MIT-CSAIL-TR-2010-007 February 10, 2010

Automatic Parallelization With Statistical Accuracy Bounds

Sasa Misailovic, Deokhwan Kim, and Martin Rinard

Automatic Parallelization With Statistical Accuracy Bounds

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

Traditional parallelizing compilers are designed to generate parallel programs that produce identical outputs as the original sequential program. The difficulty of performing the program analysis required to satisfy this goal and the restricted space of possible target parallel programs have both posed significant obstacles to the development of effective parallelizing compilers.

The QuickStep compiler is instead designed to generate parallel programs that satisfy statistical accuracy guarantees. The freedom to generate parallel programs whose output may differ (within statistical accuracy bounds) from the output of the sequential program enables a dramatic simplification of the compiler and a significant expansion in the range of parallel programs that it can legally generate. QuickStep exploits this flexibility to take a fundamentally different approach from traditional parallelizing compilers. It applies a collection of transformations (loop parallelization, loop scheduling, synchronization introduction, and replication introduction) to generate a search space of parallel versions of the original sequential program. It then searches this space (prioritizing the parallelization of the most time-consuming loops in the application) to find a final parallelization that exhibits good parallel performance and satisfies the statistical accuracy guarantee. At each step in the search it performs a sequence of trial runs on representative inputs to examine the performance, accuracy, and memory accessing characteristics of the current generated parallel program. An analysis of these characteristics guides the steps the compiler takes as it explores the search space of parallel programs.

Results from our benchmark set of applications show that QuickStep can automatically generate parallel programs with good performance and statistically accurate outputs. For two of the applications, the parallelization introduces noise into the output, but the noise remains within acceptable statistical bounds. The simplicity of the compilation strategy and the performance and statistical acceptability of the generated parallel programs demonstrate the advantages of the QuickStep approach.

1. Introduction

Developing parallel software is known to be a challenging and difficult task. One standard approach to simplifying this task is to develop a parallelizing compiler, which processes sequential programs to automatically generate corresponding parallel programs. The traditional correctness requirement for parallelizing compilers is that the generated parallel program must produce, for all inputs, a result that is identical to the result that the original sequential program would have produced on that same input. In practice, the difficulty of satisfying this correctness requirement has severely limited the range of applications that compilers can automatically parallelize.

We propose a different approach. Our QuickStep compiler generates a search space of candidate parallel programs, then reasons statistically about the accuracy of the outputs that these programs produce when run on representative inputs. Instead of attempting to generate a parallel program that is guaranteed to always produce the identical output as the sequential program, QuickStep instead generates a parallel program with a statistical guarantee that it will, with high likelihood, produce an output that is within user-defined accuracy bounds of the output that the original sequential program would have produced.

Note that because the programs that QuickStep generates only need to satisfy statistical accuracy bounds, QuickStep has the freedom to generate nondeterministic parallel programs (as long as these programs satisfy the desired accuracy bounds). Our current QuickStep implementation generates parallel programs with three potential sources of nondeterminism (data races, replication combined with reduction, and variations in the execution order of parallel loop iterations), but in general any parallel program, deterministic or nondeterministic, is acceptable as long as it satisfies the statistical accuracy guarantee.

1.1 Basic QuickStep Approach

QuickStep first runs the sequential program on representative inputs (provided by a user or developer). For each input, it records the result that the program produces and the execution time. It then runs an instrumented version of the program and uses the generated profiling information to find the most time-consuming loops. The compiler then uses the following steps to attempt to produce a parallel program that 1) produces sufficiently accurate results and 2) performs well:

- **Parallelization:** The compiler generates code that executes the iterations of a selected time-consuming loop in parallel. Aside from the barrier synchronization at the end of the loop, the parallel loop executes without synchronization.
- **Alternate Object Implementations:** QuickStep is designed to work with programs that are written in modern object-oriented style. The QuickStep library comes with several different implementations of classes that are commonly used in parallel programs. Our current library provides three different implementations:
    - **Unsynchronized:** The standard implementation that accesses the object state without synchronization.
    - **Synchronized:** The standard implementation augmented with mutual exclusion synchronization so that operations execute atomically in a multithreaded context.
    - **Replicated:** An implementation that replicates the state so that each thread accesses its own replica without synchronization and without contention. Updates are combined back into a single replica after parallel loops.

