Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2010-038 August 5, 2010

Parallelizing Sequential Programs With Statistical Accuracy Tests

Sasa Misailovic, Deokhwan Kim, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Parallelizing Sequential Programs With Statistical Accuracy Tests

Sasa Misailovic
MIT CSAIL/EECS
misailo@csail.mit.edu

Deokhwan Kim
MIT CSAIL/EECS
dkim@csail.mit.edu

Martin Rinard
MIT CSAIL/EECS
rinard@csail.mit.edu

Abstract

We present QuickStep, a novel system for parallelizing sequential programs. QuickStep deploys a set of *parallelization transformations* that together induce a search space of candidate parallel programs. Given a sequential program, representative inputs, and an accuracy requirement, QuickStep uses performance measurements, profiling information, and statistical accuracy tests on the outputs of candidate parallel programs to guide its search for a parallelization that maximizes performance while preserving acceptable accuracy. When the search completes, QuickStep produces an interactive report that summarizes the applied parallelization transformations, performance, and accuracy results for the automatically generated candidate parallel programs. In our envisioned usage scenarios, the developer examines this report to evaluate the acceptability of the final parallelization and to obtain insight into how the original sequential program responds to different parallelization strategies. It is also possible for the developer (or even a user of the program who has no software development expertise whatsoever) to simply use the best parallelization out of the box without examining the report or further investigating the parallelization.

Results from our benchmark set of applications show that QuickStep can automatically generate accurate and efficient parallel programs — the automatically generated parallel versions of five of our six benchmark applications run between 5.0 and 7.7 times faster on 8 cores than the original sequential versions. Moreover, a comparison with the Intel icc compiler highlights how QuickStep can effectively parallelize applications with features (such as the use of modern object-oriented programming constructs or desirable parallelizations with infrequent but acceptable data races) that place them inherently beyond the reach of standard approaches.

1. Introduction

For the last several decades, improvements in uniprocessor performance have been the primary source of increased computing power. It is clear, however, that fundamental limits on the underlying computing substrate will eliminate this source of performance improvement. The computing community will instead be forced to rely on parallel computing for future performance improvements. Indeed, this trend is already evident. Within the last several years, uniprocessor performance has not significantly improved. During this same time, computers with multiple processor cores have become the industry standard. Unfortunately, the difficulty of developing parallel software is a key obstacle to exploiting the considerable computational power that multiprocessors (such as modern multicore machines) otherwise have to offer.

We present a new system, QuickStep, that is designed to help developers obtain parallel software that can effectively exploit modern multicore computing platforms. Instead of relying on the developer to manually identify and implement appropriate parallelization strategies, QuickStep implements a set of *parallelization transformations*. Given an initial sequential program, these transformations induce a search space of candidate parallel programs. Guided by performance measurements, a simple static analysis, loop profiling information, memory profiling information, and statistical accuracy tests on the outputs of test executions, QuickStep automatically searches this space to find a parallel program that maximizes performance subject to a specified accuracy goal. The developer can then examine this program and the accompanying automatically generated interactive parallelization report to determine the overall acceptability of the final parallelization and obtain insight into the parallelization process. We anticipate that varying degrees of investigation into the parallelization will be appropriate in different contexts. In some contexts we expect the developer (or even a user who has no software development expertise at all) to simply accept and use the best parallelization with further investigation into the parallelization strategy or automatically generated parallel code.

1.1 Transformations

QuickStep implements three kinds of transformations:

- **Parallelism Introduction Transformations:** Transformations that introduce parallel execution. QuickStep currently implements one parallelism introduction transformation: **loop parallelization**. Note that because the iterations of the resulting parallel loop execute without synchronization, anomalies such as data races may cause the parallel program to crash or produce an unacceptably accurate result.
- **Accuracy Enhancing Transformations:** Transformations that are designed to enhance the accuracy of the parallel program. If a parallelism introduction transformation produces an unacceptably accurate program, accuracy enhancing transformations may restore acceptably accurate execution. QuickStep implements two accuracy enhancing transformations: **synchronization introduction**, which replaces unsynchronized operations with synchronized operations (the goal is to eliminate data races), and **privatization**, which gives each thread its own copy of otherwise shared local variables (the goal is to eliminate interference between parallel loop iterations).
- **Performance Enhancing Transformations:** Transformations that are designed to enhance the performance of the parallel program. QuickStep implements two performance enhancing transformations: **replication introduction**, which replicates objects accessed by parallel tasks, then combines the replicas for sequential access (the goal is to eliminate bottlenecks associated with frequently executed synchronized operations on shared objects) and **loop scheduling**, which applies different parallel loop scheduling algorithms (the goal is to find a scheduling policy that interacts well with the specific characteristics of the loop).

1.2 Searching The Parallelization Space

Given an initial sequential program, the parallelization transformations induce a space of corresponding parallel programs. QuickStep searches this space as follows. QuickStep attempts to parallelize a single loop at a time, prioritizing the attempted parallelization of loops that consume more execution time over the attempted parallelization of loops that consume less execution time. QuickStep first applies the parallelization transformation to the current loop. It then explores the parallelization space for this loop by repeatedly applying accuracy and performance enhancing transformations (prioritizing accuracy enhancing transformations until it has obtained acceptably accurate execution).

If it is unable to obtain an acceptably accurate parallelization, QuickStep abandons the current loop and moves on to the next. Once it has processed the most time-consuming loops and obtained the final parallelization, QuickStep produces an interactive parallelization report that a developer can navigate to evaluate the acceptability of this parallelization and obtain insight into how the application responds to different parallelization strategies.

1.3 Accuracy Metric

In many cases it may be desirable to produce a parallel program that produces the same result as the original sequential program. But in other cases the best parallel version may, because of phenomena such as infrequent data races or reordered parallel accumulations, produce a result that differs within acceptable bounds from the result that the sequential program produces. QuickStep is therefore designed to work with an *accuracy metric* that quantifies the difference between an output from the original sequential program and a corresponding output from the automatically generated parallel program run on the same input [30, 32]. The accuracy metric first uses an *output abstraction* (which typically selects relevant output components or computes a measure of the output quality) to obtain a sequence of numbers $o_1, \ldots, o_m$ from a sequential execution and a corresponding sequence $\hat{o}_1, \ldots, \hat{o}_m$ from a parallel execution on the same input. It then uses the following formula to compute the *distortion* d, which measures the accuracy of the parallel execution:

$$d = \frac{1}{m} \sum_{i=1}^{m} \left| \frac{o_i - \hat{o}_i}{o_i} \right|$$

The closer the distortion d is to zero, the less the parallel execution distorts the output. Note that because each difference $o_i - \hat{o}_i$ is scaled by the corresponding output component o_i from the sequential execution and because the sum is divided by the number of output components m, it is possible to meaningfully compare distortions d obtained from executions on different inputs even if the number of selected outputs is different for the different inputs. Given an accuracy metric, an *accuracy bound* b is an upper bound on the acceptable distortion.

1.4 Statistical Accuracy Test

QuickStep's statistical accuracy test executes the parallel program multiple times, treating each execution as a Bernoulli trial which succeeds if the execution satisfies the accuracy bound b (i.e., if the observed distortion d is at most b) and fails if it does not. The test terminates when QuickStep has observed enough executions to make a statistically well-founded judgement to either accept or reject the parallelization. QuickStep uses the Wald sequential probability ratio (SPR) test (see Section 4) to make this judgement. During its search of the parallel program space, QuickStep sets the parameters of this test to accept a parallelization if the program satisfies the accuracy bound at least 90% of the time with a false positive rate of at most (approximately) 10% (i.e., QuickStep accepts an unacceptably inaccurate program at most approximately 10% of the time). During the final accuracy test, the program must satisfy the accuracy bound at least 99% of the time with a false positive rate of at most (approximately) 1%. See Section 4 for more details.

It is possible for a parallelization to cause the program to fail to produce a well-formed output (typically because the program crashes). In such cases QuickStep rejects the parallelization (although it would be possible to configure QuickStep to accept parallelizations that have a low likelihood of failing in this way).

1.5 Primary Usage Scenario

Our primary usage scenario involves a developer who wants to obtain a parallel version of an existing sequential program. The developer provides QuickStep with an output abstraction, an accuracy goal, and a set of representative inputs. QuickStep then searches the induced space of parallel programs, using the representative inputs to perform the test executions required to statistically evaluate the accuracy of the candidate parallel programs, drive the exploration of the search space, and find the final parallelization. Finally, QuickStep produces a parallelization report that summarizes the applied transformations and the accuracy and performance results.

The developer examines the report to obtain an overview of the characteristics of the final parallelization, then uses the report to evaluate the applied parallelization and determine the overall acceptability of this parallelization. If appropriate, the developer may either accept or reject the parallelization or use the parallelization as a basis for further development. The developer may also use the report to investigate the characteristics of other candidate parallelizations to obtain insight into the characteristics of the explored space of parallel programs.

More generally, we anticipate a spectrum of usage scenarios. At one end of the spectrum, the developer will examine the generated parallel program as outlined above, fully understand the parallel program, make an acceptability decision based on this understanding, and potentially apply further transformations to manually obtain an even more desirable parallelization. At the other end of the spectrum, the developer (or even a user with no software development expertise at all) may use QuickStep as a parallelizing compiler — he or she may simply rely on the accuracy results from the representative inputs and accept the final parallelization without examining the parallel program at all. And of course, we envision intermediate scenarios based on more or less thorough examinations of the automatically generated parallel program, with the specific context determining the usage pattern.

1.6 Experimental Results

To evaluate the effectiveness of QuickStep's approach, we obtained a set of benchmark sequential programs, then used QuickStep to parallelize this set of programs and generate corresponding parallelization reports. Our results show that QuickStep is able to effectively parallelize five out of the six programs, with the final parallel versions running, on our test inputs, between a factor of 5.0 and 7.7 faster (on 8 cores) than the corresponding sequential versions.

We used the parallelization reports to evaluate the acceptability of the final parallelizations. Our evaluation shows that, for the programs in our benchmark set, QuickStep is able to produce parallelizations that are acceptable for all inputs (and not just the representative inputs that it uses to drive the exploration of the search space). Moreover, each final parallel program contains at most a handful of parallel loops, each of which requires at most only several synchronization or replication transformations to produce an acceptably accurate program with good parallel performance. The parallelizations are therefore amenable to developer evaluation with reasonable developer effort.

1.7 Comparison With the Intel icc Compiler

For comparison purposes, we also compiled the benchmarks with the Intel icc parallelizing compiler [1]. When configured to parallelize only loops that will increase performance, the icc compiler finds only one small loop to parallelize. When this constraint is removed, it finds only small inner loops whose parallelization does not improve and in some cases significantly degrades the performance (because the parallelization overhead overwhelms the performance gains from parallel execution).

These results highlight a key advantage of the QuickStep approach. QuickStep, like almost all parallelizing compilers, is designed to exploit parallelism available in loops. But unlike standard compilers (which statically analyze the sequential program to produce a parallel program with identical semantics), QuickStep simply applies transformations, then uses the results of test executions to determine if the resulting parallelization is (statistically) acceptably accurate. QuickStep can therefore easily parallelize outer loops with large loop bodies that use constructs (such as object references) that are difficult or impossible for standard compilers to successfully analyze. QuickStep can also produce desirable parallelizations (for example, efficient parallelizations that may contain infrequent data races) that are inherently outside the scope of standard techniques. As our comparison with the Intel icc compiler illustrates, these properties can enable QuickStep to find effective parallelizations that lie well beyond the reach of standard parallelizing compilers.

1.8 Comparison with Jade Parallelizations

We also discuss the difference between the QuickStep parallelizations and the Jade [31] parallelizations for corresponding applications (when such applications exist). Even though Jade is an implicitly parallel language with sequential semantics, we find that there are significant differences in the amount of effort required to use the two different approaches. To use Jade, a developer must understand the computation in enough detail to modify the application to introduce Jade task constructs at appropriate locations within the source code. QuickStep, on the other hand, operates directly on unmodified sequential programs with no need for the developer to modify the source code (unless such modifications are necessary to create or expose enough parallelism for QuickStep to find a parallelization with good parallel performance). QuickStep can also produce statistically accurate parallel computations that may contain data races and execute nondeterministically; Jade programs execute have no data races and execute deterministically.

Perhaps more significantly, some Jade programs require substantial developer effort to reallocate data structures so that the Jade object structure works effectively with the Jade task structure to enable the Jade run-time system to exploit sufficient parallelism. QuickStep, on the other hand, works directly with the application's original data structures. There are counterbalancing advantages: QuickStep requires less developer involvement, but Jade applications can execute on both shared-memory and message-passing platforms without modification. QuickStep can only produce parallelizations for computing platforms that provide the abstraction of a single shared address space.

Finally, QuickStep and Jade can exploit different kinds of parallelism. QuickStep is designed to exploit loop-level parallelism; Jade was designed to exploit potentially irregular parallelism available across tasks regardless of the specific control constructs used to generate the computation. QuickStep is designed to produce potentially nondeterministic but statistically accurate computations that may contain data races (or potentially reordered loop iterations that may introduce acceptable amounts of noise into the output); Jade was designed to produce computations that execute deterministically without data races in accord with the sequential semantics.

1.9 Scope

QuickStep is designed to parallelize programs whose main source of parallelism is available in loops. The loop iterations need not be independent — they can update shared data as long as the updates to shared objects commute [34]. They can also contain unsynchronized data races and other forms of nondeterminism as long as these phenomena do not make the output unacceptably inaccurate. QuickStep can also automatically eliminate bottlenecks that arise when the program combines multiple contributions into a single result.

We acknowledge that there are many programs that require different parallelization strategies. Examples include programs with producer/consumer parallelism and programs with complex data-dependent parallel dependence graphs. QuickStep is not designed to effectively parallelize such programs. But within its intended scope, QuickStep can offer unprecedented parallelization capabilities. Because it is not handicapped by the need to analyze the program, it can parallelize programs with complex features that lie well beyond the reach of any previously envisioned parallelizing compiler. Because it does not need to generate a parallel program that always produces the identical result as the sequential program, also it has access to a broader range of parallelization strategies than previously envisioned parallelizing compilers.

Of particular interest are programs that heavily use modern programming constructs such as objects and object references. Several of the sequential programs in our benchmark set are written in C++ and use these constructs heavily. QuickStep has no difficulty parallelizing programs that use these constructs. Traditional parallelizing compilers, of course, have demonstrated impressive results for programs that use affine access functions to access dense matrices, but are typically not designed to analyze and parallelize programs that heavily use object references. Our results show that QuickStep, in contrast, can effectively parallelize programs that use such constructs.

QuickStep's statistical accuracy guarantees are based on sampling executions of the parallel program on representative inputs. These guarantees are not valid for inputs that elicit behavior significantly different from the behavior that the representative inputs elicit. It is therefore the responsibility of the developer or user to provide inputs with characteristics representative of the inputs that will be used in production. The developer evaluation of the final parallelization can also help to ensure that the final parallelization is acceptable for all inputs.

