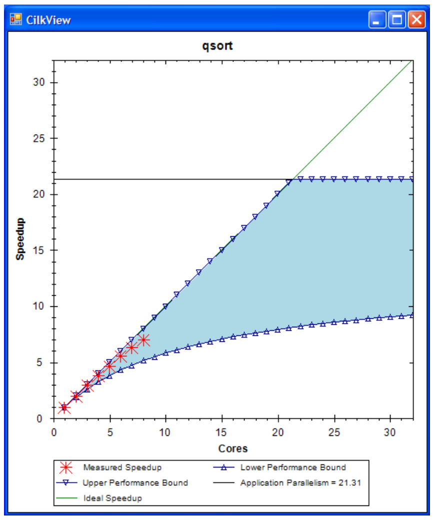


Figure 2: The scalability profile produced by Cilkview for the quicksort program from Figure 1.

dag scheduling [7,17,25]. In this model, parallelism is an intrinsic property of an application execution which depends on its logical construction but not on how it is scheduled. The dag model uses two performance measures — work and span — to gauge the theoretical parallelism of a computation and predict its scalability.

The actual performance of a multithreaded application is determined not only by its intrinsic parallelism, but also by the performance of the runtime scheduler. The Cilk++ runtime system contains a provably efficient work-stealing scheduling algorithm [5,20], which on an ideal parallel computer scales application performance linearly with processor cores, as long as the application exhibits sufficient parallelism. In practice, however, scheduler performance is also influenced by other factors, such as overheads for task migration, which is needed for load balancing, and bandwidth to memory, if the application is memory bound. We introduce the concept of a "burdened" dag, which embeds the task-migration overhead of a job into the dag model. Burdening allows Cilkview to estimate lower bounds on application speedup effectively. Cilkview does not currently analyze the impact of limited memory bandwidth on application scalability.

Cilkview employs dynamic instrumentation [8, 35] to collect scalability statistics during a serial execution of the program code. Since performance analysis can suffer from perturbations due to the instrumentation, Cilkview attempts to minimize this impact in two ways. First, Cilkview operates on the optimized production binary executable, rather than on a specially compiled "debug" version of the application. Debug versions require recompilation and may differ significantly from the production executable. Second, Cilkview relies on metadata that the Cilk++ compiler embeds in the binary executable to identify the parallel control constructs (spawns and syncs), rather than placing calls to null functions or other runtime instrumentation in the execution path. By using metadata, rather than embedding instrumentation in-line in the code, the impact on the performance of the production executable is negligible.

The remainder of this paper is organized as follows. Section 2 provides a brief tutorial on the theory of parallelism, which provides the foundation for Cilkview's upper bounds on speedup. Section 3 describes the Cilk++ runtime system and introduces the concept of "burdening," the basis for Cilkview's lower-bound estimations. With this theoretical understanding of Cilk++'s performance model in hand, Section 4 describes the output of Cilkview in de-

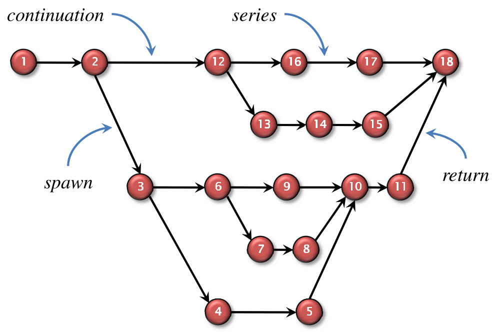


Figure 3: A dag representation of a multithreaded execution. Each vertex is a strand. Edges represent ordering dependencies between instructions.

tail. Section 5 illustrates how Cilkview output can be used to diagnose performance problems in a "stencil" program. Section 6 presents the binary instrumentation framework that Cilkview uses and how Cilkview computes scalability metrics during a job's execution. Section 8 discusses related work. Finally, Section 9 concludes with a discussion of directions for future research.

2. THE DAG MODEL FOR MULTITHREADING

For a parallel program to obtain good performance, the program must exhibit sufficient parallelism. This section reviews the dag model of multithreading [4,5], which provides a general and precise quantification of parallelism based on the theory developed by Graham [25]; Brent [7]; Eager, Zahorjan, and Lazowska [17]; and Blumofe and Leiserson [4,5]. Tutorials on the dag model can be found in [13, Ch. 27] and [33].

The ***dag model of multithreading*** views the execution of a multithreaded program as a set of vertices called ***strands*** — sequences of serially executed instructions containing no parallel control — with graph edges indicating ordering dependencies between strands, as illustrated in Figure 3. We say that a strand x ***precedes*** a strand y, denoted $x \prec y$, if x must complete before y can begin. If neither $x \prec y$ nor $y \prec x$, we say that the strands are in ***parallel***, denoted by $x \parallel y$. In Figure 3, for example, we have $1 \prec 2$, $6 \prec 8$, and $4 \parallel 9$. A strand can be as small as a single instruction, or it can represent a longer chain of serially executed instructions. A ***maximal strand*** is one that cannot be included in a longer strand. We can dice a maximal strand into a series of smaller strands in any manner that is convenient.

The dag model of multithreading can be interpreted in the context of the Cilk++ programming model. Normal serial execution of one strand after another creates a ***serial edge*** from the first strand to the next. A `cilk_spawn` of a function creates two dependency edges emanating from the instruction immediately before the `cilk_spawn`: the ***spawn edge*** goes to the strand containing the first instruction of the spawned function, and the ***continuation edge*** goes to the strand containing the first instruction after the spawned function. A `cilk_sync` creates a ***return edge*** from the strand containing the final instruction of each spawned function to the strand containing the instruction immediately after the `cilk_sync`. A `cilk_for` can be viewed as parallel divide-and-conquer recursion using `cilk_spawn` and `cilk_sync` over the iteration space.

The dag model admits two natural measures that allow us to define parallelism precisely, as well as to provide important bounds on performance and speedup.

The Work Law

The first measure is ***work***, which is the total time spent in all the strands. Assuming for simplicity that it takes unit time to execute a strand, the work for the example dag in Figure 3 is 18.

We can adopt a simple notation to be more precise. Let T_P be the fastest possible execution time of the application on P processors. Since the work corresponds to the execution time on 1 processor, we denote it by T_1. One reason that work is an important measure is that it provides a lower bound on P-processor execution time:

$$T_p \geq T_1/P \ . \tag{1}$$

This ***Work Law*** holds because, in our simple theoretical model, P processors can execute at most P instructions in unit time. Thus, with P processors, to do all the work requires at least T_1/P time.

We can interpret the Work Law (1) in terms of the ***speedup*** on P processors, which using our notation, is just T_1/T_P. The speedup tells us how much faster the application runs on P processors than on 1 processor. Rewriting the Work Law, we obtain $T_1/T_P \leq P$, which is to say that the speedup on P processors can be at most P. If the application obtains speedup P (which is the best we can do in our model), we say that the application exhibits ***linear speedup***. If the application obtains speedup greater than P (impossible in our model due to the Work Law, but possible in practice due to caching and other processor effects), we say that the application exhibits ***superlinear speedup***.

The Span Law

The second measure is ***span***, which is the maximum time to execute along any path of dependencies in the dag. Assuming that it takes unit time to execute a strand, the span of the dag from Figure 3 is 9, which corresponds to the path $1 \prec 2 \prec 3 \prec 6 \prec 7 \prec 8 \prec 10 \prec 11 \prec 18$. This path is sometimes called the ***critical path*** of the dag, and span is sometimes referred to in the literature as critical-path length. Since the span is the theoretically fastest time the dag could be executed on a computer with an infinite number of processors (assuming no overheads for communication, scheduling, etc.), we denote it by T_∞. Like work, span also provides a bound on P-processor execution time:

$$T_P \geq T_\infty \ . \tag{2}$$

This ***Span Law*** arises for the simple reason that a finite number of processors cannot outperform an infinite number of processors, because the infinite-processor machine could just ignore all but P of its processors and mimic a P-processor machine exactly.