  Depending on the characteristics of the application, different implementations are appropriate. The unsynchronized implementation is appropriate when there are no data races or the inaccuracy introduced by any data races is acceptable. The synchronized implementation is appropriate when there is little contention on the object, but the accesses must still execute atomically for the application to produce acceptable output. The replicated implementation is appropriate when contention on the synchronized implementation would create a bottleneck with unacceptable performance degradation.

- **Implementation Selection:** When presented with a parallelization with unacceptable accuracy, QuickStep proceeds under the assumption that the inaccuracy is caused by data races. It therefore replaces unsynchronized implementations of involved objects with synchronized implementations in an attempt to eliminate the data races responsible for the unacceptable accuracy.

  When presented with a parallelization with unacceptable performance, QuickStep proceeds under the assumption that multiple threads accessing a single object concurrently create a bottleneck responsible for the unacceptable performance. It therefore replaces unsynchronized or synchronized implementations of involved objects with replicated implementations in an attempt to eliminate the bottleneck and deliver good performance.

- **Memory Profiling:** The implementation selection is driven by memory profiling. QuickStep executes an instrumented version of the program that records the memory locations that the instructions in the parallel loop access. It then analyzes this memory profiling information to find objects with data races.

  QuickStep uses the density of the races to prioritize the implementation selection. When presented with multiple races, it replaces implementations of objects with high data race density before implementations of objects with low data race density.

Together, these steps generate a space of parallel programs that the compiler searches as it attempts to find an accurate program with good parallel performance. The compiler evaluates each candidate parallel program as follows:

- **Intermediate Evaluation Runs:** It executes the program on a set of representative inputs (obtained from a user or developer). If these runs demonstrate the hoped for accuracy or performance, the compiler accepts the candidate parallel program as a starting point for further exploration (via additional parallelization, synchronization introduction, or replication transformations).

- **Final Evaluation Runs:** When the compiler has found a program with acceptable performance and accuracy and no further plausible parallelization or replication transformations, it performs the final evaluation runs. These runs execute the program repeatedly on representative inputs to obtain the statistical accuracy guarantee. If the final evaluation fails, the compiler continues to explore alternate parallelizations. Otherwise it proceeds on to the developer evaluation step.

- **Developer Evaluation:** The compiler generates a report that presents the parallel loops and object implementations in the final version. The memory profiling information is also available. The developer can then examine this information and the source code of the program to 1) determine if the parallelization is acceptable and 2) if not, guide the compiler to modify the parallelization. Because the other evaluation steps rely on testing, this step is essential in ensuring that the final parallel program will behave acceptably on all envisioned inputs (and not just the representative inputs in the test set).

The final issue is obtaining a sufficiently efficient search algorithm. Our currently implemented compiler attempts to parallelize all loops that account for at least 10% of the executed instructions. It attempts to parallelize outer loops before inner loops, attempting inner loops only if the attempt to parallelize the enclosing outer loop fails. It tries synchronized implementations before replicated implementations (in effect, searching first for accuracy, then for performance).

1.2 Results

We evaluate QuickStep by using it to automatically parallelize four applications in the Jade benchmark suite [23, 28] plus the Barnes-Hut N-body simulation [2]. Our results show that QuickStep is able to effectively parallelize all but one of these applications. The sole exception is a sparse Cholesky factorization algorithm, whose irregular concurrency pattern exhibits dynamically determined cross-iteration dependences that place it well beyond the reach of existing or envisioned parallelizing compiler technology [29]. For the remaining four applications, QuickStep is able to automatically generate parallel programs that (on a machine with four Intel Xeon quad-core processors running eight threads) execute, on average, between 3.7 and 6.4 times faster than the corresponding sequential programs. All of the automatically parallelized applications satisfy precise statistical accuracy guarantees.

Interestingly enough, two of the successfully parallelized applications contain (unsynchronized) data races. While these data races introduce some noise into the output that the parallel program produces, the noise is small and does not threaten the viability of the parallelization. In particular, the effect of any individual race on the computation is close to negligible and the races occur infrequently enough to make their aggregate effect on the computation more than acceptable. The presence of these parallelizations illustrates a key advantage of the QuickStep approach — because QuickStep evaluates the acceptability of the the generated parallel program empiricially by observing the outputs that it produces, it can exploit parallelization strategies (such as the generation of parallel loops that contain acceptable data races) that are inherently beyond the reach of traditional parallelizing compilers. This broader range of parallelization strategies can, in turn, expand the range of programs that QuickStep can effectively parallelize.