The statistical accuracy guarantees are also not valid for hardware platforms whose characteristics differ significantly from those on which the final accuracy test was performed. Moving applications between hardware platforms may therefore require the reexecution of at least the final accuracy test. It is also possible for the best parallelization to change, in which case it may be advisable to reexecute the entire parallelization process.

1.10 Contributions

This paper makes the following contributions:

- **Basic Approach:** It introduces the basic QuickStep approach of developing a set of parallelization transformations, then searching the resulting induced space of parallel programs to find a program that maximizes performance while preserving acceptably accurate execution.
- **Parallelization Transformations:** It identifies three kinds of parallelization transformations (parallelism introduction, accuracy enhancing, and performance enhancing transformations) and presents specific instances of each kind of transformation. These instances are designed to work together to produce efficient and accurate parallel programs.

- **Search Algorithm:** It presents an algorithm for automatically searching the induced space of parallel programs. This algorithm uses profiling information, performance measurements, and accuracy results from executions on representative inputs to guide the search.
- **Statistical Accuracy Tests:** It introduces the use of statistical accuracy tests to determine if the likelihood that a candidate parallel program will produce an acceptably accurate result is acceptable.
- **Interactive Parallelization Reports:** It introduces interactive parallelization reports that present the performance and accuracy characteristics of the candidate parallel programs, identify the applied transformations for each program, and summarize the overall search process. These reports are designed to facilitate developer evaluation of the automatically generated parallel programs and to help the developer obtain insight into how the original sequential program responds to various parallelization strategies.
- **Experimental Results:** It presents experimental results obtained by using QuickStep to parallelize a set of benchmark sequential applications. These results show that QuickStep is able to produce accurate and efficient parallel versions of five of the six applications. And our examination of the resulting parallelizations indicates that they are acceptable for all inputs and not just the representative inputs used to drive the parallelization process.
- **Comparison With Other Approaches:** It presents a comparison with the Intel icc parallelizing compiler and the Jade parallelizations of corresponding applications. The icc comparison highlights how QuickStep (in comparison with other parallelizing compilers) is capable of automatically parallelizing applications that are inherently beyond the reach of standard approaches. The Jade comparison highlights how QuickStep (in comparison with an implicitly parallel language) can minimize or even eliminate developer effort in obtaining a successful parallelization.

2. Example

Figure 1 presents an example that we use to illustrate the operation of QuickStep. The example computes pairwise interactions between simulated water molecules (both stored temporarily in `scratchPad`s for the purposes of this computation). The two loops in `interf` generate the interactions. `interact` calls `cshift` to compute the results of each interaction into two 3 by 3 arrays (`Res1` and `Res2`). `updateForces` then uses the two arrays to update the vectors that store the forces acting on each molecule, while `addval` updates the `VIR` accumulator object, which stores the sum of the virtual energies of all the interactions.

2.1 Profiling Runs

QuickStep starts with the source code of the program, some representative inputs, an output abstraction (which identifies relevant outputs) and an accuracy bound of 0.003, all specified by the developer. QuickStep next runs the sequential computation on the representative inputs and records the running times and outputs. It next generates an instrumented version of the program that counts the number of instructions executed in each loop and records the dynamic loop nesting information. It then runs this instrumented version of the program to obtain the loop profiling information (which identifies the most time-consuming loops in the program and the dynamic nesting relationships between these programs). QuickStep next generates an instrumented version of the program that generates a trace of the addresses that the program reads and writes. It then runs this version to obtain the memory profiling information (which identifies potentially interfering accesses from parallel loop iterations).

```
void ensemble::interf(){
  int i, j;
  scratchPad *p1, *p2;

  for(i = 0; i < numMol-1; i++) {
    for(j = i+1; j < numMol; j++){
      p1 = getPad(j);
      p2 = getPad(i);
      interact(p1,p2);
    }
  }
}

void ensemble::interact
  (scratchPad *p1, scratchPad *p2) {
  double incr, Res1[3][3], Res2[3][3];

  incr = cshift(p1,p2,Res1,Res2);
  p1->updateForces(Res1);
  p2->updateForces(Res2);
  VIR.addval(incr);
}

void scratchPad::updateForces(double Res[3][3]) {
  this->H1force.vecAdd(Res[0]);
  this->Oforce.vecAdd(Res[1]);
  this->H2force.vecAdd(Res[2]);
}
```

Figure 1: Example Computation

2.2 Search of the Parallelization Space

The loop profiling information indicates that almost all of the execution time is spent in an outer loop that iterates over the time steps in the simulation. QuickStep's attempted parallelization of this loop fails because it produces computations with unacceptable accuracy.

QuickStep next moves on to the `interf` outer loop. It generates parallel code that executes the iterations of this loop in parallel. This initial parallelization (with no synchronization except the barrier synchronization at the end of the loop) produces parallel executions that are close to accurate enough but, in the end, fail the statistical accuracy test.

The memory profiling information indicates that there are potential data races at multiple locations in the parallel loop, with the densest races occurring within the `accumulator` `addval` operation invoked from the `interact` method. Figure 2 presents (relevant methods of) the `accumulator` class. Each `accumulator` contains several additional implementations of the basic `addval` operation — the `addvalSync` operation, which uses a multiple exclusion lock to make the `addval` execute atomically, and the `addvalRepl` operation, which adds the contributions into local replicas without synchronization.[1] Based on the memory profiling information, QuickStep invokes the synchronized version of the `addval` method, changing the call site in `interact` to invoke the `addvalSync` method instead of the `addval` method.

This transformation produces a parallel program with statistically acceptable accuracy but unacceptable performance. The parallel program takes almost the same time to execute as the original sequential program. QuickStep operates on the assumption that there is a bottleneck on the synchronized `addvalSync` updates and that this bottleneck is the cause of the poor performance. It therefore replaces the call to `addvalSync` with a call to the replicated

[1] The actual implementation of this accumulator uses padding to avoid potential false sharing interactions.

```
class accumulator {
 double *vals;
 volatile bool isCached;
 volatile double cachedVal;
 pthread_mutex_t mutex;
 double cache() {
  double val = 0.0;
  for (int i = 0; i < num_thread; i++) {
   val+= vals[i]; vals[i] = 0;
  }
  cachedVal += val; isCached = true;
  return val;
 }
public:
 double read() {
  if (isCached) return cachedVal;
  return cache();
 }
 void addval(double d) {
  if (!isCached) cache();
  cachedVal = cachedVal + d;
  isCached = true;
 }
 void addvalSync(double d) {
  pthread_mutex_lock(&mutex);
  addval(d);
  pthread_mutex_unlock(&mutex);
 }

 void addvalRepl(double d) {
  if (isCached) isCached = false;
  vals[this_thread] += d;
 }
};
```

Figure 2: Example Accumulator

version of the `addval` method, changing the call site in `interact` to invoke the `addvalRepl` method. The initial test executions indicate that this version has good performance. And the remaining data races (which occur when the computation updates the vectors that store the forces acting on each molecule) occur infrequently enough so that the computation produces acceptably accurate results.

After a similar parallelization process for the remaining time intensive loop, the outer loop in the `poteng` method (not shown), which accounts for the vast majority of the remaining execution time, QuickStep has found several parallelizations that exhibit both good performance and acceptably accurate output. QuickStep takes the parallelization with the best performance and runs the final, more stringent, statistical accuracy test on this parallelization, which passes the test to become the final accepted parallelization.

2.3 Interactive Report

As it explores the induced space of parallel programs, QuickStep produces a log whose entries specify the applied parallelization transformations, the performance, and the accuracy for each execution of each generated parallel program. QuickStep processes this log to produce an interactive report that the developer can examine to evaluate the final parallelization and gain insight into how the different transformations affect the parallel behavior of the application. The interactive report takes the form of a set of (dynamically generated) linked web pages. The interactive reports for the benchmark programs in this paper are available at http://people.csail.mit.edu/dkim/quickstep/.

2.3.1 The Start Page

Figure 3 presents a screen shot of the start page of the interactive report for the Water application from our benchmark suite (the example presented in this section is a simplified version of this application) running on the 1000 input. QuickStep produces a similar page

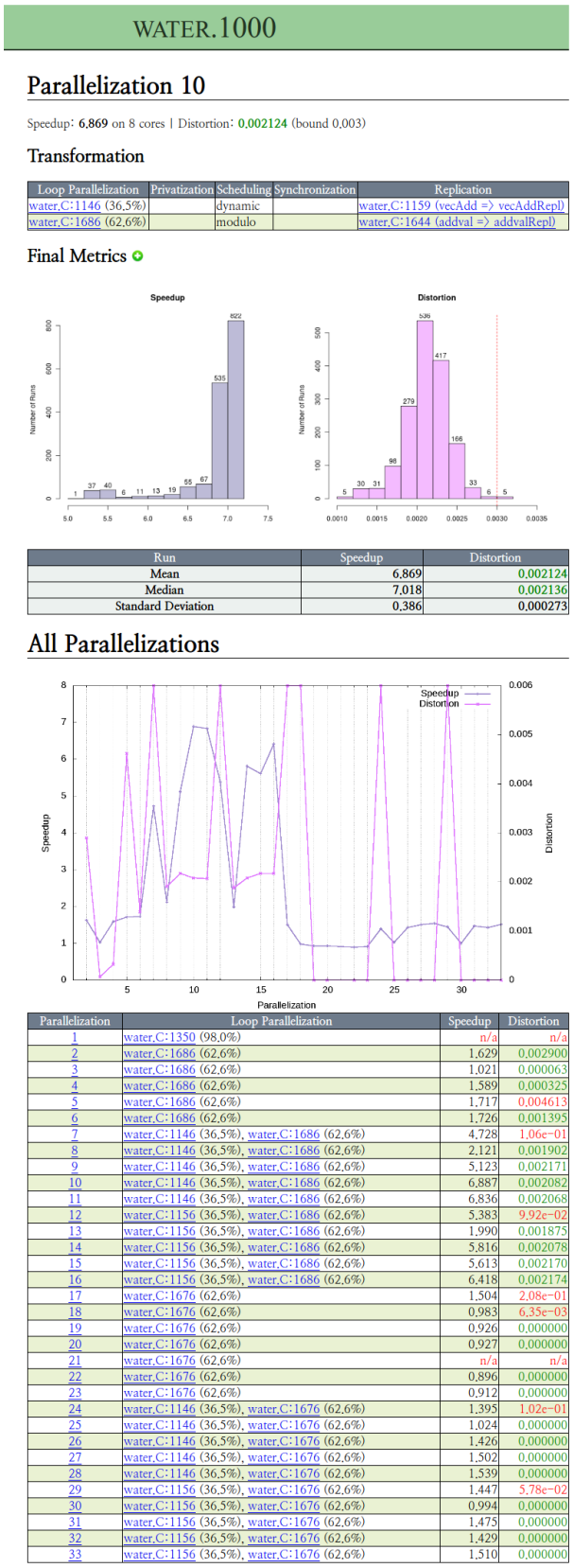

Figure 3: QuickStep Interactive Report Screen Shot

```
1685 void ensemble::interfLoop1(){
1686   int i;
1687   int max =  numMol-1;
1688    #pragma omp parallel for schedule(static, 1)
1689   for(i = 0; i < max; i++) {
1690     interfLoop2(i);
1691   }
1692 }
```

Figure 4: Source Code for Parallelized Loop

```
1153 void ensemble::potengLoop2(int idx){
1154   int i;
1155
1156
1157   int max =  numMol;
1158   for(i=idx+1; i < max; i++) {
1159      double tmp[3];
1160     interPoteng2Aux(&pad[idx],&pad[i],tmp);
1161     POT.vecAddRepl(tmp);
1162   }
1163 }
```

Figure 5: Source Code for Replication Transformation

for each combination of application and input in the benchmark suite. This page summarizes the final parallelization of this application. The header presents the number (in this case, Parallelization 10) of the final parallelization, the mean speedup (in this case, 6.869 on 8 cores), and the mean distortion (in this case, 0.002).

The Transformation section of the page provides information about the applied transformations for the final version, with links to the transformed source code. In our example, the final parallel program has two parallel loops (at lines 1146 and 1686 of the file Water.C). The computation spends 36.5% of its time in the first loop and 62.6% of its time in the second. The links in the report take the developer directly to these loops in the source code. Figure 4 presents the linked source code containing the second parallelized loop. QuickStep inserts the `#pragma` at line 1688 to parallelize the loop.

The report indicates that each parallel loop has a single applied replication transformation (the call to `vecAdd` at line 1159 of Water.C was replaced with a call to the replicated version `vecAddRepl` of this operation and the call to `addval` at line 1644 of Water.C was replaced with a call to the replicated version `addvalRepl` of this operation), no private variables, and no applied synchronization transformations (other parallel versions have applied synchronization transformations). Figure 5 presents the linked source code containing the transformed call site (line 1161) where QuickStep replaced the call to `vecAdd` with a call to `vecAddRepl`.

The report indicates that the `interf` loop uses the `modulo` scheduling policy while the `poteng` loop uses the `dynamic` loop scheduling policy. The Final Metrics section of the page presents speedup and distortion results from the final statistical accuracy test. Note that only 5 of the 1606 executions exceeded the accuracy bound 0.003. Clicking on the green + sign (following the Final Metrics title) opens up a table with results from the individual executions.

The All Parallelizations section summarizes the search process. A single graph plots the speedup (left Y axis) and distortion (right Y axis) as a function of the different parallel versions of the program as a function of the revision number (each time QuickStep generates a new parallel program, it assigns the new version of the program the next available revision number).

Parallelization 2

Speedup: **1.629 (16.450s / 10.103s)** on 8 cores | Distortion: **0.002900** (bound 0.003) | Total Run Time: 252.582s

Transformation

Loop Parallelization	Privatization	Scheduling	Synchronization	Replication
water.C:1686 (62.6%)		static		

Metrics

Run	Run Time (s)	Speedup	Distortion
1	10.816	1.521	0.002700
2	10.156	1.620	0.002830
3	10.136	1.623	0.002650
4	10.166	1.618	0.002750
5	10.146	1.621	0.002900
6	9.947	1.654	0.003130
7	10.186	1.615	0.002900
8	10.067	1.634	0.002820
9	9.887	1.664	0.002870
10	10.047	1.637	0.002810
11	10.067	1.634	0.002780
12	9.887	1.664	0.003320
13	9.997	1.646	0.003000
14	9.977	1.649	0.002620
15	9.947	1.654	0.002700
16	10.077	1.633	0.003250
17	10.216	1.610	0.002850
18	10.057	1.636	0.002990
19	10.096	1.629	0.002830
20	10.206	1.612	0.002910
21	10.226	1.609	0.002880
22	10.037	1.639	0.002910
23	10.047	1.637	0.002910
24	10.027	1.641	0.003170
25	10.166	1.618	0.003010
Avg.	**10.103**	**1.629**	**0.002900**

Figure 6: QuickStep Parallelization 2 Report

The final section contains links to similarly formatted pages that summarize the applied parallelization transformations, performance results, and distortion results for the full set of parallel programs that QuickStep visited (starting with Parallelization 1) as it explored the parallel program search space.