Parallelism

We define ***parallelism*** as the ratio of work to span, or T_1/T_∞. Parallelism can be viewed as the average amount of work along each step of the critical path. Moreover, perfect linear speedup cannot be obtained for any number of processors greater than the parallelism T_1/T_∞. To see why, suppose that $P > T_1/T_\infty$, in which case the Span Law (2) implies that the speedup satisfies $T_1/T_P \leq T_1/T_\infty < P$. Since the speedup is strictly less than P, it cannot be perfect linear speedup. Another way to see that the parallelism bounds the speedup is to observe that, in the best case, the work is distributed evenly along the critical path, in which case the amount of work at each step is the parallelism. But, if the parallelism is less than P, there isn't enough work to keep P processors busy at every step.

As an example, the parallelism of the dag in Figure 3 is $18/9 = 2$. Thus, there is little point in executing it with more than 2 processors, since additional processors surely will be starved for work.

In general, one does not need to estimate parallelism particularly accurately to diagnose scalability problems. All that is necessary is that the parallelism exceed the actual number of processors by a reasonable margin. Thus, the measures of work and span need not be particularly precise. A binary order of magnitude is usually more than sufficient. Cilkview takes advantage of this looseness in measurement by performing simple instruction-counting for work and span, rather than attempting to use high-resolution timers.

Upper bounds on speedup

The Work and Span Laws engender two important upper bounds on speedup. The Work Law implies that the speedup on P processors can be at most P:

$$T_1/T_P \leq P\ . \qquad (3)$$

The Span Law dictates that speedup cannot exceed parallelism:

$$T_1/T_p \leq T_1/T_\infty. \qquad (4)$$

In Figure 2, the upper bound on speedup provided by the Work Law corresponds to the line of slope 1. The upper bound provided by the Span Law corresponds to the horizontal line at 23.07.

rinstru

3. THE BURDENED-DAG MODEL

The dag model in Section 2 provides upper bounds on the best possible speedup of a multithreaded application based on the work and span. Actual speedup is influenced not only by these intrinsic characteristics, but also by the performance of the scheduling algorithm and the cost of migrating tasks to load-balance the computation across processor cores. This section discusses prior work on scheduling bounds and proposes a new model, called "burdened dags," for incorporating migration costs.

Work-stealing scheduling

Although optimal multiprocessor scheduling is NP-complete [23], Cilk++'s runtime system employs a "work-stealing" scheduler [5,20] which achieves provably tight asymptotic bounds. In theory, an application with sufficient parallelism can rely on the Cilk++ runtime system to dynamically and automatically exploit an arbitrary number of available processor cores near optimally.

Cilk++'s work-stealing scheduler operates as follows. When the runtime system starts up, it allocates as many system threads, called ***workers***, as there are processors (although the programmer can override this default decision). In the common case, each worker's stack operates just as in C++. When a subroutine is spawned, the subroutine's activation frame containing its local variables is pushed onto the bottom of the stack. The worker begins work on the child (spawned) subroutine. When the child returns to its parent, the parent's activation frame is popped off the bottom of the stack. Since Cilk++ operates just like C++ in the common case, ordinary execution imposes little overhead.

When a worker runs out of work, however, it becomes a ***thief*** and "steals" the top (oldest) frame from another ***victim*** worker's stack. Thus, the stack is in fact a double-ended queue, or ***deque***, with the worker operating on the bottom and thieves stealing from the top. This strategy has the great advantage that all communication and synchronization is incurred only when a worker runs out of work. If an application exhibits sufficient parallelism, one can prove mathematically [5] that stealing is infrequent, and thus the overheads of communication and synchronization to effect a steal is negligible. Specifically, the Cilk++ randomized work-stealing scheduler can execute an application with T_1 work and T_∞ span on P processors in expected time

$$T_P \leq T_1/P + \delta T_\infty\ , \qquad (5)$$

```
// Snippet A
cilk_for (int i=0; i<2; ++i) {
    for (int j=0; j<n; ++j) {
        f(i,j);
    }
}

// Snippet B
for (int j=0; j<n; ++j) {
    cilk_for (int i=0; i<2; ++i) {
        f(i,j);
    }
}
```

Figure 4: An example of how migration overhead can affect performance.

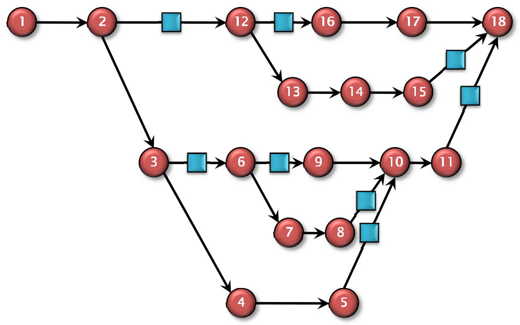


Figure 5: The burdened dag of the computation from Figure 3. Squares indicate the burdens on continuation and return edges and represent potential migration overhead.

where δ is a constant called the ***span coefficient***. (We omit the notation for expectation for simplicity.)

Inequality (5) can be interpreted as follows. If the parallelism T_1/T_∞ exceeds the number P of processors by a sufficient margin, the bound guarantees near-perfect linear speedup. To see why, assume that $T_1/T_\infty \gg P$. Equivalently, we have $T_\infty \ll T_1/P$. Thus, in Inequality (5), the T_1/P term dominates the δT_∞ term, and thus the running time is $T_P \approx T_1/P$, leading to a speedup of $T_1/T_P \approx P$.

The key weakness in this argument stems from what is meant by "sufficient." The proof of the bound assumes that the scheduling overheads are at most a constant. The larger the constant hidden by the big-O, however, the greater the factor by which the parallelism T_1/T_∞ must exceed the number P of processors to guarantee near-perfect linear speedup. In particular, to estimate speedup accurately for all P, asymptotics are not good enough, especially when parallelism is modest. Moreover, naively using an upper bound on the constant hidden by the big-O yields poor scalability estimates.

Migration overhead

As an example of how migration overhead can affect performance, consider the two snippets in Figure 4. In this code, `f(i,j)` is some small function. The work of each of the two snippets involves $2n$ function calls. The span of each snippet involves n function calls, since the `for` loops in n run serially. Thus, the parallelism of each snippet is about 2. Nevertheless, Snippet A generally outperforms Snippet B by a wide margin. The reason is that Snippet A only requires one steal in order to realize the parallelism, whereas Snippet B requires one steal for each of the n iterations of the `for` loop. Thus, Snippet B incurs substantial migration overhead.

Burdened dags

The migration cost, or ***burden***, for a steal includes the explicit costs of bookkeeping to set up the context to run the stolen task and the implicit costs of cache misses in order to migrate the stolen task's working set. Although the scheduling cost modeled by the burden generally depends upon what is stolen, we have found that a fixed cost suffices in practice. Cilkview assumes an upper-bound for burden of 15,000 instructions.

The ***burdened-dag model*** augments the dag model by including the burden on each continuation and return edge of the dag, which is akin to assuming that every possible place that a task could be migrated, it is. Figure 5 shows the burdened dag for the computation from Figure 3. Cilkview computes the ***burdened span*** by finding the longest path in the burdened dag.

The following theorem shows how migration costs can be incorporated into the bound from Inequality (5).