1.3 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 [27]. They can also contain unsynchronized data races, accesses to out of date versions of data (such as those in chaotic relaxation algorithms [32]), 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 unable 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.

1.4 Contributions

This paper makes the following contributions:

- **Statistical Accuracy Guarantees:** It introduces, for the first time, the use of statistical accuracy guarantees as a basis for the acceptable parallelization of sequential programs.
- **Parallelization Strategy:** It presents the first parallelization strategy built around searching an automatically generated space of parallel programs, using executions on representative inputs to evaluate the performance and accuracy of the generated parallel programs.
- **Data Races:** Enabled by the use of statistical accuracy guarantees, it introduces, again for the first time, the use of automatic parallelization strategies that can produce parallel programs with acceptable unsynchronized data races.
- **Simplicity:** It shows, once again for the first time, how an empirical evaluation of parallelized programs founded on a statistical analysis can enable the simple, effective, and automatic exploitation of parallelism available in sequential programs.
- **Results:** It presents results from our implemented QuickStep parallelizing compiler. These results show that QuickStep can effectively parallelize programs that use a range of programming constructs to produce computations that combine outstanding parallel performance with tight statistical accuracy guarantees.

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 `add` updates the `VIR` accumulator object, which stores the sum of the virtual energies of all the interactions.

QuickStep first runs the sequential computation on some representative inputs and records the running times and outputs. It next executes several profiling runs. The profiling results indicate that 63% of the execution time is spent in the `interf` outer loop. QuickStep therefore 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 a parallel program with unacceptable accuracy. Specifically, the outputs change, on average, by close to 40%. QuickStep next executes the memory profiling runs. The memory profiling information indicates that there are data races at multiple locations

```
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.add(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

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

This transformation produces a parallel program with acceptable accuracy but unacceptable performance. In fact, the parallel program takes longer to execute than the original serial program! QuickStep operates on the assumption that there is a bottleneck on the synchronized `add` updates and that this bottleneck is the cause of the poor performance. It therefore replaces the call to `Syncadd` with a call to the replicated version of the `add` method, changing the call site in `interact` to invoke the `Repladd` method. The initial test runs 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 an acceptably accurate result. Specifically, the outputs differ, on average, by less than 0.5% from the output that the serial program produces. After a similar parallelization process for the remaining time intensive loop, the outer loop in the `poteng` method (not shown), which accounts for 36% of the execution time, QuickStep obtains a final parallel program that exhibits both good performance (on 8 processors it runs 4.7 times faster than the original serial program) and accurate output (the outputs differ, on average, by less than 0.5% from the output of the serial program).

The final step is to obtain the statistical accuracy guarantee. We start with a user-defined accuracy test t. This test takes an output from the sequential program and a corresponding output from the

[1] The actual implementation 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 add(double d) {
  if (!isCached) cache();
  cachedVal = cachedVal + d;
  isCached = true;
 }
 void Syncadd(double d) {
  pthread_mutex_lock(&mutex);
  addvalUnsync(d);
  pthread_mutex_unlock(&mutex);
 }

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

Figure 2: Example Accumulator

generated parallel program executing on the same input. It produces either success (if the output from the parallel program is acceptably accurate) or failure (if the output from the parallel program is not accurate enough). Note that for any generated parallel program there is a (forever unknown) actual probability p that an execution of the program will produce an output that passes the accuracy test t. The goal is to obtain a statistically valid bound on p. We support two methods for obtaining such a bound: the Hoeffding Bound and the Wald Sequential Probability Ratio Test.

2.1 Hoeffding Bound

To use the Hoeffding Bound, we start with an accuracy goal g and a accuracy precision $\langle \delta, \epsilon \rangle$. QuickStep will produce a parallel program and estimator $\hat{p} \geq g + \epsilon$ such that $P[\hat{p} > p + \epsilon] \leq \delta$, where p is the probability that an execution of the generated parallel program will produce an output that passes the accuracy test t. This test compares the output of the sequential program and an output of the parallel program to determine if the parallel program produced an acceptably accurate output.