2.3.2 Interactive Report Navigation

We anticipate that developers may navigate the interactive reports as follows. They may first direct their attention to the header information on the start page to obtain an overview of the performance and accuracy of the final parallelization. They may next use the links in the page to locate the parallel loops and synchronization and replication transformation sites (navigating to pages such as those presented in Figures 4 and 5). They may then use their own source code navigation system to explore relevant parts of the source code of the application and determine if the parallelization is acceptable for all inputs (and not just the representative inputs provided by the developer).

The developers may also examine the pages for other parallelizations to obtain a better understanding of the characteristics of these parallelizations. For example, the developer may examine versions that use synchronization transformations instead of replication transformations or different loop schedulers to better understand the performance consequences of these alternatives. The developer may also browse through the set of parallelizations with best performance.

The developer accesses the reports for different parallelizations by clicking on the corresponding identifiers in the All Parallelizations Table in the start page. Figure 6, for example, presents the report for Parallelization 2 (the initial parallelization of the loop on line 1686 of Water.C, which accounts for 62.6% of the executed instructions). The report presents the mean speedup, mean distortion, and total execution time required to perform the trial executions. The developer can also use the links to access the transformed source code locations. The Metrics table contains data obtained on individual test executions, including the execution time, speedup

Parallelization 3

Speedup: **1.021 (16.450s / 16.117s)** on 8 cores | Distortion: **0.000063** (bound 0.003) | Total Run Time: 693.033s

Transformation

Loop Parallelization	Privatization	Scheduling	Synchronization	Replication
water.C:1686 (62.6%)		static	water.C:1644 (addval => addvalSync)	

Metrics

Run	Run Time (s)	Speedup	Distortion
1	16.248	1.012	0.000070
2	16.198	1.016	0.000080
3	16.158	1.018	0.000020
4	16.188	1.016	0.000050
5	16.039	1.026	0.000050
6	16.128	1.020	0.000100
7	16.388	1.004	0.000090
8	16.118	1.021	0.000050
9	16.069	1.024	0.000060
10	16.118	1.021	0.000040
11	15.849	1.038	0.000090
12	16.178	1.017	0.000080
13	16.378	1.004	0.000040
14	16.098	1.022	0.000050
15	16.029	1.026	0.000070
16	16.118	1.021	0.000060
17	16.178	1.017	0.000070
18	16.428	1.001	0.000060
19	16.248	1.012	0.000030
20	16.188	1.016	0.000110
21	16.248	1.012	0.000070
22	16.228	1.014	0.000070
23	16.238	1.013	0.000080
24	16.168	1.017	0.000060
25	16.168	1.017	0.000070
26	16.218	1.014	0.000030
27	15.859	1.037	0.000140
28	16.118	1.021	0.000080
29	16.019	1.027	0.000060
30	15.909	1.034	0.000070
31	16.118	1.021	0.000060
32	16.078	1.023	0.000040
33	16.208	1.015	0.000050
34	15.869	1.037	0.000060
35	16.138	1.019	0.000070
36	16.058	1.024	0.000010
37	16.009	1.028	0.000010
38	16.009	1.028	0.000040
39	15.969	1.030	0.000090
40	16.029	1.026	0.000090
41	16.078	1.023	0.000070
42	15.929	1.033	0.000090
43	16.019	1.027	0.000030
Avg.	**16.117**	**1.021**	**0.000063**

Figure 7: QuickStep Parallelization 3 Report

Parallelization 4

Speedup: **1.589 (16.450s / 10.359s)** on 8 cores | Distortion: **0.000325** (bound 0.003) | Total Run Time: 445.425s

Transformation

Loop Parallelization	Privatization	Scheduling	Synchronization	Replication
water.C:1686 (62.6%)		static		water.C:1644 (addval => addvalRepl)

Metrics

Run	Run Time (s)	Speedup	Distortion
1	10.266	1.602	0.000290
2	10.186	1.615	0.000240
3	10.186	1.615	0.000550
4	10.206	1.612	0.000400
5	10.186	1.615	0.000330
6	10.206	1.612	0.000330
7	10.506	1.566	0.000380
8	10.576	1.555	0.000320
9	10.246	1.605	0.000390
10	10.596	1.552	0.000330
11	10.566	1.557	0.000350
12	10.446	1.575	0.000400
13	10.516	1.564	0.000460
14	10.556	1.558	0.000230
15	10.646	1.545	0.000320
16	10.616	1.550	0.000370
17	10.556	1.558	0.000250
18	10.446	1.575	0.000280
19	10.536	1.561	0.000240
20	10.556	1.558	0.000280
21	10.566	1.557	0.000230
22	10.216	1.610	0.000240
23	10.586	1.554	0.000370
24	10.945	1.503	0.000280
25	10.256	1.604	0.000420
26	10.176	1.616	0.000310
27	10.246	1.605	0.000350
28	10.186	1.615	0.000290
29	10.506	1.566	0.000130
30	10.186	1.615	0.000360
31	10.186	1.615	0.000360
32	10.266	1.602	0.000250
33	10.186	1.615	0.000360
34	10.396	1.582	0.000440
35	10.256	1.604	0.000240
36	10.206	1.612	0.000350
37	10.206	1.612	0.000270
38	10.206	1.612	0.000440
39	10.216	1.610	0.000410
40	10.196	1.613	0.000120
41	10.226	1.609	0.000310
42	10.206	1.612	0.000340
43	10.196	1.613	0.000360
Avg.	**10.359**	**1.589**	**0.000325**

Figure 8: QuickStep Parallelization 4 Report

and distortion for each execution. For Parallelization 2, even though the mean distortion is 0.0029, which is less than the accuracy bound b=0.003, the results from the individual test executions exceed the bound frequently enough to cause QuickStep to reject the parallelization.

Based on an analysis of the memory profiling information, QuickStep next applies a synchronization transformation to the densest potential source of data races, a call (at line 1644 of Water.C) to `vecAdd` of the class `vector`. Figure 7 presents the report for this parallelization. A comparison of the reports from Parallelizations 2 and 3 indicates that this transformation significantly decreased the distortion, but also significantly decreased the performance. The next parallelization (Parallelization 4, see Figure 8) replaces the synchronization with a replication, preserving acceptably accurate execution and restoring good performance.

Figures 9 and 10 present the second (Parallelization 11) and third (Parallelization 16) best parallelizations. By comparing the Transformation tables from the best (Parallelization 10) and second best (Parallelization 11) parallelizations, the developer can see that the difference between the parallelizations is the scheduling policy for the loop at line 1146 of Water.C — the best parallelization uses the `dynamic` policy, while the second best uses the `modulo` policy. A similar comparison (as well as using the links to examine the corresponding source code for the parallelized loops) indicates that the difference between the best (Parallelization 10) and third best (Parallelization 16) parallelizations is that the best parallelizes the outer loop at line 1146 of Water.C while the third best parallelizes the nested loop at line 1156 of Water.C.

3. Analysis and Transformation

QuickStep is structured as a source-to-source translator that augments the original sequential program with OpenMP directives to obtain a parallel program. The QuickStep analysis phases use the LLVM compiler infrastructure [20].

3.1 Loop Profiler

The QuickStep loop profiler produces an instrumented version of the sequential program that, when it executes, counts the number of times each basic block executes. The instrumentation also maintains a stack of active nested loops and counts the number of (LLVM bit code) instructions executed in each loop, propagating the instruction counts up the stack of active nested loops so that outermost loops are credited with instructions executed in nested loops.

The loop profiler produces two outputs. The first is a count of the number of instructions executed in each loop during the loop profiling execution. The second is a directed graph that captures the dynamic nesting relationships between different loops (note that the loops may potentially be in different procedures).

Parallelization 11

Speedup: **6.836 (16.450s / 2.415s)** on 8 cores | Distortion: **0.002068** (bound 0.003) | Total Run Time: 103.851s

Transformation

Loop Parallelization	Privatization	Scheduling	Synchronization	Replication
water.C:1146 (36.5%)		modulo		water.C:1159 (vecAdd => vecAddRepl)
water.C:1686 (62.6%)		modulo		water.C:1644 (addval => addvalRepl)

Metrics

Run	Run Time (s)	Speedup	Distortion
1	3.445	4.774	0.001590
2	2.367	6.950	0.002150
3	2.357	6.980	0.002230
4	2.397	6.863	0.001930
5	2.387	6.892	0.002070
6	2.357	6.980	0.001840
7	2.587	6.360	0.002250
8	2.357	6.980	0.001970
9	2.367	6.950	0.002240
10	2.377	6.921	0.001890
11	2.367	6.950	0.002180
12	2.367	6.950	0.002090
13	2.367	6.950	0.002110
14	2.367	6.950	0.001840
15	2.367	6.950	0.002120
16	2.407	6.835	0.002510
17	2.417	6.807	0.002220
18	2.357	6.980	0.002340
19	2.387	6.892	0.001950
20	2.367	6.950	0.002210
21	2.447	6.723	0.001910
22	2.367	6.950	0.001990
23	2.367	6.950	0.002370
24	2.377	6.921	0.001940
25	2.367	6.950	0.002340
26	2.367	6.950	0.002150
27	2.377	6.921	0.002460
28	2.427	6.779	0.001880
29	2.417	6.807	0.002110
30	2.377	6.921	0.001960
31	2.377	6.921	0.002230
32	2.686	6.123	0.001830
33	2.377	6.921	0.002130
34	2.407	6.835	0.001640
35	2.367	6.950	0.002020
36	2.387	6.892	0.002110
37	2.377	6.921	0.001980
38	2.367	6.950	0.001900
39	2.367	6.950	0.002070
40	2.377	6.921	0.002260
41	2.377	6.921	0.001920
42	2.367	6.950	0.002080
43	2.367	6.950	0.001910
Avg.	**2.415**	**6.836**	**0.002068**

Figure 9: QuickStep Parallelization 11 Report

Parallelization 16

Speedup: **6.418 (16.450s / 2.565s)** on 8 cores | Distortion: **0.002174** (bound 0.003) | Total Run Time: 110.282s

Transformation

Loop Parallelization	Privatization	Scheduling	Synchronization	Replication
water.C:1156 (36.5%)		modulo		water.C:1159 (vecAdd => vecAddRepl)
water.C:1686 (62.6%)		modulo		water.C:1644 (addval => addvalRepl)

Metrics

Run	Run Time (s)	Speedup	Distortion
1	2.587	6.360	0.002290
2	2.577	6.385	0.001720
3	2.537	6.485	0.002540
4	2.537	6.485	0.002240
5	2.537	6.485	0.002060
6	2.537	6.485	0.002260
7	2.547	6.460	0.002310
8	2.676	6.146	0.002170
9	2.557	6.434	0.002150
10	2.587	6.360	0.002150
11	2.537	6.485	0.002140
12	2.577	6.385	0.002080
13	2.537	6.485	0.002340
14	2.537	6.485	0.002430
15	2.537	6.485	0.002090
16	2.537	6.485	0.002180
17	2.547	6.460	0.002100
18	2.696	6.101	0.001880
19	2.537	6.485	0.002320
20	2.537	6.485	0.002330
21	2.736	6.012	0.002210
22	2.537	6.485	0.002210
23	2.527	6.511	0.002270
24	2.527	6.511	0.001960
25	2.527	6.511	0.001810
26	2.527	6.511	0.002370
27	2.537	6.485	0.002360
28	2.537	6.485	0.002420
29	2.537	6.485	0.001810
30	2.587	6.360	0.002130
31	2.826	5.820	0.002650
32	2.537	6.485	0.002170
33	2.577	6.385	0.002120
34	2.537	6.485	0.002380
35	2.537	6.485	0.002370
36	2.537	6.485	0.002000
37	2.527	6.511	0.002080
38	2.527	6.511	0.002300
39	2.686	6.123	0.001880
40	2.567	6.409	0.002250
41	2.527	6.511	0.001930
42	2.537	6.485	0.001960
43	2.537	6.485	0.002060
Avg.	**2.565**	**6.418**	**0.002174**

Figure 10: QuickStep Parallelization 16 Report

3.2 Memory Profiler

The QuickStep memory profiler produces an instrumented version of the sequential program that, when it executes, generates a trace of the addresses that it reads and writes. Given the memory profiling information for a given loop, the memory profiler computes the *interference density* for each store instruction s, i.e., the sum over all occurrences of that store instruction s in the trace of the number of store instructions p in parallel iterations that write the same address. Conceptually, the interference density quantifies the probability that a store instruction p in a parallel iteration will interfere with a sequence of (otherwise atomic) accesses that completes with the execution of the store instruction s. The interference density is used to prioritize the application of the synchronization and replication transformations, with the transformations applied first to operations that contain store instructions with the higher interference density. QuickStep is currently configured to apply synchronization introduction transformations only to loops in which all writes to a given address are preceded by a corresponding read from that address. The goal is to apply the transformation only within loops that access the object exclusively with atomic read/write updates (and not other accesses such as initializations).

```
int t;
#pragma omp parallel for private(t) schedule(static)
for (i = 0; i < n; i++) { t = a[i]+b[i]; c[i] = t;  }
```

Figure 11: OpenMP Parallel Loop Directives

3.3 Parallelization and Privatization

Figure 11 presents an example that illustrates how QuickStep uses OpenMP directives to specify parallel loops. This `parallel for` OpenMP directive causes the iterations of the loop to execute in parallel. When the parallel loop executes, it uses the `static` scheduler (we discuss available schedulers below in Section 3.5) to schedule its iterations onto the underlying parallel machine. Each iteration of the loop is given its own private version of the variable `t` so that its accesses to `t` do not interfere with those of other parallel iterations.

QuickStep uses a simple intraprocedural dataflow analysis to determine which variables to privatize. This analysis finds all local scalar variables that iterations of the parallel loop first write, then read. The parallelization transformation inserts all such variables into the `private` clause of the OpenMP loop parallelization directive.

3.4 Synchronization and Replication Transformations

QuickStep's synchronization and replication transformations operate on objects that provide multiple implementations of standard operations. Some operations are synchronized for atomic execution in parallel environments. Others operate on thread-local replicas of the object state (with the replicas coalesced when appropriate). QuickStep can work with any object that provides synchronized or replicated methods. We have implemented a QuickStep component that, given a class, automatically generates an augmented class that contains synchronized and replicated versions of the methods in the class. This component performs the following steps:

- **Synchronization:** It augments instances of the class with a mutual exclusion lock, then generates synchronized versions of each method. Each synchronized method acquires the lock, invokes the standard (unsynchronized) version of the method, then releases the lock.
- **Replication:** It augments instance of the class with an array of replicas. Each array element contains a replica of the state of the object. There is one array element (and therefore one replica) for each thread in the parallel computation. It also generates the replicated version of each method (this version accesses the local replica).
- **Caching:** It modifies the original versions of the methods to check whether the object state is distributed across multiple replicas and, if so, to combine the replicas to obtain a single cached version of the state. The method then operates on this cached version. The specific combination mechanism depends on the object. In our current system the automatically generated code simply adds up the values in the replicas. QuickStep, of course, supports arbitrary combination mechanisms.
- **Padding:** It generates additional unused memory (pads) to separate the replicas so that they fall on different cache lines. The goal is to eliminate any false sharing [9] that might otherwise degrade the performance.

