The Cilk++ Concurrency Platform

Charles E. Leiserson
MIT CSAIL and *Cilk Arts, Inc.*

Abstract

The availability of multicore processors across a wide range of computing platforms has created a strong demand for software frameworks that can harness these resources. This paper overviews the Cilk++ programming environment, which incorporates a compiler, a runtime system, and a race-detection tool. The Cilk++ runtime system guarantees to load-balance computations effectively. To cope with legacy codes containing global variables, Cilk++ provides a "hyperobject" library which allows races on nonlocal variables to be mitigated without lock contention or substantial code restructuring.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Algorithms, Performance, Design, Reliability, Languages.

Keywords

Amdahl's Law, dag model, hyperobject, multicore programming, multithreading, parallelism, parallel programming, race detection, reducer, span, speedup, work.

1 Introduction

Although the software community has extensive experience in serial programming using the C [18] and C++ [30] programming languages, they have found it hard to adapt C/C++ applications to run in parallel on multicore systems. In earlier work, the MIT Cilk system [14, 32] extended the C programming language with parallel computing constructs. The Cilk++ solution similarly extends C++, offering a gentle and reliable path to enable the estimated three million C++ programmers [31] to write parallel programs for multicore systems. Cilk++ is available for the Windows Visual Studio and the Linux/gcc compilers.

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 Cilkscreen are registered trademarks of Cilk Arts, Inc.

```
// Parallel quicksort
using namespace std;

#include <algorithm>
#include <iterator>
#include <functional>

template <typename T>
void qsort(T begin, T end) {
  if (begin != end) {
    T middle = partition(begin, end, bind2nd(
         less<typename iterator_traits<T>::
         value_type>(),*begin));
      cilk_spawn qsort(begin, middle);
      qsort(max(begin + 1, middle), end);
      cilk_sync;
  }
}

// Simple test code:
#include <iostream>
#include <cmath>

int main() {
  int n = 100;
  double a[n];

  cilk_for (int i=0; i<n; ++i) {
    a[i] = sin((double) i);
  }

  qsort(a, a + n);
  copy(a, a + n, ostream_iterator<double>(cout
       , "\n"));

  return 0;
}
```

Figure 1: Parallel quicksort implemented in Cilk++.

Like the MIT Cilk system [14, 32], Cilk++ is a ***faithful*** linguistic extension of C++, which means that parallel code retains its serial semantics when run on one processor. The Cilk++ extensions to C++ consist of just three keywords, which can be understood from an example. Figure 1 shows a Cilk++ program adapted from `http://www.cvgpr.uni-mannheim.de/heiler/qsort.html`, which implements the quicksort algorithm [7, Chapter 7]. Observe that the program would be an ordinary C++ program if the three keywords `cilk_spawn`, `cilk_sync`, and `cilk_for` were elided.

Parallel work is created when the keyword `cilk_spawn` precedes the invocation of a function. The semantics of spawning differ from a C++ function (or method) call only in that the parent can continue to execute in parallel with the child, instead of waiting for the child to complete as is done in C++. The scheduler in the Cilk++ runtime system takes the responsibility of scheduling the spawned functions on the individual processor cores of the multicore computer.

A function cannot safely use the values returned by its children until it executes a `cilk_sync` statement. The `cilk_sync` statement is a local "barrier," not a global one as, for example, is used in message-passing programming [23, 24]. In the quicksort example, a `cilk_sync` statement occurs on line 14 before the function returns to avoid the anomaly that would occur if the preceding calls to `qsort` were scheduled to run in parallel and did not complete before the return, thus leaving the vector to be sorted in an intermediate and inconsistent state.

In addition to explicit synchronization provided by the `cilk_sync` statement, every Cilk function syncs implicitly before it returns, thus ensuring that all of its children terminate before it does. Thus, for this example, the `cilk_sync` before the return is technically unnecessary.