THEOREM 1. *Let T_1 be the work of an application program, and let $\widehat{T}_\infty$ be its burdened span. Then, a work-stealing scheduler running on P processors can execute the application in expected time*

$$T_P \leq T_1/P + 2\delta\widehat{T}_\infty ,$$

where δ is the span coefficient.

PROOF. Let G be the ordinary dag arising from the P-processor execution, and let $\widehat{G}$ be the corresponding burdened dag. Consider the more-explicit dag G' that arises during the P-processor execution, where all migration overheads incurred explicitly by the scheduler or implicitly by the application are accounted for with additional vertices and edges. Let T_1' and T_∞' be the work and span, respectively, of G'. The accounting argument in [5] can be used to show that for any constant $\varepsilon > 0$, we have

$$T_P \leq T_1'/P + \delta T_\infty' + O(\log(1/\varepsilon))$$

with probability at least $1-\varepsilon$. Moreover, the number of steals is at most $\delta P T_\infty' + O(P\log(1/\varepsilon))$.

Let us now look at work. Every path in G' corresponds to a path in the burdened dag $\widehat{G}$, except that G' omits overheads for continuations that are not stolen. Thus, we have $T_\infty' \leq \widehat{T}_\infty$. Since we only have a migration overhead when a continuation is stolen, the total amount of overhead in G' is at most $\delta P T_\infty' + O(P\log(1/\varepsilon))$, from which it follows that $T_1' \leq T_1 + \delta P T_\infty' + O(P\log(1/\varepsilon)) \leq T_1 + \delta P\widehat{T}_\infty + O(P\log(1/\varepsilon))$. Consequently, we have

$$\begin{aligned} T_p &\leq T_1'/P + \delta T_\infty' + O(\log(1/\varepsilon)) \\ &\leq (T_1 + \delta P\widehat{T}_\infty + O(P\log(1/\varepsilon)))/P + \delta\widehat{T}_\infty + O(\log(1/\varepsilon)) \\ &\leq T_1/P + 2\delta\widehat{T}_\infty + O(\log(1/\varepsilon)) \end{aligned}$$

with probability at least $1-\varepsilon$. Therefore, the expected time is bounded by $T_P \leq T_1/P + 2\delta\widehat{T}_\infty$. $\square$

COROLLARY 2. *Let T_1 be the work of an application program, $\widehat{T}_\infty$ its burdened span, and δ the span coefficient. Then, a work-stealing scheduler running on P processors achieves speedup at least*

$$\frac{T_1}{T_P} \geq \frac{T_1}{T_1/P + 2\delta\widehat{T}_\infty} . \qquad (6)$$

in expectation.

Cilkview applies Corollary 2 to compute an estimated lower bound on speedup. Our experimental results show that a typical value of span coefficient δ has range $0.8-1.0$. Cilkview uses $\delta = 0.85$, and a corresponding estimated speedup lower bound $T_P \leq T_1/P + 1.7\widehat{T}_\infty$. An example is the lower curve in Figure 2. In addition, it computes the ***burdened parallelism*** as $T_1/\widehat{T}_\infty$.

```
1) Parallelism Profile
   Work:                     5,570,609,776 instructions
   Span:                     261,374,874 instructions
   Burdened span:            262,078,779 instructions
   Parallelism:              21.31
   Burdened parallelism:     21.26
   Spawns:                   8,518,398
   Syncs:                    8,518,398
   Average maximal strand:   218

2) Speedup Estimate
   2 processors:             1.85 – 2.00
   4 processors:             3.23 – 4.00
   8 processors:             5.13 – 8.00
   16 processors:            7.27 – 16.00
   32 processors:            9.20 – 21.31
```

Figure 6: Textual output for the quicksort program from Figure 1.

4. TEXTUAL OUTPUT AND BENCHMARKING

Cilkview employs the dag and burdened dag models described in Sections 2 and 3, respectively, to analyze scalability. In addition to graphical output, which was illustrated in Figure 2 for the quicksort program, Cilkview also provides a textual report with explicit numbers, as is illustrated in Figure 6 for quicksort. The textual report is broken into two parts, the Parallelism Profile and the Speedup Estimate.

The Parallelism Profile displays the statistics collected during the run of the program. The statistics include work, span, parallelism, burdened span, and burdened parallelism, whose meaning was described in Sections 2 and 3. The other three statistics shown are the following:

- Spawns — the number of spawns encountered during the run.
- Syncs — the number of syncs encountered during the run.
- Average maximal strand — the work divided by 1 plus twice the number of spawns plus the number of syncs.

When the average maximal strand is small (less than 500), the overhead from spawning, which is generally negligible, may be noticeable. In the quicksort code, for example, the value of 218 indicates that the parallel recursion might be terminated early and replaced by ordinary function calls. Although this change might lower the parallelism slightly, it would lead to a faster serial execution due to less spawn overhead.

The Speedup Estimate section profiles the estimated speedups on varying numbers of processor cores. The estimates are displayed as ranges with lower and upper bounds. The upper bound is the minimum of number of cores and parallelism, based on Inequalities (3) and (4). The lower bound is based on Inequality (6).

Whereas Cilkview generates its textual report from an application run using the Pin instrumentation framework, it also supports direct benchmarking of the application, automatically running it across a range of processor counts. Each benchmark result represents the actual speedup of the run on the given number of processor cores. To use Cilkview for benchmarking, the following practices are recommended:

- Make the system as quiet as possible by killing unnecessary processes so that measurements are as exact as possible.
- Turn off hyperthreading in the BIOS so that per-processor measurements are meaningful.
- Turn off power saving in the BIOS, which makes clock speeds unpredictable.
- Run trials multiple times to moderate effects due to specific cores on which the OS schedules the worker threads.

```
void stencil_kernel (int t, int x,
    int y, int z) {
  int s = z * NXY + y * NX + x;
  float *A_cur = &A[t & 1][s];
  float *A_next = &A[(t + 1) & 1][s];
  float div = c0 * A_cur[0]
    + c1 * ((A_cur[1] + A_cur[-1])
    + (A_cur[NX] + A_cur[-NX])
    + (A_cur[NXY] + A_cur[-NXY]))
    + c2 * ((A_cur[2] + A_cur[-2])
    + (A_cur[NX2] + A_cur[-NX2])
    + (A_cur[NXY2] + A_cur[-NXY2]))
    + c3 * ((A_cur[3] + A_cur[-3])
    + (A_cur[NX3] + A_cur[-NX3])
    + (A_cur[NXY3] + A_cur[-NXY3]))
    + c4 * ((A_cur[4] + A_cur[-4])
    + (A_cur[NX4] + A_cur[-NX4])
    + (A_cur[NXY4] + A_cur[-NXY4]));
  A_next[0] = 2 * A_cur[0] - A_next[0]
    + vsq[s] * div;
}

void stencil_loop (int t0, int t1,
     int x0, int x1, int y0, int y1,
     int z0, int z1){
  for(int t = t0; t < t1; ++t) {
    for(int z = z0; z < z1; ++z) {
      for(int y = y0; y < y1; ++y) {
        cilk_for(int x = x0; x < x1; ++x) {
          // stencil computation kernel
          stencil_kernel(t, x, y, z);
} } } } }

void cilk_main(int argc, char** argv) {
  ...
  // Compute wave equation for 20 time steps
  // on a 500x500x500 array
  stencil_loop(1, 20, 4, 495, 4, 495, 4, 495);
  ...
}
```

Figure 7: A naive implementation of the stencil computation using `cilk_for`.

- If necessary, pin the workers to specific cores to enhance repeatability.