Quickstep can work with instances of any class that use any mechanism to obtain alternate implementations of standard operations. So a developer could, for example, provide implementations that use atomicity mechanisms such as transactional memory or other lock-free synchronization mechanisms [18].

As it searches the space of parallel programs, the QuickStep search algorithm may direct the QuickStep compiler to perform synchronization or replication introduction transformations. The QuickStep compiler implements each such transformation by modifying the corresponding call site to invoke the appropriate synchronized or replicated version of the method instead of the original version.

3.5 Loop Scheduling

OpenMP supports a variety of loop scheduling policies via the `schedule` clause of the `parallel for` directive [25]. In general, these policies may vary in the cache locality, load balancing, and scheduling overhead that they elicit from the loop, with different policies working well for different loops.

QuickStep currently searches three loop scheduling policies: `dynamic` (when a thread goes idle, it executes the next available loop iteration), `static` (each thread executes a single contiguous chunk of iterations, with the chunks assigned at the start of the parallel loop), and `static,1` (each thread executes a discountiguous set of iterations, with the iterations assigned to threads at the start of the parallel loop in a round-robin manner). The parallelization reports refer to the `static,1` scheduling strategy as `modulo` scheduling.

4. Statistical Accuracy Test

QuickStep uses the Wald sequential probability ratio (SPR) test [40, 43] as the foundation of its statistical accuracy test. The SPR test works with a sequence of independent, identically-distributed (IID) Bernoulli variables $X_i \in \{0, 1\}$ from a distribution with unknown probability of success $p = Pr(X_i = 1)$. In this section we describe the details of the SPR test and its application to determining the program reliability.

4.1 Model

Ideally, QuickStep's statistical accuracy test should select all parallelizations with a high enough probability of producing an acceptable result ("good" parallelizations), and reject all other parallelizations ("bad" parallelizations). In practice, however, the statistical accuracy test may encounter an unlikely sequence of samples that cause it to make an incorrect decision (either rejecting a good parallelization or accepting a bad parallelization). We therefore consider the following four probabilities: P_G is the probability that the test will correctly accept a good parallelization, $1 - P_G$ is the probability that the test will incorrectly reject a good parallelization, P_B is the probability that the test will incorrectly accept a bad parallelization, $1 - P_B$ is the probability that the test will correctly reject a bad parallelization. Together, P_G and P_B completely characterize the risks associated with making a wrong decision. Ideally, P_G should be close to 1 and P_B should be close to 0.

QuickStep's statistical accuracy test works with a candidate parallelized application (QuickStep automatically generates this parallelization), an accuracy bound b, and an accuracy metric (see Section 1.3). The test repeatedly executes the parallelized application; for each execution, the accuracy metric produces a distortion d_i. We obtain the IID Bernoulli variables X_i for the SPR test by comparing the distortions d_i to the bound b:

$$X_i = \begin{cases} 1 & \text{if } d_i \leq b \\ 0 & \text{if } d_i > b \end{cases}$$

4.1.1 Reliability Tolerances

The probability p that the program will produce an acceptable result (i.e., $Pr(X_i = 1) = p$) is (in general, forever) unknown — the test uses sampling to reason about the relationship between p and a user-provided reliability goal r, but (in general), will never obtain complete information about p. The test therefore works with a tolerance around r within which it may give arbitrary answers. Specifically, the user provides a lower reliability interval ϵ_0 and an upper reliability interval ϵ_1. Together, r, ϵ_0, and ϵ_1 define the lower target probability $p_0 = r - \epsilon_0$ and the upper target probability $p_1 = r + \epsilon_1$. QuickStep's statistical accuracy test identifies parallelizations for which p is, with high probability, at least as large as p_1. We model this decision as a hypothesis testing problem with composite hypotheses $H_0 : p \leq p_0$ (i.e., the probability that the application will produce an acceptable result is at most p_0) and $H_1 : p \geq p_1$ (i.e., the probability that the application will produce an acceptable result is at least p_1). If the test accepts H_0, QuickStep will reject the parallelization. If the test accepts H_1, QuickStep will accept the parallelization. Since we typically view incorrectly rejecting a good parallelization as more acceptable than incorrectly accepting a bad parallelization, we typically use a one-sided test in which $\epsilon_0 = 0$.

4.1.2 Approximate True and False Positive Bounds

In general, no statistical hypothesis testing algorithm that relies on sampling can be perfect — any such algorithm may encounter an unlikely sequence of samples that causes it to incorrectly accept or reject a given hypothesis. The accuracy test therefore works with an approximate false positive bound α and an approximate true

positive bound β. As discussed below in Section 4.2, α and β bound the probability that QuickStep will incorrectly accept or reject H_0 and H_1.

4.1.3 IID Executions

The statistical accuracy test assumes that test executions of the parallel application are independent and identically distributed. To promote the validity of this assumption, we ensure that executions do not reuse results from previous executions and that no two executions run at the same time.

We note that, in the presence of execution schedule dependent phenomena such as data races, changing aspects of the underlying execution environment (such as the computational load, thread scheduler, number of cores or other hardware characteristics) may change the probability that an execution of the program will satisfy the accuracy bound b. So the statistical accuracy tests are valid only for the specific execution environment in which they were performed.

4.2 Sequential Hypothesis Testing

The SPR test performs n trials to obtain n Bernoulli variables X_i, $1 \leq i \leq n$. The number of trials n is determined by the values of the Bernoulli variables — the test terminates as soon as it has observed enough trials to accept or reject the relevant hypotheses. The test algorithm is therefore iterative and requires a single new observation at each step.

We denote the n Bernoulli variables as a single n-dimensional random variable $X = (X_1, X_2, \ldots X_n)$. At each step the algorithm calculates the ratio between two interpretations of the variables $X_1, X_2, ...X_n$ it has observed so far:

$$L(X) = \log \frac{Pr(X|H_1)}{Pr(X|H_0)} \tag{1}$$

The test compares the log likelihood ratio $L(X)$ with two values A and B ($B < A$). It stops and accepts hypothesis H_1 if $L(X) \geq A$. If $L(x) \leq B$, the test stops and accepts hypothesis H_0. If $B < L(X) < A$, it does not stop. It instead continues on to the next iteration, performs the next trial, and updates the log likelihood ratio $L(X)$.

For a single execution, $Pr(X_i|H_0) = Pr(X_i|p \leq p_0) \leq p_0$, and $Pr(X_i|H_1) = Pr(X_i|p \geq p_1) \geq p_1$. The function $L(X_i)$ has a minimum value when $Pr(X_i|H_0) = p_0$ and $Pr(X_i|H_1) = p_1$. Using these equalities will provide the most conservative decision by the test.

Since each random variable X_i represents an independent outcome of the parallel program execution, then $Pr(X|H_k) = \prod_{i=0}^{n} Pr(X_i|H_k) = (1-p_k)^f p_k^{n-f}$ ($k \in \{0,1\}$), where f is the number of executions that failed to produce an acceptable result, and $n - f$ is the number of executions that produced acceptable result. The formula for the log likelihood ratio becomes:

$$L(X) = (n-f)\log\frac{p_1}{p_0} + f\log\frac{1-p_1}{1-p_0} \tag{2}$$

The values A and B used as stopping criterion for the test, are related to the risks of test making a wrong decision. Specifically, they are related to the probability of accepting a good parallelization, $P_G = Pr[H_1|H_1]$, and the probability of accepting a bad parallelization, $P_B = Pr[H_1|H_0]$. α and β determine P_G and P_B as follows:

$$\frac{P_B}{P_G} \leq \frac{\alpha}{\beta} \leq e^{-A} \tag{3}$$

$$\frac{1-P_G}{1-P_B} \leq \frac{1-\beta}{1-\alpha} \leq e^{B} \tag{4}$$

To find suitable values for A and B, Wald showed that it suffices to set $A = \log\frac{\beta}{\alpha}$ and $B = \log\frac{1-\beta}{1-\alpha}$, which are computationally simplest solutions. In that case, since $P_B, P_G \in [0,1]$, $P_B \leq \frac{P_B}{P_G}$ and $1 - P_G \leq \frac{1-P_G}{1-P_B}$, the inequalities (3) and (4) become:

$$P_B \leq \frac{\alpha}{\beta} \tag{5}$$

$$1 - P_G \leq \frac{1-\beta}{1-\alpha} \tag{6}$$

From these inequalities we can conclude that in general case the values of α and β are not equal to P_B and P_G when A and B have the values described in previous paragraph. However, it is possible to use inequalities (5) and (6) to calculate the appropriate values of parameters α and β that yield the desired probabilities P_B and P_G. For example, if $\alpha = 0.091$ and $\beta = 0.91$, then $P_G \geq 0.9$ and $P_B \leq 0.1$. As a special case, when $\alpha \to 0$, and $\beta \to 1$, the bounds can be approximated to $P_B \leq \alpha$ and $\beta \leq P_G$. In that case, the user selected bounds are approximate representatives of P_G and P_B. For example, if $\alpha = 0.01$ and $\beta = 0.99$, $P_G = 0.9898\ldots$ (which is very close to β), and $P_B = 0.0101\ldots$ (which is very close to α).

4.3 Precision/Time Trade-offs

The values of α, β, p_0, and p_1, together with the (at the start of the test) unknown number of failed executions f determine the number of iterations required for the SPR test to terminate.

Higher accuracy requires more observations, i.e., more executions of the parallel application, for the SPR test to accept or reject the parallelization. The equation (2) suggests that the terms of $L(X)$ don't contribute equally to the final value when the probabilities p_0 and p_1 close to 1. The coefficient $\log\frac{1-p_1}{1-p_0}$ will have significantly larger absolute value than $\log\frac{p_1}{p_0}$. The number of failed executions encountered until that time, f, will thus have a dominant impact on the number of required executions, which implies that the test can decide on clearly unacceptable parallelizations after fewer runs. On the other hand, more trials will be required for the test to accept a parallelization, especially if the value of p is close to p_1, i.e. if $f \neq 0$.

For example, if $p_0 = 0.9$, $p_1 = 0.95$, $\alpha = 0.091$ and $\beta = 0.91$, the SPR test requires 43 executions in order to accept the parallelization if all results are acceptable. On the contrast, only 4 executions are required to reject the parallelization if all results are unacceptable. If there is one execution that produced unacceptable result, the SPR test will require 57 executions before accepting the parallelization. For stronger bounds, $p_0 = 0.99$, $p_1 = 0.995$, $\alpha = 0.01$ and $\beta = 0.99$, the parallelization is accepted after performing at least 913 executions and rejected after performing at least 7 executions. Accepting a parallelization when a single failed execution is observed will require 1051 executions in total.

In general, it is necessary to make a trade-off between the accuracy of the statistical test and the time required to perform the test. QuickStep uses two sets of accuracy bounds to improve the speed of exploration. The bounds are relaxed during the intermediate steps in the search space exploration, resulting in a significantly smaller number of required executions, but also a larger probability of an incorrect decision. QuickStep performs the final accuracy test with stronger accuracy bounds, and therefore a smaller probability of an incorrect decision. This design promotes a more rapid exploration of the search space at the potential cost of the search producing a final parallelization that the final test rejects (in which case the final test moves on to try the next best parallelizations until it finds a parallelization that it can accept).

During the parallelization space search we set $r = 0.90$, $\epsilon_0 = 0$, $\epsilon_1 = 0.05$, $\alpha = 0.091$, and $\beta = 0.91$. The probability that the SPR test will accept a good parallelization is $P_G \geq 0.9$ and probability

that it will accept a bad parallelization is $P_B \leq 0.1$. During the final accuracy test we set $r = 0.99$, $\epsilon_0 = 0$, $\epsilon_1 = 0.005$, $\alpha = 0.01$, and $\beta = 0.99$. The probabilities that the SPR test will accept good and bad parallelizations, are, respectively $P_G \geq 0.99$ and $P_B \leq 0.01$.

4.4 The Hoeffding Inequality

In previous work [21] we used the Hoeffding inequality [19] to determine if a candidate parallelization satisfies a desired statistical accuracy bound. The test performs test executions to obtain an estimator $\hat{p}$ of the unknown probability p that the parallelization will produce an acceptably accurate result. It works with a desired precision $\langle \delta, \epsilon \rangle$ and performs enough test executions so that it can use the Hoeffding inequality if $Pr(\hat{p} > p + \epsilon) \leq \delta$. Unlike the SPR test, the number of test executions is fixed in advance as a function of δ and ϵ only — the test does not use results from completed test executions to determine if it has enough information to determine if the parallelization is acceptably accurate and terminate. In general, using the Hoeffding inequality requires the test to perform substantially more executions than the SPR test to obtain comparable statistical guarantees.

5. Parallelization Search Space Algorithm

We next present the algorithm that searches the induced space of parallel programs. The goal is to find a parallelization that maximizes performance subject to satisfying the accuracy bound. The search algorithm is structured as an algorithm that searches the parallelization space associated with a single loop and an algorithm that invokes the single-loop algorithm as it combines single-loop parallelizations into parallelizations involving multiple loops.

5.1 Parallelization Search for Individual Loops

Figure 12 presents the algorithm that searches the parallelization space associated with a single loop. This algorithm is invoked to explore the space generated when the loop is added to a set of other loops that QuickStep has already parallelized.

The algorithm takes as input a loop ℓ whose parallelization to explore, a set P of loops that are already parallelized, and a set I of information about previously explored parallelizations. Specifically, I contains, for each such parallelization, sets S and T of synchronizations and replications (respectively) applied to call sites in the parallelized loops, a set T of scheduling policies for the parallelized loops, and a mean speedup s and distortion d for the parallelization.

The algorithm searches the space of parallel programs for the candidate loop ℓ as follows:

1. **Initial Parallelization:** QuickStep first generates a parallelization that executes ℓ in parallel (with no applied synchronization or replication transformations) in the context of the previous parallelization of the loops in P (this parallelization includes the applied synchronization and replication transformations for these loops). It uses the statistical accuracy test to determine if this parallelization is acceptably accurate.

2. **Synchronization Transformations:** If the current parallelization is not acceptably accurate, QuickStep next applies synchronization introduction transformations to ℓ in an attempt to restore acceptably accurate parallel execution. It applies these transformations according to the priority order established by the memory profiling information, with operations containing higher interference-density store instructions prioritized over operations containing lower interference-density store instructions. As it applies the transformations, it builds up a set S_ℓ of call sites to apply synchronization introduction transformations. It only places a transformation into S_ℓ if improves the accuracy of the parallelization.

INPUTS

ℓ – the loop to parallelize.
P – the set of loops that were previously parallelized.
I – the set of all results.

OUTPUTS

I – updated set of parallelization results.

LOCALS

S – the set of call sites for synchronization introduction.
R – the set of call sites for replication introduction.
T – the set of loop scheduling policies.
S_ℓ – a set of synchronization introduction call sites for loop ℓ.
R_ℓ – a set of replication introduction call sites for loop ℓ.
C – a set of call sites for synchronization or replication introduction.
p – indicator of whether the parallelization is acceptable.