Cilk++ improves upon the original MIT Cilk in several ways. It provides full support for C++ exceptions. Loops can be parallelized by simply replacing the keyword `for` with the keyword `cilk_for` keyword, which allows all iterations of the loop to operate in parallel. Within the `main` routine, for example, the loop starting on line 26 fills the array in parallel with random numbers. In the MIT Cilk system, such loops had to be rewritten by the programmer as divide-and-conquer recursion, but Cilk++ provides the `cilk_for` syntax for automatically parallelizing such loops. In addition, Cilk++ includes a library for mutual-exclusion (mutex) locks. Locking tends to be used much less frequently than in other parallel environments, such as Pthreads [17], because all protocols for control synchronization are handled by the Cilk++ runtime system.

The remainder of this paper is organized as follows. Section 2 provides a brief tutorial on the theory of parallelism. Section 3 describes the performance guarantees of Cilk++'s "work-stealing" scheduler and overviews how it operates. Section 4 briefly describes the Cilkscreen race-detection tool which guarantees to find race bugs in ostensibly deterministic code. Section 5 explains Cilk++'s "hyperobject" technology, which allows races on nonlocal variables to be mitigated without lock contention or restructuring of code. Finally, Section 6 provides some concluding remarks.

2 An overview of parallelism

The Cilk++ runtime system contains a provably efficient work-stealing scheduler [4, 14], which scales application performance linearly with processor cores, as long as the application exhibits sufficient parallelism (and the processor architecture provides sufficient memory bandwidth). Thus, to obtain good performance, the programmer needs to know what it means for his or her application to exhibit sufficient parallelism. Before describing the Cilk++ runtime system, it is helpful to understand something about the theory of parallelism.

Many discussions of parallelism begin with Amdahl's Law [1], originally proffered by Gene Amdahl in 1967. Amdahl made what amounts to the following observation. Suppose that 50% of a computation can be parallelized and 50% cannot. Then, even if the 50% that is parallel were run on an infinite number of processors, the total time is cut at most in half, leaving a speedup of at most 2. In general, if a fraction p of a computation can be run in parallel and the rest must run serially, Amdahl's Law upper-bounds the speedup by $1/(1-p)$.

Although Amdahl's Law provides some insight into parallelism, it does not *quantify* parallelism, and thus it does not provide a good understanding of what a concurrency platform such as Cilk++ should offer for multicore application performance. Fortunately, there is a simple theoretical model for parallel computing which provides a more general and precise quantification of parallelism that subsumes Amdahl's Law. The ***dag (directed acyclic graph) model of multithreading*** [3] views the execution of a multithreaded program as a set of instructions (the vertices of the dag) with graph edges indicating dependencies between instructions. (See Figure 2.) We say that an instruction x ***precedes*** an instruction y, sometimes 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 instructions are in ***parallel***, denoted $x \parallel y$. In Figure 2, for example, we have $1 \prec 2$, $6 \prec 12$, and $4 \parallel 9$.

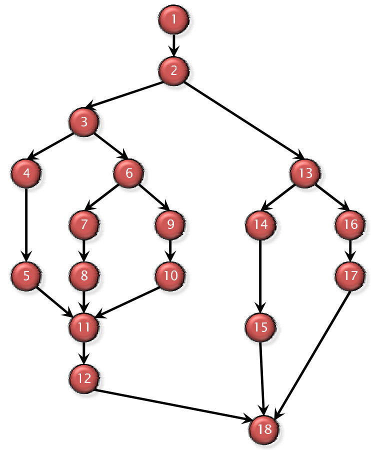

Figure 2: A directed acyclic graph representation of a multithreaded execution. Each vertex is an instruction. Edges represent ordering dependencies between instructions.

The dag model of multithreading can be interpreted in the context of the Cilk++ programming model. A `cilk_spawn` of a function creates two dependency edges emanating from the instruction immediately before the `cilk_spawn`: one edge goes to the first instruction of the spawned function, and the other goes to the first instruction after the spawned function. A `cilk_sync` creates dependency edges from the final instruction of each spawned function to the instruction immediately after the `cilk_sync`. A `cilk_for` can be viewed as divide-and-conquer parallel 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 important measure is ***work***, which is the total amount of time spent in all the instructions. Assuming for simplicity that it takes unit time to execute an instruction, the work for the example dag in Figure 2 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. Among the reasons that work is an important measure is because 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, each processor executes at most 1 instruction per unit time, and hence P processors can execute at most P instructions per unit time. Thus, with P processors, to do all the work, it must take 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 proportional to P, we say that the application exhibits ***linear speedup***. If it obtains speedup exactly

P (which is the best we can do in our model), we say that the application exhibits ***perfect linear speedup***. If the application obtains speedup greater than P (which cannot happen in our model due to the Work Law, but can happen in models that incorporate caching and other processor effects), we say that the application exhibits ***superlinear speedup***.