By repeatedly executing the parallel program to observe the percentage of executions that pass the accuracy test, it is possible to obtain an unbiased estimator $\hat{p}$ of p. Moreover, given a desired precision $\langle \delta, \epsilon \rangle$ for $\hat{p}$, it is possible to compute how many executions n are required to obtain the statistical guarantee $P[\hat{p} > p + \epsilon] \leq \delta$ for the precision of $\hat{p}$. This guarantee states that, with probability at most δ, the estimated probability $\hat{p}$ that the generated parallel program will pass the accuracy test exceeds the actual probability p by at most ϵ. Note that this framework provides a one-sided guarantee — it only bounds the amount by which the estimator $\hat{p}$ may overestimate the actual probability p, not the amount by which it may underestimate p. This kind of one-sided bound is appropriate because if users will accept a computation that is as accurate as $\hat{p} - \epsilon$, they will also accept a computation that is more accurate.

Assume that the user's accuracy test specifies an accuracy bound b of 0.01 (in other words, the outputs of the parallel and sequential computations may differ on average by at most one percent). The accuracy goal g is 0.9, and desired precision $\langle \delta, \epsilon \rangle$ is $\langle 0.95, 0.05 \rangle$. QuickStep must now compute the number of executions it must run to determine if the parallel computation satisfies the statistical accuracy guarantee $P[\hat{p} > p + \epsilon] \leq \delta$. QuickStep first applies the one-sided Hoeffding bound:

$$P[\hat{p} > p + \epsilon] \leq e^{-2n\epsilon^2}$$

Here n is the number of executions required to obtain the desired precision for $\hat{p}$. For the desired precision $\langle 0.95, 0.05 \rangle$, n is 600. QuickStep therefore runs the automatically parallelized program 600 times. In these runs the mean value of d is 0.005 and the variance is 0.0008. Because the mean is not 0, some of the potential data races in the parallel loop iterations actually occur and affect the result. But overall this effect is small. There are two reasons why the effect is small. First, because the updates in the parallel computation are distributed across a large array (the b array), the chances of any given race actually occurring are quite small. And if a race occurs, the net effect is to discard the addition of the force between two bodies into the total aggregate force acting on one of the bodies. Because there are many bodies and many interactions, discarding any one of the individual force contributions introduces only a small amount of noise into the final result.

QuickStep next applies the accuracy bound 0.01 to obtain the estimator $\hat{p}$. We performed all of the 600 statistical accuracy runs on 8 threads of an Intel Xeon machine with four quad-core processors. In our example $\hat{p}$ is 1 — in all of the 600 executions, the mean relative difference between the result from the sequential and parallel executions is less than one percent. The generated parallel program therefore satisfies the statistical accuracy guarantee (note that $\hat{p} - 0.05 > g$).

2.2 Wald Sequential Probability Ratio Test

To use the Wald Sequential Probability Ratio Test, we start with an accuracy goal g, a lower accuracy interval ϵ_1, an upper accuracy interval ϵ_2, an approximate false positive bound α, and an approximate false negative bound β. We then repeatedly execute the candidate parallel program and feed the Wald Test the ensuing sequence of results from the accuracy test. After observing some number of accuracy test results (the precise number is determined by the length of the sequence and the number of accuracy tests that succeed), the Wald Test terminates and produces either yes or no. The guarantee is that $P[\text{yes}|p < g - \epsilon_1] < \alpha/\beta$ (i.e., the likelihood that the Wald Test says that the parallel program satisfies the accuracy goal when it is in fact at least ϵ_1 less reliable than the goal is less than α/β) and $P[\text{no}|p > g + \epsilon_2] < (1 - \beta)/(1 - \alpha)$ (i.e., the likelihood that the Wald Test says that parallel program does not satisfy the accuracy goal when it is in fact at least ϵ_2 more reliable than the goal is less than $(1 - \beta)/(1 - \alpha)$).

A false positive occurs when the Wald Test says that the parallel program satisfies the accuracy test when in fact it does not. The guarantee bounds the false positive rate as follows: if the Wald Test says that the program satisfies the accuracy goal, the likelihood that the program is actually less accurate than $g - \epsilon_1$ is bounded above by (i.e., less than) α/β, which is close to α when β is close to one. A similar analysis gives a corresponding bound for the false negative rate – a false negative occurs when the Wald Test says that the program does not satisfy the accuracy test when in fact it does. The likelihood that the program rejected by the test is more accurate than $g + \epsilon_2$ is less than $(1 - \beta)/(1 - \alpha)$, which is approximately $1 - \beta$ when α approaches 0.