AUXILIARY FUNCTIONS

$\mathsf{visit}(P, S, R, T)$ – runs the program with the loops in P parallelized with loop scheduling policies T, synchronization introduction transformations applied at S, and replication introduction transformations applied at R. Returns the status (pass/fail) according to the selected statistical accuracy bounds, mean speedup and distortion from the test executions. Also generates log entries used to produce the interactive report.

$\mathsf{candidates}(\ell)$ – returns the set of candidate synchronization and replication introduction call sites, C_S and C_R for loop ℓ.

$\mathsf{next}(C)$ – the highest priority synchronization or replication introduction call site in C according to the memory profiling information.

$\mathsf{transformations}(P, I)$ – returns a set of transformations for the set of loops P from the result set I.

SEARCH-LOOP

```
S, R, T = transformations(P, I)
p, s, d = visit(P ∪ {ℓ}, S, R, T)
C_S, C_R = candidates(ℓ), C = C_S ∪ C_R, S_ℓ = ∅ B = ∅
repeat
  while b < d ∧ C ≠ ∅ do
    c = next(C)
    p′, s′, d′ = visit(P ∪ {ℓ}, S ∪ S_ℓ ∪ {c}, R, T)
    if p′ = true ∧ d′ < d
      S_ℓ = S_ℓ ∪ {c}, d = d′, s = s′, p = true
    end
    C = C \ {c}
  end
  C = S_ℓ ∩ C_R, R_ℓ = ∅
  while C ≠ ∅ do
    c = next(C)
    p′ , s′, d′ = visit(P ∪ {ℓ}, (S ∪ S_ℓ) \ {c},
                        R ∪ R_ℓ ∪ {c}, T)
    if p′ = true ∧ s < s′ ∧ d′ ≤ b
      S_ℓ = S_ℓ \ {c}, R_ℓ = R_ℓ ∪ {c}
      d = d′, s = s′, p = true
    end
    C = C \ {c}
  end
until d ≤ b or S_ℓ = ∅
foreach t in OpenMP loop schedule types do
  p′, s′, d′ = visit(P ∪ {ℓ}, S ∪ S_ℓ, R ∪ R_ℓ,
                     (T \ {⟨ℓ, _⟩}) ∪ {⟨ℓ, t⟩})
  if p′ = true ∧ s ≤ s′ ∧ d′ ≤ b
    S = S ∪ S_ℓ, R = R ∪ R_ℓ
    T = (T \ {⟨ℓ, _⟩}) ∪ {⟨ℓ, t⟩}, p = true
  end
end
if p = true then return I ∪ {(P ∪ {ℓ}, S, R, T, s, d)}
else return I
```

Figure 12: Parallelization Space Search Algorithm for One Loop

3. **Replication Transformations:** QuickStep next applies replication introduction transformations in an attempt to maximize the performance while preserving acceptably accurate execution. For each synchronization introduction call site in S_ℓ, it replaces the synchronization introduction transformation with the corresponding replication introduction transformation. As it applies the transformations, it builds up a set R_ℓ of replication introduction transformations. It only places a replication transformation into R_ℓ (and removes it from S_ℓ) if it improves performance while maintaining acceptable accuracy.

   Note that it is possible for the performance improvements associated with replication to increase the data race density within the loop. In the worst case this increase can cause the parallelization to exceed its accuracy bound. In this case the algorithm returns back to apply additional synchronization transformations (potentially followed by additional replication transformations) to restore acceptably accurate execution.

4. **Loop Scheduling:** QuickStep finally tries all of the different loop scheduling policies for the current loop ℓ. If it encounters an acceptably accurate parallelization with better performance than the previous best alternative, it accepts this new scheduling policy.

5.2 Exploring the Search Space

Figure 13 presents the algorithm for the exploration of the entire parallelization space. QuickStep prioritizes the parallelization of the most time consuming loops by traversing the dynamic loop nesting graph G. It starts from the most time-consuming loops in priority order, with the priority determined by the amount of time spent in the loop (as indicated by the loop profiling information).

QuickStep explores the graph in depth-first fashion, pruning the search space in cases (1) when the resulting parallelization is unacceptably inaccurate, (2) when parallel execution of any remaining loops cannot theoretically, according to Amdahl's law, deliver better performance than the current best parallelization, and (3) when it has successfully parallelized an outer loop, it does not attempt to parallelize any fully nested inner loops (the algorithm finds such nested inner loops by finding predominators in G; this option can be turned off). Exploring fully nested inner loops can optionally be turned on, to find a regions that can benefit from nested parallelism.

QuickStep attempts to parallelize one loop at a time from the priority list of loops, and tries to discover the optimal synchronization, replication, and scheduling policies that for the current loop when parallelized together with previously parallelized loops. At each point in time, QuickStep maintains a set of all acceptable parallelizations I. For each such parallelization I records the sets S and R of applied synchronization and replication transformations, the set T of loop scheduling policies, and the observed mean speedup s and distortion d. The algorithm updates I whenever it successfully parallelizes another loop.

When QuickStep finishes exploring the parallel program search space, it orders the accepted parallelizations according to the performance, then runs the final statistical accuracy test on the parallelizations in order until it finds the parallelization with the best performance that passes the final test. This parallelization is the final parallelization.

5.3 Interactive Parallelization Report

The report generator (written in Ruby) processes the log generated by the search algorithm, extracting the relevant information and placing this information in an SQLite database. It uses the Ruby on Rails framework to retrieve the appropriate information from the database and dynamically generate web pages that present this information to the developer as the developer navigates through the report.

INPUTS

L – the set of loops in execution time priority order.
G – the dynamic loop nesting graph.

OUTPUTS

I – set of parallelization results for the explored loops.

LOCALS

V – the set of visited loop sets .
R – the set of results for all loops.
S – the stack containing loop states to visit.

AUXILIARY FUNCTIONS

$\mathsf{successor}(\ell, G)$ – returns loops that are immediate successors of a loop ℓ in graph G.

$\mathsf{profitable}(P, L, G, I)$ – finds the theoretical speedup for the part of the graph G that can be accessed from set of loops P, and returns true if it is greater than best result in I.

$\mathsf{acceptable}(P, I)$ – returns true if there is a set of transformations for a loop set P which can produce acceptable result.

$\mathsf{filter}(L, P, G)$ – filters out the loops from the set L that are fully nested inner loops of loops in P.

$\mathsf{order}(I)$ – orders the results according to speedup and returns the set of loops and belonging transformations.

QUICKSTEP

```
V = ∅, I = ∅, S = ∅
foreach ℓ0 in L
   S ← (ℓ0, ∅)
   while S ≠ ∅ do
      S → (ℓ, P′), P = P′ ∪ {ℓ}
      if P ∈ V then continue
      V = V ∪ {P}
      if not profitable(P, L, G, I) then continue
      I = SEARCH − LOOP ( ℓ, P′, I )
      if not acceptable(P, I) then continue
      foreach ℓs in filter(L, P, G) do
         S ← (ℓs, P)
      end
   end
end
foreach P, S, R, T in order(I) do
   p, s, d = visit(P, S, R, T)
   if p = true then return (P, S, R, T, s, d)
end
return ∅
```

Figure 13: QuickStep Search Algorithm

6. Experimental Results

We used QuickStep to parallelize six scientific computations:

- **Barnes-Hut:** A hierarchical N-body solver that uses a space-subdivision tree to organize the computation of the forces acting on the bodies [3]. Barnes-Hut is implemented as an object-oriented C++ computation.
- **Search:** Search is a program from the Stanford Electrical Engineering department [11]. It simulates the interaction of several electron beams at different energy levels with a variety of solids. It uses a Monte-Carlo technique to simulate the elastic scattering of each electron from the electron beam into the solid. The result of this simulation is used to measure how closely an empirical equation for electron scattering matches a full quantum-mechanical expansion of the wave equation stored in tables. Search was originally parallelized as part of the Jade project and is implemented in C.

- **String:** String uses seismic travel-time inversion to construct a two-dimensional discrete velocity model of the geological medium between two oil wells [17]. Each element of the velocity model records how fast sound waves travel through the corresponding part of the medium. The seismic data are collected by firing seismic sources in one well and recording the seismic waves digitally as they arrive at the other well. The travel times of the waves can be measured from the resulting seismic traces. The application uses the travel-time data to iteratively compute the velocity model. String was originally parallelized as part of the Jade project [31] and is implemented in C.
- **Volume Rendering:** Volume Rendering that renders a three-dimensional volume data set for graphical display [23]. It uses ray tracing to produce a sequence of views of the volume data. Volume Rendering was originally developed as part of the SPLASH-2 project [44] and is implemented in C.
- **Water:** Water evaluates forces and potentials in a system of water molecules in the liquid state. Water is derived from the Perfect Club benchmark MDG [6] and performs the same computation. Water is implemented as an object-oriented C++ computation.
- **Panel Cholesky:** A program that factors a sparse positive-definite matrix. The columns of the matrix have been aggregated into larger-grain objects called panels. This aggregation increases both the data and task grain sizes [35]. Panel Cholesky was originally parallelized as part of the Jade project [31] and is implemented in C.

Table 1 presents the number of lines of code (excluding whitespace and comments) and the programming language used to implement each benchmark.

Application	Lines of Code	Language
Barnes Hut	1,224	C++
Search	272	C
String	864	C
Volume Rendering	2,822	C
Water	2,246	C++
Panel Cholesky	1,597	C

Table 1: Benchmark Applications

6.1 Methodology

We obtained the applications in our benchmark suite along with two representative inputs for each application. We specified appropriate accuracy requirements for each application and used QuickStep to automatically parallelize the applications, using the representative inputs to perform the required profiling and parallel executions. We performed all executions on an Intel Xeon E5520 dual quad-core machine running Ubuntu Linux, using LLVM 2.7 to compile all of the versions of the applications. All parallel executions use eight threads. We set the accuracy bound b to be 0.003 for all benchmarks. We use the statistical bounds specified in section 4.3.

6.2 Quantitative Results

Table 2 presents the quantitative results for each application. All of the numbers in this table either appear directly in the corresponding interactive report or are computed from numbers that appear in this report. The table contains one row for each combination of application and input. The first column (Best Parallel Version) contains entries of the form x of y, where x is the revision number of the final parallel version of the application (i.e., the version with the best performance out of all parallel versions that satisfy the accuracy requirement) and y is the number of revisions that QuickStep generated during its exploration of the search space. The second column (Speedup) presents the mean speedup (over all test executions) of the final version of the application when run on eight processors. The speedup is calculated as the mean execution time of the original sequential program (with no parallelization overhead whatsoever) divided by the mean execution time of the final version. The third column (Distortion) presents the mean distortion of the final version. The fourth column (Search Time) presents the total time (in minutes) required to perform all of the parallel executions during the search of the space of parallel programs. The fifth column (Check Run/Fail) shows the number of executions that were required in order to pass the statistical accuracy test for the final parallelization. Five of 1606 executions of Water failed to satisfy the accuracy bound b. For all of the other benchmarks all of the executions satisfied the accuracy bound. The sixth, and final column (Check Time) presents the total time (in minutes) required to perform the test executions during the final statistical accuracy test for the final version of the application. Comparison of the parallelization space search time and final statistical accuracy test time shows that considerable time saves are obtained by using two sets of accuracy bounds.

We note that, with the exception of String, both inputs induce identical parallelizations. As discussed further below, for String the different parallelizations produce roughly equivalent performance for both inputs and both parallelizations are acceptable. Out of all applications, Panel Cholesky was the only application for which QuickStep was not able to find a viable parallelization that increases the performance.

The table also presents data for two versions of Water — one with the standard bound $b = 0.003$; the other with the bound $b = 0.0$ (which forces the parallel version to deterministically produce the same result as the sequential version). As described further in Section 6.7 below, we attribute the lower performance with $b = 0.0$ to the additional synchronization required to eliminate the data races that otherwise introduce noise into the output.

Table 3 shows the loops that were parallelized. The first column (Loop) contains the location of the loop – source file name and line number. The second column (Parallel) presents the percentage of instructions that are executed within the loop. It represents the work that can be done in parallel. The third column (WPI) presents the amount of work that is executed within single invocation of the loop. The fourth column (Instructions) presents total number of instructions that were executed within the loop. The sixth column shows the number of times the loop was invoked. The seventh column (Synchronized/Replicated Sites) shows the number of method call sites in the code that were replaced with synchronized (the first number) or replicated (the second number) version of the method after parallelizing the loop. The results show that most of the loops that QuickStep parallelized execute significant amount of work in total and in each execution of the loop, amortizing for potential runtime library overhead.

6.3 Barnes-Hut

The representative inputs for Barnes-Hut differ in the number of bodies they simulate and the number of time steps they perform. The accuracy metric computes the relative differences between the various aspects of the state of the final system, specifically the total kinetic and potential energy and the position and velocity of the center of mass.

The report indicates that Barnes-Hut spends almost all of its time in the outer loop that iterates through the time steps in the simulation. The attempted parallelization of this loop fails because the resulting parallel computation crashes with a segmentation violation. QuickStep proceeds on to the main force computation loop,

Application	Input	Best Parallel Version	Speedup	Distortion	Search Time (min)	Check Run/Fail	Check Time (min)
Barnes-Hut	16K bodies	2 of 16	6.168	0.000	27.4	913/0	41.3
	256K bodies	2 of 16	5.760	0.000	47.4	913/0	77.6
Search	500 particles	16 of 16	7.743	0.000	63.6	913/0	33.5
	750 particles	16 of 16	7.756	0.000	92.9	913/0	50.0
String	`big`	4 of 10	7.608	0.0005	8.98	913/0	18.0
	`inv`	4 of 13	7.582	0.0005	17.1	913/0	34.3
Volume Rendering	`head`	3 of 7	6.153	0.000	23.5	913/0	52.4
	`head-2`	3 of 7	5.047	0.000	6.8	913/0	15.3
Water ($b = 0.003$)	1000 molecules	10 of 33	6.869	0.0021	180.2	1606/5	64.5
	1728 molecules	11 of 34	7.009	0.0010	228.9	913/0	43.9
Water ($b = 0.0$)	1000 molecules	12 of 35	6.183	0.0000	189.2	913/0	40.7
	1728 molecules	12 of 35	6.123	0.0000	231.5	913/0	50.1
Panel Cholesky	`TK15`	35 of 111	1.01	0.000	5.9	–	–

Table 2: Quantitative Results for Benchmark Applications

which iterates over all the bodies, using the space subdivision tree to compute the force acting on that body. This loop is nested within the time step loop; the report indicates that Barnes-Hut spends the vast majority of its time in this loop. Because there are no cross-iteration dependences in the force computation loop, QuickStep's parallelization of this loop produces a parallel program that deterministically produces the same result as the original sequential program. QuickStep proceeds on to attempt to parallelize several more loops (both alone and in combination with each other and the main force computation loop), but even after the application of replication introduction transformations in these loops it is unable to produce a parallelization that outperforms the parallelization of the main force computation loop by itself. We attribute the performance loss to unsuccessfully amortized parallelization overhead.