Programmer Interface

The Cilkview API allows users to control which portions of their Cilk++ application program are reported for both scalability analysis and benchmarking. The API provides the `start()` and `stop()` methods for a `cilkview` object to indicate the section to report. In addition, the API provides a `dump(const char *)` method that can label the reported section. Consequently, Cilkview can generate reports for multiple sections of code in an application by reusing the same `cilkview` object. In addition, multiple `cilkview` objects can be created if the sections overlap.

5. PERFORMANCE DIAGNOSIS

This section presents an example of how Cilkview can be used to diagnose the performance problems of a Cilk++ program that computes a 25-point 3D stencil. We start with an analysis of the program as written, and then we modify it step by step to improve its performance, each time using Cilkview to understand how the program can be improved.

Figure 7 shows a "stencil" program to solve the wave equation

$$\frac{\partial^2 u}{\partial t^2} = c\nabla^2 u$$

in parallel using the finite-difference method described by Courant, Friedrichs, and Lewy [14]. For a given 3D coordinate (x,y,z) at time t, the code derives the value of $A[x,y,z]$ at time t using the value of $A[x,y,z]$ and the 24 neighboring values $\{A[x,y,z'] : 1 \le |z-z'| \le 4\} \cup \{A[x,y',z] : 1 \le |y-y'| \le 4\} \cup \{A[x',y,z] : 1 \le |x-x'| \le 4\}$ at time $t-1$, where there are 8 neighbors for each dimension. The multiplicative coefficients

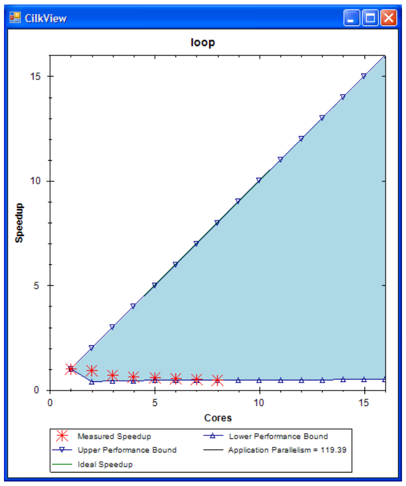


Figure 8: Cilkview's output for the code from Figure 7.

$\{c_i : 0 \le i \le 4\}$ and the phase velocities $vsq[x,y,z]$ are constants. Since intermediate values in the computation can be thrown away, the code uses only two 3D arrays of coordinates, one for even values of t and the other for odd values.

We benchmarked this application on an 8-core Intel® Core^{TM} i7 machine with two Xeon® E5520 2.27-GHz quad-core chips (hyperthreading disabled), 6 GB of DRAM, a shared 8-MB L3-cache, and private L2- and L1-caches with 256 KB and 32 KB, respectively. This machine is the same as the one we used to benchmark the quicksort program from Section 1.

Performance problem — low burdened parallelism: Running the program on 2 or more processors takes longer than its serial execution time. Cilkview's scalability estimation and benchmarking results are shown in Figure 8. The parallelism of the program is 119.39, but its burdened parallelism is only 0.87. The estimated speedup lower bound is far less than the upper bound. In fact, it is less than 1.0, portending a possible slowdown in parallel execution. The trial data confirms this hypothesis.

Diagnosis: The low burdened parallelism and the large gap between the upper and lower speedup bounds indicates that the overhead of load balancing from running the code in parallel could negate any benefit obtained from parallel execution. The amount of stolen work is too small to justify the overhead of load balancing. At the source-code level, the problem is that the granularity of spawned tasks is too small. Observe that in the source code from Figure 8, `cilk_for` is used in the inner loop, and hence the amount of work per iteration is tiny, resulting in small tasks and a low burdened parallelism.

Solution: Generally, it is better to parallelize outer loops than inner loops. We move the `cilk_for` from the innermost loop to the first nested loop, as shown in Figure 9. This change significantly increases the granularity of spawned tasks.

Performance problem — subrange benchmark results: Cilkview now reports good parallelism and burdened parallelism for the improved program, as shown in Figure 10. The estimated lower bound on speedup is close to the upper bound of perfect linear speedup, which indicates that this program has sufficient parallelism and good granularity for its spawned tasks. As seen in the figure, however, the benchmark data produced by Cilkview for an 8-core machine shows that the actual speedup does not scale linearly. The performance is well beneath the range of expectation.

```
void stencil_revised_loop (int t0, int t1,
     int x0, int x1, int y0, int y1,
     int z0, int z1){
  for(int t = t0; t < t1; ++t) {
    cilk_for(int z = z0; z < z1; ++z) {
      for(int y = y0; y < y1; ++y) {
        for(int x = x0; x < x1; ++x) {
          // stencil computation kernel
          stencil_kernel(t, x, y, z);
} } } } }
```

Figure 9: A revised implementation of the stencil in which an outer loop has been parallelized using `cilk_for`, rather than the inner loop.

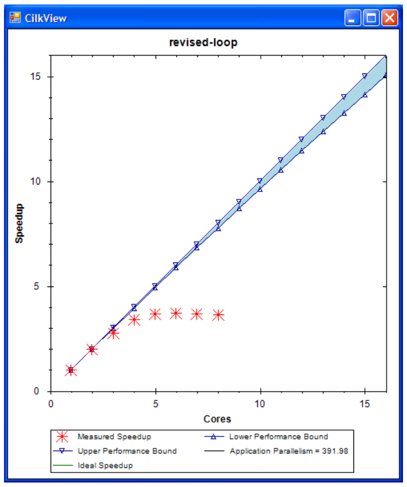


Figure 10: Cilkview's output for the code from Figure 9.

Diagnosis: Since Cilkview accurately diagnoses problems involving inadequate parallelism and insufficient grain size, we can be confident that neither of these problems is the cause of the poor performance. The programmer can now look to other common sources of meager performance, such as insufficient memory bandwidth, false sharing, and lock contention.

The algorithm in Figure 9 applies the kernel to all space-time points at time t before computing any point at time $t+1$. Modern computers operate with a cache-memory hierarchy. If the number of 3D points computed during a time step exceed the size of a given cache by a sufficient margin, the application incurs a number of cache misses proportional to the compute time. In this case, one time step of the computation involves 125 M 3D points, which exceeds even the L3-cache. Looking at Figure 10 more closely, we can see that the speedup levels off around 4–5 cores. At this point, the processor cores' demand for memory bandwidth saturates the memory subsystem. Because this stencil implementation is a memory-bandwidth hog, the performance falls below the predicted range.

How can one determine that a multithreaded program is limited by memory bandwidth? One indication is an unusually high value of CPI (cycles per instruction), but there can be other reasons for a high CPI. A simple test that often diagnoses memory-bandwidth problems is to run two benchmarks:

1. On a P-core machine, run P simultaneous serial executions.
2. Run 1 serial execution of the program alone on 1 core.

If the execution time in Case 1 is significantly larger than in Case 2, it is likely that the demand for memory bandwidth is high enough to be a limiting factor on program scalability.