The Span Law

The second important measure is ***span***, which is the longest path of dependencies in the dag. The span of the dag in our example is 9, which corresponds to the path $1 \prec 2 \prec 3 \prec 6 \prec 7 \prec 8 \prec 11 \prec 12 \prec 18$. 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 2 is $18/9 = 2$. That means that there's little point in executing it with more than 2 processors, since additional processors will be surely starved for work.

As a practical matter, many problems admit considerable parallelism. For example, matrix multiplication of 1000×1000 matrices is highly parallel, with a parallelism in the milliions. Many problems on large irregular graphs, such as breadth-first search, generally exhibit parallelism on the order of thousands. Sparse matrix algorithms can often exhibit parallelism in the hundreds.

3 Runtime system

Although optimal multiprocessor scheduling is known to be NP-complete [15], Cilk++'s runtime system employs a "work-stealing" scheduler [4, 14] that achieves provably tight bounds. 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. Moreover, on a single core, typical programs run with negligible overhead (less than 2%).

Performance bounds

Specifically, for an application with T_1 work and T_∞ span running on a computer with P processors, the Cilk++ works-stealing scheduler achieves expected running time

$$T_P \leq T_1/P + O(T_\infty) \ . \tag{3}$$

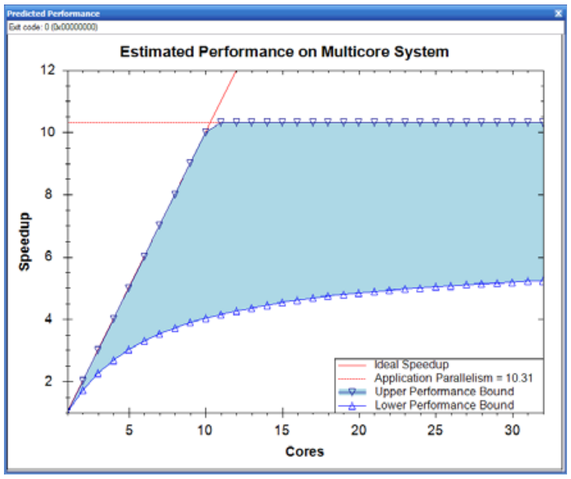

Figure 3: Parallelism profile of quicksort produced by the Cilk++ performance analyzer.

If the parallelism T_1/T_∞ exceeds the number P of processors by a sufficient margin, this bound (proved in [4]), 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 (3), the T_1/P term dominates the $O(T_\infty)$ 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 Cilk++ development environment contains a performance-analysis tool that allows a programmer to analyze the work and span of an application. Figure 3 shows the output of this tool running the quicksort program from Figure 1 on 100 million numbers. The upper bound on speedup provided by the Work Law corresponds to the line of slope 1, and the upper bound provided by the Span Law corresponds to the horizontal line at 10.31. The performance analysis tool also provides an estimated lower bound on speedup — the lower curve in the figure — based on ***burdened parallelism***, which takes into account the estimated cost of scheduling. Although quicksort seems naturally parallel, one can show that the expected parallelism for sorting n numbers is only $O(\lg n)$. Practical sorts with more parallelism exist, however. See [7, Chapter 27] for more details.

In addition to guaranteeing performance bounds, the Cilk++ runtime system also provides bounds on stack space. Specifically, on P processors, a Cilk++ program consumes at most P times the stack space of a single-processor execution. Consider the following simple code fragment:

```
for (int i=0; i<1000000000; ++i) {
    cilk_spawn foo(i);
}
cilk_sync;
```

This code conceptually creates one billion invocations of `foo` that operate logically in parallel. Executing on one processor, however, this Cilk++ code uses no more stack space than a serial C++ execution, that is, the call depth is of whichever invocation of `foo` requires the deepest stack. On two processors, it requires at most twice this space, and so on. This guarantee contrasts with that of more naive schedulers, which may create a work-queue of one billion tasks, one for each iteration of the subroutine `foo`, before executing even the first iteration, thus blowing out physical memory.