6.4 Search

The representative inputs for Search differ in the number of particles they trace. The accuracy metric computes the relative difference between the number of particles that scatter out of the front of the solid in the sequential and parallel computations.

The report indicates that Search spends almost all of its computation time within a single inner loop. All parallelizations that attempt to execute this loop in parallel fail because the program crashes with a segmentation fault. QuickStep also explores the parallelization of various combinations of five outer loops, each of which executes the inner loop. Together, these loops account for almost all of the computation time.

QuickStep is able to parallelize all of these loops with no applied synchronization or replication transformations. Further investigation reveals that all iterations of these loops are independent and the resulting final parallel program executes without distortion. An examination of the source code reveals that the loops are simply repeated versions of the same code and perform the same computation — each loop iterates over a set of points, with the inner loop tracing a fixed number of particles for each point to compute the number of particles that exit the front of the simulated material. The iterations of the outer loops are independent and the parallelization is therefore valid for all inputs.

The execution times are quite sensitive to the scheduling policy. Search obtains its best performance (over 7.7 on 8 cores) when all loops use the `dynamic` policy with proportionally worse performance as more loops use the `static` and `modulo` policy.

6.5 String

The representative inputs for String differ in the starting model of the geology between the two oil wells and in the number of rays that they trace through the velocity model. The accuracy metric is based on the final velocity model — specifically, it computes the mean scaled difference between corresponding components of the velocity models from the sequential and parallel computations.

The report indicates that String spends almost all of its computation time within a single outer loop. This loop implements a computation that traces rays from the set of seismic sources through the discretized velocity model of the geology between two oil wells. For each ray, String computes the time the ray takes to travel from one oil well to the other as it propagates through the velocity model. It then compares the traced travel time to an experimentally measured travel time and backprojects the difference along the path of the ray to update the velocity model.

QuickStep is able to parallelize this loop. Our examination of the source code indicates that the resulting parallel computation may contain data races — it is possible for two iterations to update the same location in the model of the geology at the same time. The fact that the distortion is small but nonzero (0.0005 for the two representative inputs) reinforces our understanding (based on an examination of the source code) that while some data races may actually occur in practice, they will occur infrequently enough so that they do not threaten the acceptable accuracy of the computation.

For each input, QuickStep is also able to successfully parallelize an additional loop. The loops are not the same, but they come from the same loop nest. The search for the `big` input finds an outer loop in the loop nest, while the search for the `inv` input `inv` finds an inner loop. In both cases, the loop nest contributes to less than 4% of the execution time. Manual inspection shows that the iterations of these loops are independent and therefore parallelizing the loops does not affect the distortion.

We executed each final parallelization on the input used to derive the other parallelization, with the number of executions determined by the statistical accuracy test. All executions on input satisfied the statistical accuracy bounds. The parallel version from the `inv` input produced a mean speedup of 7.55 and distortion of 0.0004 for the `big` input; the parallel version from the `big` input produced a mean speedup of 7.57 and distortion of 0.0005 for the `inv` input. These numbers suggest that both parallelizations are acceptable; inspection of the source code via the interactive report confirms this acceptability.

6.6 Volume Rendering

The representative inputs for Volume Rendering differ in the size of the input volume data. The accuracy metric is based on the final image that the application produces as output — specifically, it computes the mean scaled difference between corresponding pixels of the images from the sequential and parallel computations.

The report indicates that Volume Rendering spends almost all of its time in three nested loops: an outer loop that iterates over

Benchmark	Loop	Parallel	WPI	Instructions	Invocations	Synchronized/Replicated Sites
Barnes Hut 16K	barnes.C (1383)	94.40%	$7.49 \cdot 10^7$	$7.49 \cdot 10^9$	100	0/0
Barnes Hut 256K	barnes.C (1383)	94.30%	$1.81 \cdot 10^{10}$	$1.44 \cdot 10^{11}$	8	0/0
Search 500	search.c (396)	19.40%	$1.70 \cdot 10^8$	$1.19 \cdot 10^9$	7	0/0
	search.c (322)	35.60%	$1.82 \cdot 10^8$	$2.19 \cdot 10^9$	12	0/0
	search.c (351)	13.80%	$4.38 \cdot 10^8$	$2.19 \cdot 10^9$	5	0/0
	search.c (368)	28.40%	$1.75 \cdot 10^8$	$1.75 \cdot 10^9$	10	0/0
	search.c (291)	2.75%	$1.69 \cdot 10^8$	$1.69 \cdot 10^8$	1	0/0
Search 750	search.c (396)	19.40%	$2.55 \cdot 10^8$	$1.79 \cdot 10^9$	7	0/0
	search.c (322)	35.80%	$2.74 \cdot 10^8$	$3.29 \cdot 10^9$	12	0/0
	search.c (351)	13.50%	$2.48 \cdot 10^8$	$1.24 \cdot 10^9$	5	0/0
	search.c (368)	28.50%	$2.62 \cdot 10^8$	$2.62 \cdot 10^9$	10	0/0
	search.c (291)	2.75%	$2.53 \cdot 10^8$	$2.53 \cdot 10^8$	1	0/0
String big	doloop.c (750)	98.90%	$1.19 \cdot 10^{10}$	$3.57 \cdot 10^{10}$	3	0/0
	doslow.c (204)	1.12%	$2.94 \cdot 10^6$	$4.03 \cdot 10^8$	2	0/0
String inv	doloop.c (750)	96.10%	$3.68 \cdot 10^{10}$	$7.35 \cdot 10^{10}$	2	0/0
	doslow.c (206)	3.86%	$2.94 \cdot 10^6$	$2.95 \cdot 10^9$	1004	0/0
Volume Rendering head	adaptive.c (359)	98.70%	$1.21 \cdot 10^{11}$	$2.81 \cdot 10^{10}$	360	0/0
Volume Rendering head-2	adaptive.c (359)	98.70%	$7.81 \cdot 10^7$	$2.81 \cdot 10^{10}$	360	0/0
Water 1000 (b = 0.003)	water.C (1146)	36.50%	$7.51 \cdot 10^8$	$2.25 \cdot 10^{10}$	30	0/1
	water.C (1686)	62.60%	$1.25 \cdot 10^9$	$3.87 \cdot 10^{10}$	31	0/1
Water 1728 (b = 0.003)	water.C (1146)	35.60%	$2.24 \cdot 10^9$	$2.69 \cdot 10^{10}$	12	0/1
	water.C (1686)	63.80%	$3.71 \cdot 10^9$	$4.82 \cdot 10^{10}$	13	0/1
Water 1000 (b = 0.0)	water.C (1146)	36.50%	$7.51 \cdot 10^8$	$2.25 \cdot 10^{10}$	30	0/1
	water.C (1686)	62.60%	$1.25 \cdot 10^9$	$3.87 \cdot 10^{10}$	31	1/1
Water 1728 (b = 0.0)	water.C (1146)	35.60%	$2.24 \cdot 10^9$	$2.69 \cdot 10^{10}$	12	0/1
	water.C (1686)	63.80%	$3.71 \cdot 10^9$	$4.82 \cdot 10^{10}$	13	1/1
Panel Cholesky TK15	tree.c (243)	2.90%	$2.98 \cdot 10^7$	$2.98 \cdot 10^7$	1	0/0
	symb.c (352)	1.60%	$1.65 \cdot 10^7$	$1.65 \cdot 10^7$	1	0/0
	tree.c (218)	1.95%	$2.00 \cdot 10^7$	$2.00 \cdot 10^7$	1	0/0

Table 3: QuickStep Detailed Results

the views in the view sequence, a nested loop that iterates over y-axis of the current view, and an inner loop that, given a y-axis from the enclosing loop, iterates over the x-axis to trace the rays for the corresponding x,y points in the view. QuickStep first tries to parallelize the outer loop. This parallelization fails because the application crashes with a segmentation violation.

QuickStep next attempts to parallelize the nested loop (which iterates over the y-axis). This parallelization succeeds and produces a parallel computation with good performance and no distortion. QuickStep also succeeds in parallelizing the inner loop by itself, but this parallelization does not perform as well as the parallelization of the nested loop (which contains the inner loop). The loop scheduling policy has a significant effect on the performance — the `dynamic` and `modulo` strategies deliver speedups of between 5 and 6 on 8 processors, while the `static` strategy only produces a speedup of around 3 on 8 processors. An examination of the source code indicates that the ray tracing computations are independent and that the parallelization is valid for all inputs.

6.7 Water

The representative inputs for Water differ in the number of water molecules they simulate and the number of simulation steps. The accuracy metric for Water is based on several values that the simulation produces as output, specifically the energy of the system, including the kinetic energy of the system of water molecules, the intramolecular potential energy, the intermolecular potential energy, the reaction potential energy, the total energy, the temperature, and a virtual energy quantity.

The reports indicate that the vast majority of the computation time in Water is consumed by an outer loop that iterates over the time steps in the simulation. The attempted parallelization of this loop fails because the resulting distortion is much larger than the accuracy bound. As discussed in Section 2, QuickStep proceeds to successfully parallelize the outer loops in the `interf` and `poteng` computations. Both parallelizations involve the application of first synchronization introduction, then replication introduction transformations. While the final computation has some unsynchronized data races, these data races occur infrequently enough to keep the final distortion (0.002) within the accuracy bound (0.003 for this example). When running the statistical test on the input `1000`, five of the executions produce output whose distortion is over the accuracy bound. For that reason, the test needs to perform more executions to obtain the evidence it needs to verify that the likelihood of a parallel execution satisfying the accuracy bound is acceptable. Once again, the `static` scheduling policy (speedup of 5 on 8 processors) performs significantly worse than the `dynamic` and `modulo` policies (speedup of close to 7 on 8 processors).

When the accuracy bound is set to 0.0, QuickStep is able to find a parallelization that speeds up the execution of the program without producing any distortion. The results for this experiment are presented in Table 2, example Water (b=0.0). In addition to the loops and transformations for the $b = 0.003$ parallelization, QuickStep applies an additional synchronization introduction that eliminates the data races from the parallelization with $b = 0.003$. The speedup of the parallelization with bound $b = 0.0$ is approximately 6.2 (as opposed to approximately 7 with $b = 0.003$). We attribute the decreased performance to the additional synchronization required to force the parallelization to deterministically produce the same result as the original sequential program. These two parallelizations highlight the trade-off between maximizing performance and maximizing accuracy. It also highlights QuickStep's ability to produce the most appropriate parallelization for the specific bound b.

Benchmark	QuickStep				ICC -par-threshold=0				ICC -par-threshold=99			
	Loops	Speedup	Parallel	WPI	Loops	Speedup	Parallel	WPI	Loops	Speedup	Parallel	WPI
Barnes-Hut 16K	1	6.168	94.40%	$7.49 \cdot 10^7$	22/3	0.003	44.75%	41	0	-	-	-
Barnes-Hut 256K	1	5.760	94.30%	$1.81 \cdot 10^{10}$	22/3	t/o	45.03%	41	0	-	-	-
Search 500	5	7.743	99.95%	$4.38 \cdot 10^8$	13/0	0.984	0.00%	0	1/0	0.976	0.00%	-
Search 750	5	7.756	99.95%	$2.74 \cdot 10^8$	13/0	1.009	0.00%	0	1/0	0.994	0.00%	-
String big	2	7.608	100.00%	$3.57 \cdot 10^{10}$	12/4	0.058	11.74%	87	0	-	-	-
String inv	2	7.582	99.96%	$3.68 \cdot 10^{10}$	12/5	0.060	14.61%	1671	0	-	-	-
VolRend head	1	6.153	98.70%	$1.21 \cdot 10^{11}$	13/1	0.105	4.69%	90	5/0	0.991	0.73%	-
VolRend head-2	1	5.047	98.70%	$7.81 \cdot 10^7$	13/1	0.102	4.69%	90	5/0	0.976	0.73%	-
Water 1000 (*b*=0.003)	2	6.869	99.10%	$1.25 \cdot 10^9$	21/12	0.011	69.83%	264	0/0	-	-	-
Water 1728 (*b*=0.003)	2	7.009	99.40%	$3.71 \cdot 10^9$	21/12	0.011	69.97%	264	0/0	-	-	-
Water 1000 (*b*=0.0)	2	6.183	99.10%	$1.25 \cdot 10^9$	21/12	0.011	69.83%	264	0/0	-	-	-
Water 1728 (*b*=0.0)	2	6.123	99.40%	$3.71 \cdot 10^9$	21/12	0.011	69.97%	264	0/0	-	-	-
Panel Cholesky TK15	3	—	6.45%	$2.98 \cdot 10^7$	15/1	0.032	10.84%	4738	3/1	1.000	10.68%	4738

Table 4: Comparison between QuickStep and Intel icc Parallelizing Compiler

6.8 Panel Cholesky

The input for the Panel Cholesky benchmark is the TK15 matrix from the Harwell-Boeing sparse matrix collection. The accuracy metric is based on the ability of the computed factorization to successfully solve a linear equation with a known correct solution. If the norm of the computed solution differs from the known correct solution by more than a given tolerance, the computation fails the accuracy test.

The report indicates that the attempted parallelization of the most time-consuming loops in the computation fails because the resulting parallel computations fail the accuracy test. QuickStep is able to find one parallelization that satisfies the accuracy test during the exploration of the search space and (barely) improves the performance. But this parallelization fails the final acceptability test because some of the trial executions crash.

Several characteristics of this application place it well beyond the reach of QuickStep (or, for that matter, any existing or envisioned parallelizing compiler [35]). The computation groups the columns of the sparse matrix into panels (sequences of adjacent columns with identical non-zero structure). It then performs the sparse Cholesky factorization at the granularity of panels. The primary problem is that the structure of the important source of concurrency in this application (performing updates involving different panels) depends on the specific pattern of the nonzeros in the matrix. Moreover, any parallel execution must order all updates to each panel before all subsequent uses of the panel. These ordering constraints emerge from the semantics of the application — there is no loop in the computation that can be parallelized to exploit this source of concurrency. Instead, the standard approach is to use an inspector/executor approach to extract the nonzero pattern from the matrix, analyze the nonzero pattern to build a schedule of the legal parallel execution, then execute this schedule to execute the computation in parallel.

This application illustrates a limitation of the QuickStep approach (as well as the different approaches behind virtually all other parallelizing compilers). QuickStep is designed to exploit concurrency available in loops whose iterations can execute (when augmented with appropriate synchronization) in any order without producing a result that fails the accuracy test. The ordering constraints in Panel Cholesky place it beyond the reach of this approach. Note, however, that ordering constraints are *not* the same as data dependences — QuickStep is often capable of parallelizing loops with data dependences between iterations. In fact, iterations of parallelized loops in both Water and String have data dependences. But because the loop iterations commute (produce acceptably accurate results regardless of the order in which they execute), violating these data dependences does not cause these applications to produce a result that fails the accuracy test. QuickStep can therefore parallelize these applications even in the presence of data dependences between loop iterations.