Solution: Divide-and-conquer recursion, properly coarsened at the leaves so as not to incur too much function-call overhead, generally taxes memory bandwidth less than iterative methods that stride

```
void co_cilk(int t0, int t1, int x0, int dx0, int
    x1, int dx1, int y0, int dy0, int y1, int dy1,
     int z0, int dz0, int z1, int dz1 ) {
  int dt = t1 - t0, dx = x1 - x0;
  int dy = y1 - y0, dz = z1 - z0;

  if (dx >= dx_threshold && dx >= dy && dx >= dz &&
       dt >= 1 && dx >= 2 * ds * dt * NPIECES) {
    //divide and conquer along x direction
    int chunk = dx / NPIECES;
    for (i = 0; i < NPIECES - 1; ++i) {
      cilk_spawn co_cilk(t0, t1, x0+i*chunk, ds, x0
           + (i+1) * chunk, -ds, y0, dy0, y1, dy1,
           z0, dz0, z1, dz1);
    }
    cilk_spawn co_cilk(t0, t1, x0+i*chunk, ds, x1,
         -ds, y0, dy0, y1, dy1, z0, dz0, z1, dz1);
    cilk_sync;

    cilk_spawn co_cilk(t0, t1, x0, dx0, x0, ds, y0,
          dy0, y1, dy1, z0, dz0, z1, dz1);
    for (i = 1; i < NPIECES; ++i) {
      cilk_spawn co_cilk(t0, t1, x0+i*chunk, -ds,
           x0 + i * chunk, ds, y0, dy0, y1, dy1, z0
           , dz0, z1, dz1);
    }
    cilk_spawn co_cilk(t0, t1, x1, -ds, x1, dx1, y0
         , dy0, y1, dy1, z0, dz0, z1, dz1);
  } else if (dy >= dyz_threshold && dy >= dz && dt
       >= 1 && dy>=2*ds*dt*NPIECES) {
    //similarly divide and conquer along y.
    ......
  } else if (dz >= dyz_threshold && dt >= 1  && dz
       >= 2 * ds * dt * NPIECES) {
    //similarly divide and conquer along z.
    ......
  }  else if (dt > dt_threshold) {
    int hdt = dt / 2;
    //decompose over time t direction
    co_cilk(t0, t0 + hdt, x0, dx0, x1, dx1, y0, dy0
         , y1, dy1, z0, dz0, z1, dz1);
    co_cilk(t0 + hdt, t1, x0+dx0*hdt, dx0, x1+dx1*
         hdt, dx1, y0+dy0*hdt, dy0, y1+dy1*hdt, dy1
         , z0+dz0*hdt, dz0, z1+dz1*hdt, dz1);
  } else {
    //compute base case
    co_basecase_nv(t0, t1, x0, dx0, x1, dx1, y0,
         dy0, y1, dy1, z0, dz0, z1, dz1);
  }
}
```

Figure 11: Cache-oblivious code for the stencil computation.

through arrays multiple times. Frigo and Strumpen [21, 22] devised a so-called "cache-oblivious" [19] stencil algorithm to solve the memory-bandwidth problem and preserve parallelism. The algorithm advances time nonuniformly by strategically decomposing the space-time points recursively. On an ideal machine, when a subarray fits into a cache at a given level in the hierarchy, no cache misses are incurred except for those needed to bring the subarray into the cache.

The revised cache-oblivious implementation is shown in Figure 11. Its scalability estimation and trial results are shown in Figure 12. As the figure shows, Cilkview reports good ideal and burdened parallelism. Since access to memory is no longer a bottleneck, the program achieves linear speedup.

6. IMPLEMENTATION

Cilkview collects a program's parallelism information during a serial execution of the program code running under the Pin [35] dynamic instrumentation framework. Since Cilkview operates on the optimized executable binary produced by the compiler, no recompilation of the code is necessary. Metadata in Cilk++ binaries allows Cilkview to identify the parallel control constructs in the executing application precisely and to collect statistics efficiently. When the application executes in a normal production environment, the metadata imposes no overhead on performance.

In our implementation of Cilkview, we decided to eschew direct timing measurements, which are generally not repeatable, and

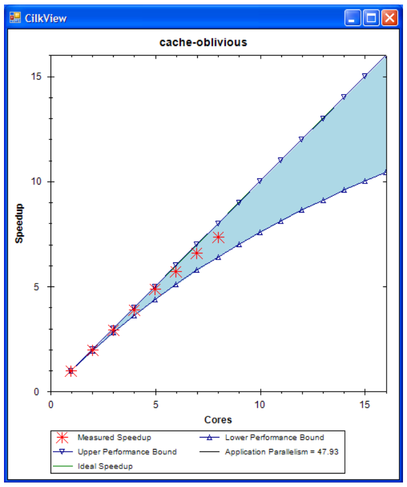


Figure 12: Cilkview's output for the code from Figure 11.

use instruction counts as a surrogate for time. Although instruction counts are a coarse measure of performance, the parallelism measurement need not be surgically precise: generally, parallelism estimates accurate to within a binary order of magnitude suffice to diagnose problems of inadequate parallelism. If an application with parallelism approximately 100 is running on 8 cores, one does not care whether the parallelism is really 95.78, 103.44, or even 1000. What matters is that the parallelism is significantly larger than the number of processors. Cilkview answers the simple question, "Does my application suffer from inadequate parallelism?" and, with the burdened analysis, "Does my application suffer from too fine-grained task creation?" Additional digits of precision rarely make a significant difference in the answers to these questions.

This section first reviews Cilkview's instrumentation strategy. Then, we present Cilkview's algorithm to compute performance measurements of a Cilk++ program according to logical dependencies of tasks. Finally, we discuss some sources of inaccuracy in Cilkview's measurements.

Instrumentation strategy

The design principle behind Cilkview's instrumentation strategy is to encode the logical parallel dependencies among tasks as part of an executable binary without compromising the performance of the production code. Cilkview relies on an enhanced version of the x86 binary executable format that provides metadata relevant to multithreaded execution. As part of the ordinary compilation process, the Cilk++ compiler embeds this metadata into the executable using a standard ***multithreaded executable format*** (***MEF***).

Cilkview operates directly on the optimized MEF binary executable distributed to end users, rather than on a "debug" version, thereby avoiding the problem of measuring an application that has been perturbed by instrumentation overhead. The MEF binary includes metadata to inform Cilkview of where Cilk++'s high-level parallel control constructs appear in the executable segment. In particular, Cilkview requires metadata to mark the beginning and end of each spawned function, the beginning and end of each called function, and the location of each `cilk_sync`.

MEF metadata is stored in a platform-specific nonloadable section of the executable. Although this section resides in the disk image, it is not loaded into memory during normal execution. Thus, when an application executes normally, the metadata introduces no overhead. When the application executes under Cilkview, however, Cilkview reads the metadata and instruments the corresponding addresses in the executable segment to collect measurements. The performance of Cilkview is typically 2–10 times normal execution time on one core. Users report that this slowdown is acceptable for the insights on scalability that Cilkview provides.

```
Function $P$ spawns function $C$:
    $C.work = P.work$
    $C.span = P.span$
    $C.bspan = P.bspan$
    $C.cwork = 0$
    $C.cspan = -\infty$
    $C.cbspan = -\infty$
    $P.work = 0$
    $P.bspan \mathrel{+}= \text{BURDEN}$

Function $C$ returns from spawn with parent $P$:
    $P.cwork \mathrel{+}= C.work$
    $P.cspan = \max\{P.cspan, C.span\}$
    $P.cbspan = \text{BURDEN} + \max\{P.cbspan, C.bspan\}$

cilk_sync in function $P$:
    $P.work \mathrel{+}= P.cwork$
    $P.cwork = 0$
    $P.span = \max\{P.span, P.cspan\}$
    $P.cspan = -\infty$
    $P.bspan = \max\{P.bspan, P.cbspan\}$
    $P.cbspan = -\infty$
```

Figure 13: The Cilkview algorithm.