Work stealing

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

When a worker runs out of work, however, it becomes a ***thief*** and "steals" the top frame from another ***victim*** worker's stack. Thus, the stack is in fact a double-ended queue, 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 [4,14] that stealing is infrequent, and thus the cost of communication and synchronization to effect a steal is negligible.

The dynamic load-balancing capability provided by the Cilk++ runtime system adapts well in real-world multiprogrammed computing environments. If a worker becomes descheduled by the operating system (for example, because another application starts to run), the work of that worker can be stolen away by other workers. Thus, Cilk++ programs tend to "play nicely" with other jobs on the system.

Cilk++'s runtime system also makes Cilk++ programs ***performance-composable***. Suppose that a programmer develops a parallel library in Cilk++. That library can be called not only from a serial program or the serial portion of a parallel program, it can be invoked multiple times in parallel and continue to exhibit good speedup. In contrast, some concurrency platforms constrain library code to run on a given number of processors, and if multiple instances of the library execute simultaneously, they end up thrashing as they compete for processor resources.

4 Race detection

The Cilk++ development environment includes a ***race detector***, called Cilkscreen, a powerful debugging tool that greatly simplifies the task of ensuring that a parallel application is correct. We define a ***strand*** to be a sequence of serially executed instructions containing no parallel control, that is, a path in the multithreaded dag, where each vertex except the first in the path has at most one incoming edge and every vertex except the last in the path has at most one outgoing edge. A ***data race*** [26] exists if logically parallel strands access the same shared location, the two strands hold no locks in common, and at least one of the strands writes to the location. A data race is usually a bug, because the program may exhibit unexpected, nondeterministic behavior depending on how the strands are scheduled. Serial code containing nonlocal variables is particularly prone to the introduction of data races when the code is parallelized.

As an example of a race bug, suppose that line 13 in Figure 1 is replaced with the following line:

```
qsort(max(begin + 1, middle-1), end);
```

The resulting serial code is still correct, but the parallel code now contains a race bug, because the two subproblems overlap, which could cause an error during execution.

Race conditions have been studied extensively [6, 8–12, 16, 20–22,25,27–29]. They are pernicious and occur nondeterministically. A program with a race bug may execute successfully millions of times during testing, only to raise its head after the application is shipped. Even after detecting a race bug, writing regression tests to ensure its continued absence is difficult.

```
 1  bool has_property(Node *);
 2  std::list<Node *> output_list;
 3  // ...
 4  void walk(Node *x)
 5  {
 6    if (x)
 7    {
 8      if (has_property(x))
 9      {
10        output_list.push_back(x);
11      }
12      walk(x->left);
13      walk(x->right);
14    }
15  }
```

Figure 4: C++ code to create a list of all the nodes in a binary tree that satisfy a given property.

The Cilkscreen race detector is based on provably good algorithms [2,6,11] developed originally for MIT Cilk. In a single serial execution on a test input for a deterministic program, Cilkscreen guarantees to report a race bug if the race bug is ***exposed***: that is, if two different schedulings of the parallel code would produce different results. Cilkscreen uses efficient data structures to track the series-parallel relationships of the executing application during a serial execution of the parallel code. As the application executes, Cilkscreen uses dynamic instrumentation [5, 19] to intercept every load and store executed at user level. Metadata in the Cilk++ binaries allows Cilkscreen to identify the parallel control constructs in the executing application precisely, track the series-parallel relationships of strands, and report races precisely. Additional metadata allows the race to be localized in the application source code.

5 Reducer hyperobjects

Many serial programs use ***nonlocal variables***, which are variables that are bound outside of the scope of the function, method, or class in which they are used. If a variable is bound outside of all local scopes, it is a ***global variable***. Nonlocal variables have long been considered a problematic programming practice [33], but programmers often find them convenient to use, because they can be accessed at the leaves of a computation without the overhead and complexity of passing them as parameters through all the internal nodes. Thus, nonlocal variables have persisted in serial programming. In the world of parallel computing, nonlocal variables may inhibit otherwise independent parts of a multithreaded program from operating in parallel, because they introduce races. This section describes Cilk++ reducer hyperobjects [13], which can mitigate races on nonlocal variables without creating lock contention or requiring code restructuring.