6.9 Comparison With Intel icc Compiler

To provide a comparison point with standard parallelizing compiler techniques, we attempted to use the Intel icc compiler [1] to parallelize our benchmark applications. The icc compiler checks for loop-carried data dependences, parallelizing the loop only if it can prove at the compile time that no such dependence exists and the loop is a profitable target for parallelization. The developer can influence the parallelization process by selecting a threshold for the profitability of parallelizing loops. The threshold can take a value between 0 — accept any parallelizable loop, and 100 — accept only loops that have high probability of being profitable (100 is the default value). For a threshold of 100, the icc compiler must be able to determine the number of loop iterations at compile time before it will parallelize the loop [39]. For smaller thresholds, the loop iteration count can be determined at runtime.

The results in Table 4 provide insight into the differences between the QuickStep and Intel icc compilers. The table presents results for QuickStep parallelizations, Intel icc parallelizations with threshold 0 (with this threshold the icc compiler parallelizes all parallelizable loops), and Intel icc parallelizations with threshold 99 (selecting profitable loops with iteration count unknown at compile time). We also compiled the applications with icc with threshold 100. We omit the results from this experiment from the table — the icc compiler was able to recognize only one loop in entire set of benchmark applications, and that loop contributed to less than 1% of execution time of the application.

For each compiler, the first column (Loops) presents the number of the loops that the compiler parallelized. For icc we present both the total number of loops (the first number), and loops that contribute to more than 1% of the executed instructions (the second number). The second column (Speedup) presents the speedup of the parallel version over the corresponding sequential version of the program. The third column (Parallel) shows the percentage of instructions that are executed from within parallelized loops, relative to the total number of parallel instructions. The fourth column (WPI – Work per Invocation) presents the mean number of instructions that were executed within single execution of the parallel loop. Table 5 presents additional details of the loops that the Intel icc compiler at threshold 0 parallelized. This table only

Benchmark	Loop	Parallel	WPI	Instructions	Invocations
Barnes-Hut 16K	lib.h (156)	16.50%	41.0	$1.92 \cdot 10^{10}$	$4.68 \cdot 10^{8}$
	barnes.C (84)	13.80%	34.0	$1.60 \cdot 10^{10}$	$4.72 \cdot 10^{8}$
	barnes.C (72)	12.70%	35.0	$1.49 \cdot 10^{10}$	$4.26 \cdot 10^{8}$
Barnes-Hut 256K	lib.h (156)	16.60%	41.0	$2.55 \cdot 10^{10}$	$6.22 \cdot 10^{8}$
	barnes.C (84)	13.90%	34.0	$2.13 \cdot 10^{10}$	$6.26 \cdot 10^{8}$
	barnes.C (72)	13.10%	35.0	$2.01 \cdot 10^{10}$	$5.74 \cdot 10^{8}$
String big	rayspace.c (87)	2.85%	87.0	$1.03 \cdot 10^{9}$	$1.18 \cdot 10^{7}$
	rayspace.c (94)	2.85%	87.0	$1.03 \cdot 10^{9}$	$1.18 \cdot 10^{7}$
	rayspace.c (79)	2.72%	83.0	$9.83 \cdot 10^{8}$	$1.18 \cdot 10^{7}$
	rayspace.c (101)	2.46%	75.0	$8.88 \cdot 10^{8}$	$1.18 \cdot 10^{7}$
String inv	doslow.c (224)	3.76%	1671.3	$2.87 \cdot 10^{9}$	$1.72 \cdot 10^{6}$
	rayspace.c (87)	2.84%	87.0	$2.17 \cdot 10^{9}$	$2.49 \cdot 10^{7}$
	rayspace.c (94)	2.84%	87.0	$2.17 \cdot 10^{9}$	$2.49 \cdot 10^{7}$
	rayspace.c (79)	2.70%	83.0	$2.07 \cdot 10^{9}$	$2.49 \cdot 10^{7}$
	rayspace.c (101)	2.44%	75.0	$1.87 \cdot 10^{9}$	$2.49 \cdot 10^{7}$
volume Rendering head	raytrace.c (102)	3.93%	90.0	$4.56 \cdot 10^{9}$	$5.06 \cdot 10^{7}$
Volume Rendering head-2	raytrace.c (102)	3.93%	90.0	$1.12 \cdot 10^{9}$	$1.25 \cdot 10^{7}$
Water 1000	water.C (1435)	11.90%	240.9	$7.34 \cdot 10^{9}$	$3.05 \cdot 10^{7}$
	water.C (1491)	11.70%	237.9	$7.25 \cdot 10^{9}$	$3.05 \cdot 10^{7}$
	water.C (1463)	11.70%	237.7	$7.24 \cdot 10^{9}$	$3.05 \cdot 10^{7}$
	lib.h (156)	7.25%	41.0	$4.48 \cdot 10^{9}$	$1.09 \cdot 10^{8}$
	water.C (1529)	6.62%	264.3	$4.09 \cdot 10^{9}$	$1.55 \cdot 10^{7}$
	water.C (1186)	6.41%	264.3	$3.96 \cdot 10^{9}$	$1.50 \cdot 10^{7}$
	water.C (1571)	2.96%	203.0	$1.83 \cdot 10^{9}$	$9.00 \cdot 10^{6}$
	water.C (1590)	2.96%	203.0	$1.83 \cdot 10^{9}$	$9.00 \cdot 10^{6}$
	water.C (1609)	2.96%	203.0	$1.83 \cdot 10^{9}$	$9.00 \cdot 10^{6}$
	water.C (1193)	1.96%	138.9	$1.21 \cdot 10^{9}$	$8.71 \cdot 10^{6}$
	water.C (120)	1.79%	28.0	$1.11 \cdot 10^{9}$	$3.95 \cdot 10^{7}$
	water.C (1208)	1.39%	128.0	$8.59 \cdot 10^{8}$	$6.71 \cdot 10^{6}$
Water 1728	water.C (1435)	11.90%	240.9	$8.99 \cdot 10^{9}$	$3.73 \cdot 10^{7}$
	water.C (1491)	11.80%	239.1	$8.92 \cdot 10^{9}$	$3.73 \cdot 10^{7}$
	water.C (1463)	11.80%	238.4	$8.89 \cdot 10^{9}$	$3.73 \cdot 10^{7}$
	lib.h (156)	7.34%	41.0	$5.54 \cdot 10^{9}$	$1.35 \cdot 10^{8}$
	water.C (1529)	6.78%	264.0	$5.12 \cdot 10^{9}$	$1.94 \cdot 10^{7}$
	water.C (1186)	6.25%	264.0	$4.73 \cdot 10^{9}$	$1.79 \cdot 10^{7}$
	water.C (1571)	2.95%	203.0	$2.23 \cdot 10^{9}$	$1.10 \cdot 10^{7}$
	water.C (1590)	2.95%	203.0	$2.23 \cdot 10^{9}$	$1.10 \cdot 10^{7}$
	water.C (1609)	2.95%	203.0	$2.23 \cdot 10^{9}$	$1.10 \cdot 10^{7}$
	water.C (120)	1.89%	28.0	$1.43 \cdot 10^{9}$	$5.09 \cdot 10^{7}$
	water.C (1193)	1.87%	139.3	$1.41 \cdot 10^{9}$	$1.01 \cdot 10^{7}$
	water.C (1208)	1.36%	128.0	$1.03 \cdot 10^{9}$	$8.06 \cdot 10^{6}$
Panel Cholesky TK15	num.c (372)	9.87%	4738.1	$1.01 \cdot 10^{8}$	$2.14 \cdot 10^{4}$

Table 5: Intel icc Parallelizing Compiler Detailed Results

includes loops that account for more than 1% of the executed instructions.

Although QuickStep parallelizes only a few loops in every benchmark, with the exception of Panel Cholesky these loops perform almost all of the work. Specifically, for all benchmarks except Panel Cholesky, the parallelized parts of the program account for more than 90% of the total computational work.

After relaxing the requirement that loop iteration count must be known at compile time (i.e. setting the threshold to 99), icc parallelizes additional loops in some of the benchmarks. But parallelizing these loops does not increase the performance. Only one loop contributes to more than 1% of the execution time. Because the parallelized loops typically execute a only small amount of work per iteration (column WPI), the parallelization usually degrades the performance.

With threshold 0, the icc compiler parallelizes over 10 loops per application. Most of these loops contribute only in a minor way, accounting for less than 1% of the work. Moreover, these loops are not good parallelization candidates — although the compiler is able to parallelize loops that together account for (in the best case) almost 70% of the executed instructions, the amount of work per iteration is so small (see the WPI column in Table 4) that the parallelization overhead usually substantially degrades the performance. In some cases the parallelized application executes more than a factor of ten slower than the sequential application.

A manual inspection shows that many of the time-consuming loops that the icc compiler was able to parallelize iterate over arrays. False sharing effects associated with the parallelization of such loops can cause an additional slowdown.

Parallelizing such loops can cause as additional slowdown of the application due to the false sharing of array elements, which are distributed and updated in all processor caches. Running parallel versions on representative inputs reveals a considerable slowdowns. Additionally, the work performed by these loops is contained within outer loops, that are identified and successfully parallelized by QuickStep.

These results highlight the difficulty of performing the static analysis required to parallelize large outer loops using standard

techniques. There are two reasons that QuickStep did not even consider the parallelization of many of the loops that the icc compiler parallelized. First, many of these loops account for less than 1% of the executed instructions (this is the cutoff below which the QuickStep compiler does not attempt to parallelize the loop). Second, many of the loops were nested inside outer loops that QuickStep was able to successfully parallelize (in its current configuration QuickStep does not attempt to parallelize loops nested inside successfully parallelized outer loops). These results highlight the importance of parallelizing outer loops and the QuickStep's effectiveness to obtain good parallel performance by parallelizing such loops.

6.10 Comparison With Jade Parallelizations

Jade is an implicitly parallel language that developers can use to subdivide the computation into tasks and specify how each task accesses data. The Jade run-time system examines the data access specifications to dynamically execute the Jade program in parallel while preserving the data dependences between tasks (and therefore the sequential semantics of the Jade application).

Several of the benchmarks in this paper (Search, String, and Panel Cholesky) were originally parallelized as part of the Jade project [31]. Parallel versions of Water and Volume Rendering were also developed as part of the Jade project, but the Water and Volume Rendering benchmarks in this paper came from other projects. We next compare the Jade and QuickStep parallelizations.

6.10.1 Search

The Search main computation is generated by two nested loops. The QuickStep parallelization only executes the outer loop in parallel; the Jade parallelization recodes the loops to exploit parallelism available across both loops. The Jade parallelization maintains a separate copy of the data modified in the inner loop — the number of particles that exit the front of the material for each tracked point. The computation then combines the counts from each task to obtain the final counts.

The QuickStep parallelization, on the other hand, maintains a single copy of the array that stores the number of particles that exit the front of the material. The parallel loop iterations write disjoint elements of this array. Even though the Jade implementation exploits more of the available concurrency, the QuickStep parallelization exploits more than enough concurrency to obtain good parallel performance on the target hardware platform — on this platform, there is no pressing need to exploit the concurrency present in both loops.

6.10.2 String

Like Search, the String main computation is generated by two nested loops, with QuickStep parallelizing only the outer loop and the Jade parallelization exploiting concurrency available across both loops. In the Jade parallelization each task maintains a separate copy of the updated geometry model. When the tasks finish, the computation combines these models to obtain the final geometry model. The QuickStep parallelization maintains a single copy of the geometry model. The unsynchronized updates to this model from parallel loop iterations may (infrequently) interfere. These data races introduce a small amount of noise into the final result.

The Jade implementation can exploit more of the available concurrency and executes deterministically to produce the same result as the sequential version. However, the outer loop parallelized by QuickStep has more than enough concurrency for the computation to deliver good parallel performance on the target hardware platform. And the QuickStep version is acceptably accurate.

6.10.3 Volume Rendering

Recall that the vast majority of the work in Volume Rendering takes place inside two nested loops that together iterate over a two-dimensional image array. The outer loop iterates over the y-axis of the image array; the inner loop iterates over the x-axis of this array. The body of the loop computes the value for the corresponding pixel of the image array, with different iterations executing independently because they write different elements of the image array. The Jade parallelization exploits concurrency available across both loop nests. The QuickStep parallelization exploits concurrency available only across the outer loop. This outer loop parallelization provides more than enough concurrency for the computation to deliver good parallel performance on our target hardware platform.

In the QuickStep parallelization, the iterations of the parallel loop write different elements of the image array. There are therefore no data races and the parallel version deterministically produces the same result as the sequential version. In the Jade parallelization different tasks also write different elements, but the image array object granularity interacts poorly with the parallelism detection and exploitation algorithms in the Jade implementation, which computes the task dependence information at the granularity of Jade objects. Because the image array is a single Jade object, parts of tasks that specify writes to the array execute sequentially, even if they write different array elements. Potential solutions include decomposing the image array (which would require modifying those parts of the computation that access the array) or replicating the image array (which would increase the amount of memory required to execute the computation).

One Jade version uses a third alternative — each task specifies a *deferred write* to the image array during the ray tracing computation, then converts this deferred write to a write specification only when it writes the raytracing information into the image array. In this way the raytracing computations can execute in parallel, with the writes to the image array executing in the same order as in the sequential version. Another Jade version uses a fourth alternative, the *disjoint write* specification, which enables tasks that specify disjoint writes to the same object to execute in parallel. The current Jade implementation implements this specification only on shared-memory platforms and does not check that the writes are actually disjoint (it is possible to build an implementation without these limitations).

6.10.4 Water

The two most time-consuming subcomputations in Water (the `interf` and `poteng` subcomputations) are generated by two nested loops that iterate over interactions between pairs of water molecules. QuickStep parallelizes only the outermost loops of these two subcomputations (these outermost loops have more than enough concurrency to deliver good parallel performance on the target hardware platform). The Jade parallelization exploits concurrency available across both the outer and inner loops in both loop nests. It also replicates all data structures that the parallel tasks write and combines the replicas at the end of the each subcomputation.

The QuickStep parallelization, on the other hand, replicates only the scalar variables that the parallel loop iterations update. The iterations update the remaining non-replicated shared array variables efficiently without synchronization — the resulting data races occur infrequently enough so that the parallelization can still produce a statistically acceptable result. In this way the QuickStep parallelization exploits the ability of the application to tolerate infrequent data races to insert synchronization and replication only when required to ensure statistically accurate execution.

6.10.5 Panel Cholesky

The Jade parallelization of Panel Cholesky relies on Jade's ability to dynamically discover and exploit dynamic dependences between different tasks. The Jade dynamically analyzes the data access specifications to execute tasks in parallel while preserving the ordering constraints between tasks with dependences. In this way, unlike QuickStep, Jade is able to exploit the concurrency available in Panel Cholesky.

6.10.6 Discussion

Because of its support for irregular concurrency, Jade can exploit forms of concurrency that are inherently beyond QuickStep's loop-based approach. However, Jade requires the developer to insert the Jade task constructs and allocate Jade objects at a granularity that works well with the task decomposition. QuickStep, on the other hand, operates directly on sequential programs, minimizing and potentially even eliminating developer involvement. Because QuickStep parallelizations insert standard OpenMP tags into the original source code of the application, it is possible for the developer to inspect (and, if desired, modify) the final parallelization.