Algorithm

With the support of the instrumentation framework, Cilkview computes three basic performance measures: work, span, and burdened span. As Cilkview executes the Cilk++ program serially under Pin, it associates six state variables — *work*, *span*, *bspan*, *cwork*, *cspan*, and *cbspan* — with each function, whether spawned or called. The variables *work*, *span* and *bspan* store the accumulated work, span, and burdened span, respectively, from the beginning of the program to the current point of execution. The variables *cwork*, *cspan*, and *cbspan* store the accumulated work and span, respectively, of child functions that run logically in parallel with the current function.

Conceptually, for each instruction, Cilkview increments the variables[2] *work*, *span*, and *bspan* of the current function. In addition, it executes certain actions at spawns and syncs, as shown in Figure 13. The constant BURDEN represents the task-migration overhead during a successful steal. This cost, which can be visualized as burdened nodes in Figure 5, is added to the spawning function's *bspan* at a `cilk_spawn`. At a `cilk_sync`, Cilkview incorporates the work and span of the parallel children into the function's own state variables.

Cilkview models the task-migration cost BURDEN as a constant. This cost could be evaluated with a more precise model that varies the estimated BURDEN value depending on the structure of the program, but since we only need rough measures of work, span, and burdened span, an upper-bound value determined by experimentation suffices to diagnose most scalability problems. Cilkview assumes a burden of 15,000 instructions.

Having gathered the three statistics: *work*, *span*, and *bspan*, Cilkview uses Inequalities (3), (4), and (6) to estimate the bounds of the instrumented program's speedup.

[2]Actually, Cilkview counts instructions on a basic-block basis, which is more efficient.

Sources of inaccuracy

When a program is deterministic, a program's work and span as determined by a serial run equal those of parallel runs, modulo scheduling overheads, caching effects, and the like. Consequently, Cilkview can collect the *work* and *span* information during a serial execution and infer that the work and span will not change substantially during a parallel run, except for these overheads, which can be shown to be minimal if the application contains sufficient parallelism [5].

When a program is nondeterministic, however, a parallel run may give different work and span from the serial one. Cilk++ allows some forms of nondeterminism to be encapsulated in ***reducer hyperobjects*** [18]. For a reducer hyperobject, the Cilk++ runtime manages local "views" of the hyperobject locally on the workers and combines them automatically with a ***reduce*** operation when subcomputations join at a sync. When and where the reduce operations occur is scheduler dependent, however, and this nondeterminism can affect the work and span. In particular, reduce operations never occur during a serial execution, and since Cilkview obtains its measurements for work and span during a serial run of the program, it never sees reducer overheads.

Reducers only occur because work was stolen, however. Since steals are infrequent for programs with sufficient parallelism [5], reduce operations seldom occur and tend to impede performance minimally. When implementing Cilkview, we therefore felt comfortable using the serial work and span without attempting to estimate reducer overheads, even though the measurements may suffer from some inaccuracies for programs with inadequate parallelism.

Another source of potential inaccuracy arises from the implementation of `cilk_for`. A `cilk_for` loop is implemented as a parallel divide-and-conquer tree over the iterations of the loop. Rather than recursing down to a single iteration, however, which would result in the overhead of one spawn per iteration, the Cilk++ runtime system automatically coarsens the recursion. That is, when the range of iterations drops below some ***grain size***, it reverts to an ordinary serial `for` loop. The default grain size is computed dynamically according to the number of loop iterations and the number of cores. Different grain sizes affect the work and span of the program. Cilkview performs its measurements assuming that the default grain size is 1 unless the user specifies the grain size explicitly. Thus, it may overestimate the true work due to spawning overhead, while assuming as much parallelism as possible. As a result, Cilkview's estimate of parallelism may indicate somewhat more parallelism than actually exists, but generally not enough to influence diagnosis of performance problems.

7. BENCHMARK APPLICATIONS

This section illustrates Cilkview's scalability analysis on six benchmark applications. These applications were selected to cover a wide variety of multithreaded programs across different domains. The speedup estimation of each application was produced by the instrumented execution under Cilkview, and we used Cilkview to benchmark runs of the application on our 8-core Intel®Core^{TM} i7 processor using 1 to 8 workers. Figure 14 shows the graphical outputs of Cilkview on these benchmark applications. We now briefly describe each application and its Cilkview output.

Bzip2. Bzip2 [46] is a popular package for file compression. The Cilk++ implementation [10] of bzip2 was coded by John F. Carr of Intel Corporation. It compresses several blocks of data in parallel and applies a stream "reducer" [18] to ensure that the parallel compressed data is written to the output file in the correct order. As shown in Figure 14(a), this application exhibits a parallelism of 12.25 when run on a data file of 28 MB, and it obtained about 7 times speedup on the 8-core machine. The performance fits within the estimated scalability range.

Murphi. This application [16] is a finite-state machine verification tool used widely in the design of cache-coherence algorithms, link-level protocols, and executable memory-model analyses. Murphi verifies a protocol by traversing the entire state space in what is essentially a graph-traversal process. The Cilk++ implementation [27] parallelizes two of the search algorithms in the standard Murphi 3.1 package. Parallelizing the depth-first search is straightforward, and breadth-first search was parallelized using the PBFS algorithm [34]. Figure 14(b) shows the Cilkview output for the breadth-first implementation on adash, an example program included in the Murphi package.

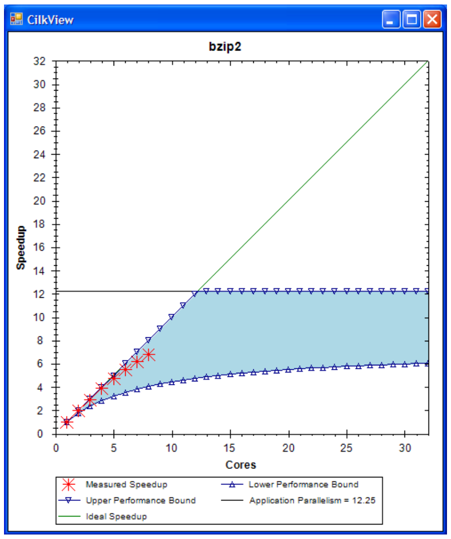
Collision detection. This application was adapted from an industrial CAD application. The application reads two data files, each of which comprises a model of an assembly of three-dimensional parts. It then compares the models and outputs another data file listing all the parts from one assembly that collide with parts from the other assembly. Each model is stored in a ternary tree, and the collision-detection search performs recursive traversals of these ternary trees. The Cilk++ implementation searches the tree branches in parallel. It uses a list "reducer" [18] to collect the colliding parts. As shown in Figure 14(c), Cilkview predicts a good speedup on an input problem of two 9.9 MB files, and the implementation performed well.

Memcpy. This utility copies a block of memory from a source location to a destination location. The Cilk++ parallel memcpy replaces the `for` loop of the serial implementation with a `cilk_for` loop to enable parallelism. The Cilkview output in Figure 14(d) for copying a 477 MB block indicates linear speedup for the estimated lower and upper bounds. The speedups for actual runs were smaller than predicted, since memory bandwidth is a major bottleneck.