As an example of how a nonlocal variable can introduce a data race, consider the problem of walking a binary tree to make a list of those nodes that nodes satisfy a given property. A C++ code to solve the problem is abstracted in Figure 4. If the node `x` being visited is nonnull, the code checks whether `x` has the desired property in line 8, and if so, it appends `x` to the list stored in the global variable `output_list` in line 10. Then, it recursively visits the left and right children of `x` in lines 12 and 13.

Figure 5 illustrates a straightforward parallelization of this code in Cilk++. In line 12 of the figure, the `walk` function is spawned recursively on the left child, while the parent continues on to execute an ordinary recursive call of `walk` in line 13. As the recursion unfolds, the running program generates a tree of parallel execution

```
 1  bool has_property(Node *);
 2  std::list<Node *> output_list;
 3  // ...
 4  void walk(Node *x)
 5  {
 6    if (x)
 7    {
 8      if (has_property(x))
 9      {
10        output_list.push_back(x);
11      }
12      cilk_spawn walk(x->left);
13      walk(x->right);
14      cilk_sync;
15    }
16  }
```

Figure 5: A naive Cilk++ parallelization of the code in Figure 4. This code has a data race in line 10.

```
 1  bool has_property(Node *);
 2  std::list<Node *> output_list;
 3  mutex L;
 4  // ...
 5  void walk(Node *x)
 6  {
 7    if (x)
 8    {
 9      if (has_property(x))
10      {
11        L.lock();
12        output_list.push_back(x);
13        L.unlock();
14      }
15      cilk_spawn walk(x->left);
16      walk(x->right);
17      cilk_sync;
18    }
19  }
```

Figure 6: Cilk++ code that solves the race condition using a mutex.

that follows the structure of the binary tree. Unfortunately, this naive parallelization contains a data race. Specifically, two parallel instantiations of `walk` may attempt to update the shared global variable `output_list` in parallel at line 10.

The traditional solution to fixing this kind of data race is to associate a mutual-exclusion lock (mutex) `L` with `output_list`, as is shown in Figure 6. Before updating `output_list`, the mutex `L` is acquired in line 11, and after the update, it is released in line 13. Although this code is now correct, the mutex may create a bottleneck in the computation. If there are many nodes that have the desired property, the contention on the mutex can destroy all the parallelism. For example, on one set of test inputs for a real-world tree-walking code that performs collision-detection of mechanical assemblies, lock contention actually degraded performance on 4 processors so that it was worse than running on a single processor. In addition, the locking solution has the problem that it jumbles up the order of list elements. That might be okay for some applications, but other programs may depend on the order produced by the serial execution.

An alternative to locking is to restructure the code to accumulate the output lists in each subcomputation and concatenate them when the computations return. If one is careful, it is also possible to keep the order of elements in the list the same as in the serial execution. For the simple tree-walking code, code restructuring may suffice, but for many larger codes, disrupting the original logic can be time-consuming and tedious undertaking, and it may require expert skill,

```
 1  #include <reducer_list.h>
 2  bool has_property(Node *);
 3  cilk::hyperobject<cilk::reducer_list_append<
        Node *> > output_list;

 4  // ...
 5  void walk(Node *x)
 6  {
 7    if (x)
 8    {
 9      if (has_property(x))
10      {
11        output_list().push_back(x);
12      }
13      cilk_spawn walk(x->left);
14      walk(x->right);
15      cilk_sync;
16    }
17  }
```

Figure 7: A Cilk++ parallelization of the code in Figure 4, which uses a reducer hyperobject to avoid data races.

making it impractical for parallelizing large legacy codes.