Assume that $g = 0.95$, $\epsilon_1 = 0.01$, $\epsilon_2 = 0.01$, $\alpha = 0.01$, $\beta = 0.99$, and the parallel program always produces an acceptably accurate result. Then the Wald Test says yes (i.e., that the program satisfies the accuracy requirement) after 219 runs.

2.3 Environment Sensitivity

We note that the statistical accuracy guarantee is valid only for environments with the same characteristics as the environment executing the runs used to obtain the statistical accuracy guarantee. Significant changes in the execution environment (for example, changing the characteristics of the underlying hardware platform or using larger numbers of threads) require the revalidation of the statistical accuracy guarantee.

3. Analysis and Transformation

QuickStep is built using the LLVM compiler infrastructure [15].

3.1 Profiling

QuickStep obtains its profiling information as follows. When it compiles the sequential program for profiled execution, it inserts instrumentation that counts the number of times each basic block is executed. It also inserts instrumentation that maintains a stack of active nested loops. It also inserts additional instrumentation so that, when the sequential program executes for profiling purposes, it 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. QuickStep uses the resulting generated instruction counts to attempt to prioritize the parallelization of the loops that execute more instructions in the profiling runs over loops that execute fewer instructions. Note that this policy prioritizes outermost loops over nested loops.

3.2 Parallelization Transformation

Given a loop to parallelize, our transformation first extracts the loop body into a separate function. This function takes two parameters: the iteration of the loop to execute, and a pointer to a structure of loop body parameters. This structure contains one field for each local variable from the originally enclosing procedure that the loop body accesses. If the loop body reads but does not write the local variable, the variable is passed by value — the structure contains a copy of the variable. If the loop body writes the local variable, the variable is passed by reference — the structure contains a pointer to the variable in the stack frame of the procedure that originally contained the loop body. In the enclosing procedure where the loop body originally occurred in the sequential program, the transformation allocates and initializes the structure containing the parameters for the loop body. This structure is allocated on the stack of the thread executing the enclosing procedure. The loop body itself is replaced with a call to the QuickStep runtime system. This call specifies the lower bound of the loop, the upper bound of the loop, the separate function containing the extracted loop body, and a pointer to the structure containing the loop body parameters. The call to the runtime system triggers the parallel execution of the loop.

3.3 Memory Profiling

The QuickStep memory profiler instruments the program to log the addresses that reads and writes access. Given the memory profiling information for a given loop, it computes the *interference density* for each store instruction s, i.e., the sum over all occurrences of that store instruction s in the log of the number of store instructions p in parallel iterations that write the same address. Conceptually, the interference density quantifies the likelihood that some store instruction p in a parallel loop will interfere with an update operation 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 updates with the highest interference density. Because the log also contains dynamic call stack information, it is possible to find the enclosing methods on the stack for each execution of the logged load or store instruction.

3.4 Synchronization and Replication Transformations

QuickStep's synchronization and replication transformations operate on objects with multiple implementations of synchronizable and replicatable operations. The synchronized version of operation *op* is named `sync`*op*; the version with replication is named `repl`*op*. When the compiler is asked to perform a synchronization or replication introduction transformation, it is given a call site that invokes the operation to transform (in our current implementation this information comes from the memory profiler, which uses the call stack information to find the call site). The compiler performs the transformation by simply changing the invoked method at the call site, using the naming convention to find the synchronized or replicated method to invoke.

QuickStep can work with any object that provides the methods necessary to apply the synchronization and replication transformations at the call sites that invoke the corresponding methods on the object. Our current system uses versions that we manually developed. It is also straightforward to build a system to automatically transform a sequential implementation to an implementation that exports the required synchronized and replicated versions of the appropriate methods.

3.5 Parallelization Space Search

The parallelization, synchronization, and replication transformations create a search space of parallel programs. The compiler searches this space as follows.