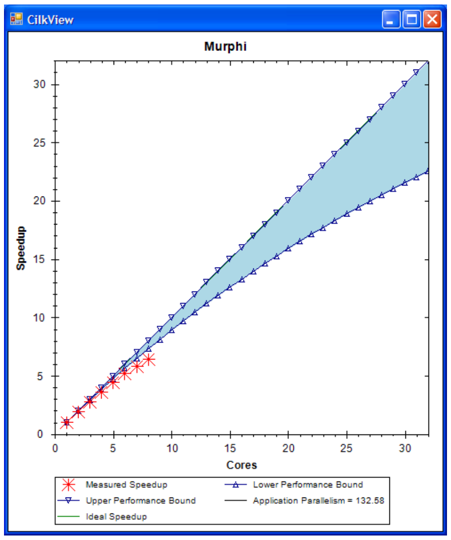
Min poset. Computing the minimal elements of a partially ordered finite set (poset) is a fundamental problem with numerous applications in combinatorics and algebraic geometry. One of them is optimizing the size of dags encoding polynomial expressions. In this case, given the terms of a polynomial, one is interested in determining the monomials that make up the set of minimal elements of a poset generated by the terms of the polynomial using the divisibility relation. The Cilk++ implementation of a parallel divide-and-conquer sieving algorithm for min poset was coded by Yuzhen Xie of the University of Western Ontario. The application was run on the poset generated by a polynomial with 14,869 terms, each with up to 28 variables, and produced 14 minimal elements. As shown in Figure 14(e), it obtained near-perfect linear speedup, as predicted by Cilkview.

Bivariate polynomial multiplication. The Basic Polynomial Algebra Subroutines (BPAS) package [36, 37] provides a complete set of parallel routines in Cilk++ for FFT-based dense polynomial arithmetic over finite fields, such as polynomial multiplication and normal-form computations. Polynomial arithmetic sits at the core of every computer-algebra system, including Maple and Mathematica, and the efficiency of polynomial computations greatly impacts the responsiveness of these software packages. A Cilk++ version of polynomial multiplication that uses the truncated Fourier transform was coded by Yuzhen Xie and Marc Moreno Maza of the University of Western Ontario. The application input was two bivariate polynomials all of whose partial degrees are equal to 1023. As shown in Figure 14(f), Cilkview predicts a linear speedup, which the benchmarking results confirm.

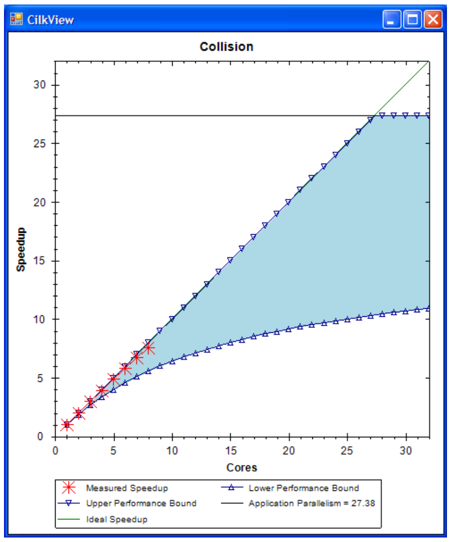
8. RELATED WORK



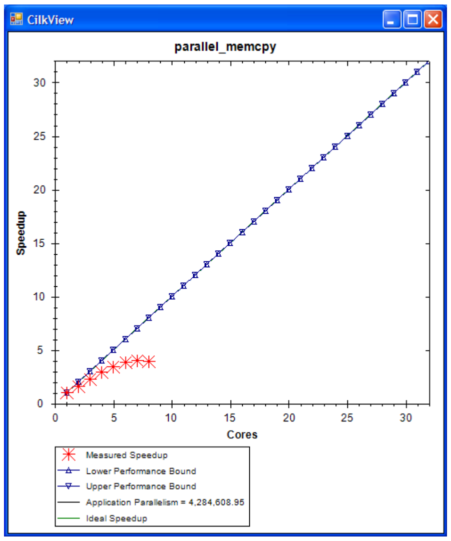
(a) Bzip2.



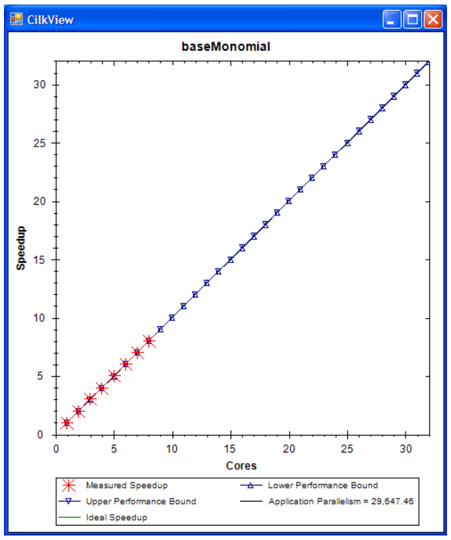
(b) Murphi.



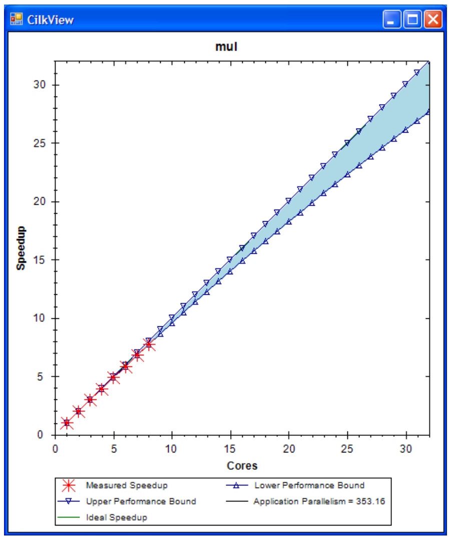
(c) Collision detection.



(d) Memcpy.



(e) Min poset.



(f) Bivariate polynomial multiplication.

Figure 14: Cilkview graphical output for six benchmark applications.

This section overviews tools for measuring parallel application performance. We survey the tools and discuss strategies for instrumenting performance.

Much work [49–51] has been devoted to diagnosing communication and synchronization problems in message-passing systems such as MPI. Tools for shared-memory systems [3,24] tend to focus on an explicit threading (e.g., Pthreads, WinAPI threads, etc.). MPI and explicit-threading programs statically partition their work into threads and then map each thread to a core. Since these models do not perform dynamic load balancing, the performance tools focus on analyzing communication and synchronization, rather than on scalability *per se*.

Dynamic multithreaded concurrency platforms — e.g., Cilk-5 [20], Cilk++ [33], Fortress [2], Hood [6], Java Fork/Join Framework [31], OpenMP 3.0 [42], Task Parallel Library (TPL) [32], Intel® Threading Building Blocks (TBB) [45], and X10 [12] — have engendered their share of tools. Most, including Thread Profiler (in VTune [44]), Parallel Amplifier (in Parallel Studio [29]), and OMPtrace [11], gather low-level information specific to how the computation was scheduled and which cannot be easily adapted to forecast scalability. An exception is Cilk-5, which provided an option for computing work and span during an instrumented run. Unlike Cilkview, Cilk-5 instrumentation runs required recompilation, were based on actual time as measured by hardware cycle counters, and introduced overhead into the instrumented binary.

HPCToolkit [47, 48] is a profiling tool for multithreaded programming models ranging from explicit threading to dynamic multithreading. Its metrics are parallel idleness and overhead, which are qualitatively analogous to parallelism and burden. High idleness of a code segment means its parallelism is low and concurrency should be increased. High overhead means high burden, and concurrency should be decreased. Idleness and overhead are insufficient quantitatively to compute the logical concurrency of the program such as span and parallelism, which are critical for scalability prediction, but they are useful metrics for tuning.

Performance tools differ with respect to their strategies for instrumenting applications. Tools such as OPARI [40], Pablo [43], and Tau [41] add instrumentation to source code during the build process. Since source instrumentation can interfere with compiler optimizations, such as inlining and loop transformations, measurements may not accurately reflect the performance of fully optimized code. VTune [44] uses static binary instrumentation to augment application binaries for code profiling. Some tools use instrumented libraries [11, 39, 49]. One major problem with source instrumentation, static binary instrumentation, and instrumentation libraries is that they require source recompilation, binary rewriting, or library relinking, which can be inconvenient for analyzing large production codes.