Cilk++ provides a novel approach [13] to avoiding data races in code with nonlocal variables. A Cilk++ ***reducer hyperobject*** is a linguistic construct that allows many strands to coordinate in updating a shared variable or data structure independently by providing them different but coordinated views of the same object. The state of a hyperobject as seen by a strand of an execution is called the strand's "view" of the object at the time the strand is executing. A strand can access and change any of its view's state independently, without synchronizing with other strands. Throughout the execution of a strand, the strand's view of the reducer is private, thereby providing isolation from other strands. When two or more strands join, their different views are combined according to a system- or user-defined `reduce()` method. Thus, reducers preserve the advantages of parallelism without forcing the programmer to restructure the logic of his or her program.

As an example, Figure 7 shows how the tree-walking code from Figure 4 can be parallelized using a reducer. Line 3 declares `output_list` to be a reducer hyperobject for list appending. The `reducer_list_append` class implements a `reduce` function that concatenates two lists, but the programmer of the tree-walking code need not be aware of how this class is implemented. All the programmer does is identify the global variables as the appropriate type of reducer when they are declared. No logic needs to be restructured, and if the programmer fails to catch all the use instances, the compiler reports a type error.

This parallelization takes advantage of the fact that list appending is associative. That is, if we append a list L_1 to a list L_2 and append the result to L_3, it is the same as if we appended list L_1 to the result of appending L_2 to L_3. As the Cilk++ runtime system load-balances this computation over the available processors, it ensures that each branch of the recursive computation has access to a private view of the variable `output_list`, eliminating races on this global variable without requiring locks. When the branches synchronize, the private views are reduced (combined) by concatenating the lists, and Cilk++ carefully maintains the proper ordering so that the resulting list contains the identical elements in the same order as in a serial execution.

6 Conclusion

Multicore microprocessors are now commonplace, and Moore's Law is steadily increasing the pressure on software developers to multicore-enable their codebases. Cilk++ provides a simple but effective concurrency platform for multicore programming which leverages almost two decades of research on multithreaded programming. The Cilk++ model builds upon the sound theoretical framework of multithreaded dags, allowing parallelism to be quantified in terms of work and span. The Cilkscreen race detector allows race bugs to be detected and localized. Cilk++'s hyperobject library mitigates races on nonlocal variables. Although parallel programming will surely continue to evolve, Cilk++ today provides a full-featured suite of technology for multicore-enabling any compute-intensive application.

Acknowledgments

Many thanks to the great team at Cilk Arts and to our many customers who have helped us refine the Cilk++ system. Thanks to Patrick Madden of SUNY Binghamton for proposing extensive revisions to the original manuscript.

References

[1] Gene Amdahl. The validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2004)*, pages 133–144, Barcelona, Spain, June 2004.

[3] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Twenty Fifth Annual ACM Symposium on Theory of Computing*, pages 362–371, San Diego, California, May 1993.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[5] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2004.

[6] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 298–309, Puerto Vallarta, Mexico, June 28–July 2 1998.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[8] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM Press, 1990.

[9] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM Press, May 1991.

[10] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '91*, pages 580–588, November 1991.

[11] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June22–25 1997.

[12] Yaacov Fenster. Detecting parallel access anomalies. Master's thesis, Hebrew University, March 1998.

[13] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-First Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '09)*, Calgary, Canada, August 2009. To appear.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, 1979.

[16] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Analyzing traces with anonymous synchronization. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II70–II77, August 1990.

[17] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Standard 1003.1, 1996 Edition.

[18] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[20] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing'91*, pages 24–33. IEEE Computer Society Press, 1991.

[21] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 135–144, Atlanta, Georgia, June 1988.

[22] Sang Lyul Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 235–244, Palo Alto, California, April 1991.

[23] The MPI Forum. MPI: A message passing interface. In *Supercomputing '93*, pages 878–883, Portland, Oregon, November 1993.

[24] The MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996. Available from: `citeseer.ist.psu.edu/517818.html`.

[25] Robert H. B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *Proceedings of the 1992 International Conference on Parallel Processing*, St. Charles, Illinois, August 1992.

[26] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[27] Itzhak Nudler and Larry Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[28] Dejan Perković and Peter Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington, October 1996.

[29] Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multithreaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[31] Bjarne Stroustrup. *C++ in 2005*. Addison-Wesley, 2005. Preface to the Japanese translation.

[32] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.2.3 Reference Manual*, April 2006. Available from: `http://supertech.csail.mit.edu/cilk/home/software.html`.

[33] William Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, 1973.