For our benchmarks, the Jade parallelizations often exploit parallelism available in nested loops. While it is possible for QuickStep to exploit this additional parallelism, the QuickStep parallelizations for our benchmarks exploit only outer loop parallelism — it is not necessary to exploit nested loop parallelism to obtain good parallel performance on our target hardware platform.

QuickStep uses execution profile information to guide the application of the parallelism introduction, accuracy enhancing, and performance enhancing transformations. The goal is to produce an acceptable parallelization — i.e., a parallelization with good performance and statistically accurate results. And in fact, the QuickStep Water and String parallelizations both contain infrequent data races that cause them to execute nondeterministically within acceptable statistical bounds. The Jade parallelizations, of course, execute deterministically with no data races. The Jade parallelizations of Search, String, and Water achieve this deterministic execution by replicating more data structures than the corresponding QuickStep parallelizations:

- **Search:** For Search, the replication interacts with the exploitation of nested loop parallelism. Because different iterations of the inner loop access the same array entries, exploiting nested loop parallelism would either introduce data races or require the application of synchronization or replication transformations to eliminate the data races. The QuickStep version sidesteps this issue entirely by exploiting outer-loop parallelism only.
- **String:** The QuickStep version of String maintains a single copy of the geometry model. Multiple parallel iterations of the outer loop update the same elements of this model without synchronization. The QuickStep version therefore has data races. The Jade version avoids these data races (or, strictly speaking, the sequentialization of the corresponding loop iterations) by replicating the geometry model. String illustrates how QuickStep's ability to tolerate infrequent data races makes it possible to avoid both replication and synchronization overhead for shared data structures when the resulting data races occur infrequently enough to preserve acceptably accurate execution.
- **Water:** Water exhibits a mix of characteristics — the QuickStep version of Water ensures acceptably accurate execution by replicating the updated scalar variables, but operates with only a single version of updated array variables (the data races on the updated array variables occur infrequently enough to leave the computation acceptably accurate). The Jade version ensures deterministic execution by replicating all updated variables. Water illustrates how QuickStep can productively select different synchronization and replication strategies for variables with different access patterns and characteristics. Of particular interest is the fact that it can, when appropriate, produce efficient parallelizations with infrequent unsynchronized data races.

Together, these examples illustrate how QuickStep's exploration of the search space induced by its parallelization transformations enables it to tailor its parallelization strategy to the characteristics of the application and target hardware platform at hand.

7. Threats To Validity

This paper has two basic conclusions. First, QuickStep's combination of parallelization transformations and search of the resulting induced parallel program space can produce efficient and accurate parallel programs. Second, the final parallelizations are amenable to developer evaluation for acceptability and the automatically generated interactive parallelization reports effectively facilitate this evaluation.

We identify several threats to the validity of these conclusions. First, the ability of QuickStep to find an acceptable parallelization depends on the characteristics of the application. Based on our experience with parallel applications, we believe that QuickStep will work well for most applications with loop-level parallelism [5, 44]. It would also be relatively straightforward to generalize QuickStep to include parallelism introduction transformations that would enable QuickStep to parallelize recursive divide and conquer applications [8, 36]. It is less clear how to generalize QuickStep to handle computations (such as the Panel Cholesky factorization benchmark) that require irregular concurrency generation approaches. And, of course, some sequential programs may simply not be amenable to parallelization by any means at all.

Second, for our benchmark set QuickStep produced parallel programs that are acceptable for all inputs and not just the representative inputs. This result may not generalize to other programs — QuickStep may produce parallel programs that are acceptably accurate for the representative inputs but not for other inputs. Maintaining comprehensive input suites (as is standard practice for many software development projects) can minimize the chances of this happening. The interactive reports also facilitate developer determination of the overall acceptability of the parallelizations. But it is also possible for QuickStep to produce complex parallelizations with enough parallel loops and parallelization transformations to make it difficult for developers to evaluate the acceptability of the final parallelizations.

Third, the parallelization may depend on the environment where the program executes, including hardware, compiler and the operating system. These factors can influence the outcomes of parallel program executions, including the accuracy. The same parallelization may generate acceptable results in one environment but not in another. It is therefore important to perform at least the final accuracy tests in the same environment as will be used for production executions.

Finally, we found that the automatically generated interactive parallelization reports effectively supported our acceptability evaluation. It is possible that other developers may prefer to have the information presented in another form or may not find the information useful.

8. Related Work

We discuss related work in parallelizing compilers, interactive profile-driven parallelization, statistical accuracy models for parallel computations, and unsound program transformations.

8.1 Parallelizing Compilers

There is a long history of research in developing compilers that can automatically exploit parallelism available in programs that manipulate dense matrices using affine access functions. This research has produced several mature compiler systems with demonstrated success at exploiting this kind of parallelism [7, 16, 24]. Our techniques, in contrast, are designed to exploit parallelism available in loops regardless of the specific mechanisms the computation uses to access data. Because the acceptability of the parallelization is based on an analysis of the output of the parallelized program rather than an analysis of the program itself with the requirement of generating a parallel program that produces identical output to the sequential program, QuickStep is dramatically simpler and less brittle in the face of different programming constructs and access patterns. To cite just one example, our results show that it can effectively parallelize object-oriented computations written in C++ that heavily use object references and pointers. The use of any one of these programming language features is typically enough to place the program beyond the reach of standard parallelizing compilers.

Commutativity analysis [2, 34] analyzes sequential programs to find operations on objects that produce equivalent results regardless of the order in which they execute. If all of the operations in a computation commute, it is possible to execute the computation in parallel (with commuting updates synchronized to ensure atomicity). Our techniques, in contrast, analyze the output of the parallelized program rather than program itself. The goal is to find a parallelization that produces acceptably accurate results rather than results that are identical or equivalent to those that the sequential program produces (of course, if the developer does require a parallelization that produces identical or equivalent results, QuickStep is capable of searching the space to find such a parallelization). Once again, our approach produces a dramatically simpler compiler that can successfully parallelize a broader range of programs.

Motivated by the difficulty of exploiting concurrency in sequential programs by a purely static analysis, researchers have developed approaches that use speculation. These approaches (either automatically or with the aid of a developer) identify potential sources of parallelism before the program runs, then run the corresponding pieces of the computation in parallel, with mechanisms designed to detect and roll back any violations of dependences that occur as the program executes [10, 27, 29, 41]. These techniques typically require additional hardware support, incur dynamic overhead to detect dependence violations, and do not exploit concurrency available between parts of the program that violate the speculation policy. Our approach, in contrast, operates on stock hardware with no dynamic instrumentation. It can also exploit concurrency available between parts of the program with arbitrary dependences (including unsynchronized data races) as long as the violation of the dependences does not cause the program to produce an unacceptably inaccurate result.

Another approach to dealing with static uncertainty about the behavior of the program is to combine static analysis with run-time instrumentation that extracts additional information (available only at run time) that may enable the parallel execution of the program [15, 28, 29, 38]. Once again, the goal of these approaches is to obtain a parallel program that always produces the same result as the sequential program. Our approach, on the other hand, requires no run-time instrumentation of the parallel program and can parallelize programs even though they violate the data dependences of the sequential program (as long as these violations do not unacceptably perturb the output).

8.2 Profile-Driven Parallelization

Profile-driven parallelization approaches run the program on representative inputs, dynamically observe the memory access patterns, then use the observed access patterns to suggest potential parallelizations that do not violate the observed data dependences [14, 37, 42]. The dynamic analysis may be augmented with a static analysis to recognize parallel patterns such as reductions. These potential parallelizations are typically presented to the developer for approval.

QuickStep is also designed for usage scenarios in which the developer examines the parallelizations to determine their acceptability. But QuickStep uses a fundamentally different parallelization approach — instead of attempting to preserve the data dependences, it deploys a set of parallelization strategies that enable it to explore a broad range of potential parallelizations. Specifically, QuickStep explores parallelizations that use synchronization to make parallel operations atomic, use replication to eliminate bottlenecks in the parallel execution, or contain infrequent unsynchronized data races that do not unacceptably perturb the execution. And of course, it can also parallelize simpler computations in which the iterations of the resulting parallel loops are independent.

Our experimental results show that QuickStep's broader reach is important in practice. The final parallelizations of two of the applications in our benchmark set (String and Water) contain unsynchronized data races that violate the underlying data dependences, which places these efficient parallelizations beyond the reach of any technique that attempts to preserve these dependences.

8.3 Statistical Accuracy Models for Parallel Computations

Recent research has developed statistical accuracy models for parallel programs that discard tasks, either because of failures or to purposefully reduce the execution time [32]. A conceptually related technique eliminates idle time at barriers at the end of parallel phases of the computation by terminating the parallel phase as soon as there is insufficient computation available to keep all processors busy [30]. The results are largely consistent with the results reported in this paper. Specifically, the bottom line is that programs can often tolerate perturbations in the execution (discarding tasks, reordering loop iterations, or data races) without producing unacceptable inputs. There are several differences between this previous research and the research presented in this paper. First, the goals are different: the techniques presented in this paper are designed to parallelize the program; previous techniques are designed to enable parallel programs to discard tasks or eliminate idle time. The potential performance benefits of the research presented in this paper are significantly larger (but could be enhanced by the previous techniques). Second, the statistical approaches are significantly different. Previous research uses multiple linear regression to produce a statistical model of the distortion as a function of the number of discarded tasks. The research presented in this paper, on the other hand, uses user-defined accuracy tests in combination with the SPR test to obtain a statistical guarantee of the accuracy of the resulting parallel computation. In comparison with previous approaches, this approach requires fewer assumptions on the behavior of the parallel computation but more test executions to obtain tight statistical distortion bounds.

8.4 Unsound Program Transformations

We note that this paper presents techniques that are yet another instance of an emerging class of unsound program transformations. In contrast to traditional sound transformations (which operate under the restrictive constraint of preserving the semantics of the original program), unsound transformations have the freedom to change the behavior of the program in principled ways. Previous unsound transformations have been shown to enable applications to productively survive memory errors [4, 33], code injection attacks [26, 33], data structure corruption errors [12, 13], memory leaks [22], and infinite loops [22]. The fact that all of these

techniques provide programs with capabilities that were previously unobtainable without burdensome developer intervention provides even more evidence for the value of this new approach.

9. Conclusion

The difficulty of developing parallel programs by hand has inspired the development of compilers that are designed to automatically parallelize sequential programs. During its long history, the field has achieved demonstrated successes within specific computation domains, but many computations remain well beyond the reach of traditional approaches (which statically analyze the sequential program to find tasks that can execute in parallel). The difficulty of building compilers that use these approaches and the large classes of programs that currently (and in some cases inherently) lie beyond their reach leaves room for simpler and more effective techniques that can parallelize a wider range of programs. Our experimental results indicate that QuickStep's basic approach, which involves the combination of parallelization transformations and search of the resulting induced space of parallel programs guided by test executions on representative inputs, provides both the simplicity and broader applicability that the field requires.

10. Acknowledgements

We would like to thank Dan Roy for his help with the statistical accuracy test and Stelios Sidiroglou for his inputs on the earlier drafts of this work. This research was supported by DARPA Cooperative Agreement FA8750-06-2-0189, the Samsung Corporation, and NSF Awards 0811397, 0835652, and 0905244. The line number data for Table 1 was generated using David A. Wheeler's SLOC-Count utility.

References

[1] Intel compiler. `software.intel.com/en-us/intel-compilers/`.

[2] F. Aleen and N. Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *ASPLOS 2009*.

[3] J. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324(4):446–449, Dec. 1986.

[4] E. Berger and B. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI*, June 2006.

[5] M. Berry et al. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. ICASE Report 827, CSRD, Urbana, IL, May 1989.

[6] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), Nov. 1992.

[7] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Raughwerger, P. Tu, and S. Weatherford. Effective automatic parallelization with Polaris. In *International Journal of Parallel Programming*, May 1995.

[8] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, Aug. 1996.

[9] W. Bolosky and M. Scott. False sharing and its effect on shared memory performance. In *SEDMS 1993*.

[10] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO 2007*.

[11] R. Browning, T. Li, B. Chui, J. Ye, R. Pease, Z. Czyzewski, and D. Joy. Low-energy electron/atom elastic scattering cross sections for 0.1-30keV. *Scanning*, 17(4):250–253, July/August 1995.

[12] B. Demsky, M. Ernst, P. Guo, S. McCamant, J. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA '06*.

[13] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05*, 2005.

[14] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI 2007*.

[15] Y. Ding and Z. Li. An Adaptive Scheme for Dynamic Parallelization. *Lecture notes in computer science*, pages 274–289, 2003.

[16] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, Dec. 1996.

[17] J. Harris, S. Lazaratos, and R. Michelena. Tomographic string inversion. In *Proceedings of the 60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, 1990.

[18] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA 1993*, San Diego, CA, May 1993.

[19] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[20] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO 2004*.

[21] S. Misailovic, D. Kim, and M. Rinard. Automatic Parallelization with Statistical Accuracy Bounds. Technical Report MIT-CSAIL-TR-2010-007, 2010.

[22] H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM '07*.

[23] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory MIMD architectures. Technical Report CSL-TR-92-537, Computer Systems Laboratory, Stanford Univ., Stanford, Calif., Aug. 1992.

[24] Open64. Open64: Open research compiler. `www.open64.net`.

[25] OpenMP. `www.openmp.org`.

[26] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, et al. Automatically patching errors in deployed software. In *SOSP '09*.

[27] M. Prabhu and K. Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP 2005*.

[28] L. Rauchwerger, N. Amato, and D. Padua. Run-time methods for parallelizing partially parallel loops. In *ICS*, 1995.

[29] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI 1995*.

[30] M. Rinard. Using early phase termination to eliminate load imbalancess at barrier synchronization points. In *OOPSLA 2007*.

[31] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Dept. of Computer Science, Stanford Univ., Stanford, Calif., 1994.

[32] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS 2006*, Cairns, Australia, June 2006.

[33] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, December 2004.

[34] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *TOPLAS*, 19(6), Nov. 1997.

[35] E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Dept. of Computer Science, Stanford Univ., Stanford, Calif., Jan. 1993.

[36] R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPoPP 1999*.

[37] S. Rul, H. Vandierendonck, and K. De Bosschere. A dynamic analysis tool for finding coarse-grain parallelism. In *HiPEAC Industrial Workshop*, 2008.

[38] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *ICS 2007*.

[39] O. Sachan. Performance tools for software developers - auto parallelization and /qpar-threshold. July 2009.

[40] D. Siegmund. *Sequential analysis*. Springer Verlag, 1985.

[41] P. Tinker and M. Katz. Parallel execution of sequential Scheme with Paratran. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 28–39, Snowbird, UT, July 1988.

[42] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI 2009*.

[43] A. Wald. *Sequential analysis*. John Wiley and Sons, 1947.

[44] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA 1995*.