Dynamic instrumentation supports the analysis of fully optimized binaries by inserting instrumentation in the executing application. Cilkview and various other tools [9, 15, 38] use dynamic instrumentation to collect metrics directly on the optimized binary. This strategy introduces little or no overhead to the program binary in normal runs. During performance-collection runs, however, dynamic instrumentation can dilate total execution time with overhead [48]. For Cilkview, it does not matter, however, because Cilkview counts instructions instead of measuring execution time.

9. CONCLUSION

This section discusses directions for future research concerning how Cilkview's capabilities might be extended.

Most contemporary mainstream multicore processors have limited memory bandwidth. Indeed, a single core of the Intel® Core™ 2, for example, can saturate the available memory bandwidth. Thus, programs such as the standard `daxpy` loop[3], which are bandwidth-limited, do not run significantly faster on multiple cores than on one. The current implementation of Cilkview does not consider the influence of memory bandwidth on a program's execution time. Therefore, for a program like `daxpy` loop or parallel memcpy, where memory-bandwidth is the major performance bottleneck, real speedup of the program running on a multicore processor can deviate from Cilkview's estimation. Section 5 suggested one kind of test to diagnose bandwidth problems based on running concurrent copies of the serial program. It would be interesting to determine whether a tool could be built to diagnose memory-bandwidth problems by modeling caches during an instrumented run.

Some profilers, such as gprof [26], help a programmer analyze the running time of a serial program and diagnose performance bottlenecks without obligating the programmer to insert instrumentation by hand. Profilers typically provide data concerning the time spent in each called function and the function's children. For parallel programs, it would be desirable to have a profiler that produces comparable data for span. By knowing which parts of the code are bottlenecks for parallelism, the programmer can better focus her or his efforts on improving multicore performance. How to obtain and display profile data for span is a good research question. It seems difficult to employ gprof's strategy of sampling the programming counter noninvasively to obtain a good estimate of span, but binary instrumentation may be a reasonable approach. However the profile data is obtained, it remains a research question how best to lay out the data to convey parallelism bottlenecks to the programmer.

After computing a program's span, the programmer may wish to determine how much speedup might be gained by optimizing a function on the critical path. It could be that speeding up a function by just a small amount causes it to fall off the critical path, whereas speeding up another function may reduce the critical path by the full savings in time. Algorithms for this kind of sensitivity analysis are well known if the dag is given in full. Computing a sensitivity analysis as the dag unfolds on the fly seems like an interesting research problem, especially if one wishes to take into account that certain lines of code may appear multiple times on the critical path.

10. ACKNOWLEDGMENTS

Many thanks to our great team members at Cilk Arts — now at Intel Corporation since the acquisition of our little company in July 2009 — and to the many customers who helped us refine the Cilkview tool. In particular, many thanks to Barry Tannenbaum of Intel, who implemented the original graphical output application for Cilkview. Matteo Frigo, formerly of Intel and now of Axis Semiconductor, offered much valuable advice on the implementation of Cilkview. John F. Carr of Intel helped with the benchmarking of applications. We greatly appreciate the constructive comments made by the anonymous reviewers.

11. REFERENCES

[1] A. Alexandrov, S. Bratanov, J. Fedorova, D. Levinthal, I. Lopatin, and D. Ryabtsev. Parallelization made easier with Intel Performance-Tuning Utility. *Intel Technology Journal*, 2007. `http://www.intel.com/technology/itj/2007/v11i4//1-abstract.htm`.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification, Version 1.0*. Sun Microsystems, Inc., 2008.

[3]The `daxpy` routine computes $ax+y$ for scalar a and vectors x and y.

[3] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS Perform. Eval. Rev.*, 18(1):115–125, 1990.

[4] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, 1998.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[6] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[7] R. P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21(2):201–206, 1974.

[8] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT EECS, 2004.

[9] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perf. Comput. Appl.*, 14(4):317–329, 2000.

[10] J. Carr. A parallel `bzip2`. Available from `http://software.intel.com/en-us/articles/a-parallel-bzip2/`, 2009.

[11] J. Caubet, J. Gimenez, J. Labarta, L. D. Rose, and J. S. Vetter. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *WOMPAT*, pp. 53–67, 2001.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pp. 519–538, 2005.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[14] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM J. R&D*, 11(2):215–234, 1967.

[15] L. DeRose, T. Hoover Jr., and J. K. Hollingsworth. The Dynamic Probe Class Library - an infrastructure for developing instrumentation for performance tools. In *IPDPS*, p. 10066b, 2001.

[16] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *ICCD*, pp. 522–525, 1992.

[17] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, 1989.

[18] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, pp. 79–90, 2009.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pp. 285–297, New York, New York, 1999.

[20] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pp. 212–223, 1998.

[21] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, pp. 361–366, 2005.

[22] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA*, pp. 271–280, 2006.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.

[24] A. J. Goldberg and J. L. Hennessy. Performance debugging shared memory multiprocessor programs with MTOOL. In *SC'91*, pp. 481–490, 1991.

[25] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[26] S. Graham, P. Kessler, and M. McKusick. An execution profiler for modular programs. *Software—Practice and Experience*, 13(8):671–685, 1983.

[27] Y. He. Multicore-enabling the Murphi verification tool. Available from `http://software.intel.com/en-us/articles/multicore-enabling-the-murphi-verification-tool/`, 2009.

[28] Intel Corp. *Intel Cilk++ SDK Programmer's Guide*, 2009. Available from `http://software.intel.com/en-us/articles/download-intel-cilk-sdk/`. Document No. 322581-001US.

[29] Intel Corp. Intel Parallel Amplifier. Available from `http://software.intel.com/sites/products/documentation/studio/amplifier/en-us/2009/ug_docs/index.htm`. Document No. 320486-003US, 2009.

[30] Intel Corp. Intel Thread Profiler. Available from `http://software.intel.com/en-us/articles/intel-thread-profiler-for-windows-documentation/`, 2010.

[31] D. Lea. A Java fork/join framework. In *Java Grande*, pp. 36–43, 2000.

[32] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA*, pp. 227–242, 2009.

[33] C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomput.*, 51(3):244–257, 2010.

[34] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, 2010.

[35] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pp. 190–200, 2005.

[36] M. M. Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. In *PDCAT*, pp. 1–9, 2009.

[37] M. M. Maza and Y. Xie. FFT-based dense polynomial arithmetic on multi-cores. In *HPCS*, pp. 378–399, 2009.

[38] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[39] B. Mohr, A. D. Malony, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah. A performance monitoring interface for OpenMP. In *IWOMP*, 2002.

[40] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. *J. Supercomput.*, 23(1):105–128, 2002.

[41] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. A scalable approach to MPI application performance analysis. In *EUROPVMMPI*, pp. 309–316, 2005.

[42] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, 2008.

[43] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Scalable Parallel Lib. Conf.*, pp. 104–113, 1993.

[44] J. Reinders. *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[45] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[46] J. Seward. `bzip2` and `libbzip2`, version 1.0.5: A program and library for data compression. Available from `http://www.bzip2.org`.

[47] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPoPP*, pp. 229–240, 2009.

[48] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *PLDI*, pp. 441–452, 2009.

[49] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *SIGMETRICS*, pp. 240–250, 2002.

[50] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. *SIGPLAN Not.*, 36(7):123–132, 2001.

[51] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, and E. Lusk. From trace generation to visualization: A performance framework for distributed parallel systems. In *SC'00*, p. 50, 2000.