- **Loops:** At any given point in time, the compiler is searching the parallelization space associated with a given loop. It starts the search by parallelizing the loop. The loop parallelization order is given by the amount of time the loop consumes in the sequential information, with the most time-consuming loops parallelized first.
- **Synchronization:** Given a parallelized loop that causes the application to produce an unacceptable result, the compiler applies the synchronization transformation until either it runs out of synchronization transformation opportunities or the application produces an acceptable result on the representative inputs. The synchronization transformations are applied according to the interference density priority order.
- **Replication:** Given a parallelized loop that causes the application to exhibit poor performance, the compiler applies the replication transformation until either it runs out of replication transformation opportunities or the application produces an acceptable result on the representative inputs. The replication transformations are applied according to the interference density priority order.

The search algorithm evaluates the performance and output acceptability with one test run on each representative input. If it is unable to find an acceptable parallelization for a given loop (i.e., a parallelization with good performance that enables the application to produce an acceptable output), it does not parallelize the loop.

When the search algorithm finishes its attempt to parallelize all of the time-consuming loops (the current compiler does not attempt to parallelize loops that take up less than 10% of the execution time), it performs the statistical accuracy guarantee runs.

Application	Sequential	2	4	8
Barnes-Hut (First Input)	1 (27.09s)	1.9 (14.5s)	3.4 (8.06s)	5.7 (4.8s)
Barnes-Hut (Second Input)	1 (61.73s)	1.9 (33.0s)	3.3 (18.39s)	5.7 (10.8s)
Search (First Input)	1 (23.6)	1.9(12.2)	3.7(6.3)	5.7(4.2)
Search (Second Input)	1 (123.9)	1.9(63.8)	3.7(33.7)	5.7(21.7)
String (First Input)	1(7.84s)	1.9(4.1s)	3.5(2.2s)	6.2(1.3s)
String (Second Input)	1(19.2s)	1.9(10.2s)	3.7(5.3s)	6.4(3.0s)
Water (First Input)	1(8.2s)	1.4(5.9s)	2.7(3.0s)	3.8(2.1s)
Water (Second Input)	1(25.0s)	1.4(17.4s)	2.8(11.1s)	4.7(5.3s)

Table 1: Speedups over Sequential Program

4. QuickStep Runtime System

The QuickStep runtime system is implemented as a library linked into the automatically parallelized program. It follows the standard master/worker structure characteristic of programs that execute parallel loops. At the start of the execution the QuickStep runtime system creates a single master thread and a (configurable) number of worker threads. The master thread starts the execution of the parallelized program. The worker threads wait at a barrier until there is a parallel loop to execute.

When the master thread encounters a parallel loop and makes the call to the runtime system that triggers the parallel execution of the loop, the master thread sets up the data structure that the threads use to control the execution of the loop, then enters the barrier. This entry causes all threads to drop through the barrier. Each thread then accesses the data structure that controls the parallel execution, acquires loop iterations to execute, and executes each of its acquired loop iterations by invoking the function containing the extracted loop body (see Section 3.2). Our currently runtime system implementation provides three different loop scheduling policies. Each policy is selectable under the control of the user running the program. The first policy dynamically load balances the computation — each worker thread repeatedly accesses the central loop scheduling data structures to acquire a single loop iteration. Only when the thread finishes that iteration does it return to acquire the next iteration. This scheduling policy is appropriate for loops whose iterations have (on average) large but potentially dramatically different execution times.

The second scheduling policy assigns blocks of iterations to threads at the start of the parallel loop. Each thread has a thread id i. If there are n loop iterations and t threads, thread i executes iterations $i * n/t$ through $((i+1) * n/t) - 1$. This scheduling policy is appropriate for loops whose iterations all take roughly equivalent time and exhibit better locality if adjacent loop iterations are executed on the same thread. The third scheduling policy cyclically assigns iterations to threads at the start of the parallel loop. Specifically, thread i is assigned iterations i, $t{+}i$, $2t{+}i$, ... This scheduling policy has less overhead than the first policy (which dynamically assigns loop iterations as the loop executes) and is appropriate for loops whose iterations may take different amounts of time, but the variation in execution times is such that the cyclic mapping evenly balances the load across the different threads.

5. Acceptability Criteria

QuickStep's statistical approach to the accuracy of a parallel computation is designed to work with any accuracy test that takes as input the result of a sequential execution and a result from a parallel execution and produces as output either success (indicating that the parallel execution produced an acceptably accurate result) or failure (indicating that the parallel execution produced a result that was not accurate enough). Given such a test, QuickStep (by sampling sufficiently many parallel executions as described in Section 2) is able to provide a statistical guarantee that characterizes the likelihood that a given parallel execution will produce an acceptably accurate result.

5.1 Distortion Metric

It is often appropriate to use an accuracy test based on the relative scaled difference between selected outputs from the sequential and parallel executions. Specifically, we assume the program produces an output of the form $o_1, \ldots, o_m$, where each output component o_i is a number. Often the program will produce more than m outputs, with the developer selecting m of the outputs as the basis for the acceptability test.

Given an output $o_1, \ldots, o_m$ from a sequential execution and an output $\hat{o}_1, \ldots, \hat{o}_m$ from a parallel execution, the following quantity d, which we call the *distortion*, measures the accuracy of the output from 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.

5.2 Program Failures

In some circumstances the parallelization may cause the program to fail to produce any output whatsoever (typically because the program crashes before it can produce the output). QuickStep is currently configured to reject such parallelizations. However, it would be clearly possible to extend the approach to allow such parallelizations. The straightforward approach would be to augment the current statistical accuracy criteria with additional components specifying acceptable likelihoods of failing. We note that in our experience to date, each candidate parallelization either never fails or almost always fails, which minimizes the motivation to extend the statistical accuracy criteria to include program failure likelihoods.

6. Experimental Results

We use QuickStep to parallelize five 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.
- **Search:** Search [7] is a program from the Stanford Electrical Engineering department. 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.

- **String:** String [13] uses seismic travel-time inversion to construct a two-dimensional discrete velocity model of the geological medium between two oil wells. 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 non-destructive 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.
- **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 [4] and performs the same 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 [29].

6.1 Methodology

We obtained the applications in our benchmark suite and two representative inputs for each application, with the second input requiring substantially more computation than the first input. We then used QuickStep to automatically parallelize the appplications, using the representative inputs to perform the executions required to check if the candidate parallelizations satisfied the statistical accuracy guarantees. All of the steps are automated except the synchronization and replication introduction transformations, which we perform manually under the guidance of the memory profiler and the compiler. For each application we used the scheduling policy (see Section 4) that provided the best performance for that application.

We ran the resulting parallelized applications on an Intel Xeon E7340 with four quad-core processors. We ran a sequence of trials with two, four, and eight threads. We performed the statistical accuracy trials with eight threads and 600 trials, as required by Hoeffding bound with $\epsilon = 0.05$ and $\delta = 0.95$. To better analyze the scalability and performance in a range of conditions, we also ran 200 trials with two and four threads. For each trial run we recorded the execution time and the distortion, enabling the *a-posteriori* application of accuracy tests with different distortion bounds b.

6.2 Speedups

Table 1 presents the speedups the automatically parallelized programs achieve over the sequential versions on the representative inputs. Each entry is of the form $x(y)$. Here y is the mean execution time over all trial runs (200 runs for two and four threads, 600 runs for eight threads); x is the corresponding mean speedup (the mean execution time of the parallel version divided by the execution time of the sequential version). Barnes-Hut, Search, and String all obtain speedups in excess of five at eight threads.

6.3 Time Breakdowns

Figures 3 through 10 present the breakdowns for the total amount of work that the threads executing each application perform. The sequential work is the time the master thread spends executing the sequential part of the application. The sequential idle time is the sum over the worker threads of the time spent idle while the master thread executes the sequential part of the computation. The parallel work time is the sum over all threads of the amount of time spent executing iterations of parallel loops. The parallel idle time is the sum over all threads of the amount of time spent waiting at barriers at the end of parallel loops for other threads to finish their iterations in parallel loops. The numbers in the charts are the mean values of these quantities over all of the executions. Because these charts include the total time over all threads required to complete the computation, the computation scales perfectly if the heights of the bars stay the same as the number of threads increases. Any increase in the bar heights indicate sources of ineffiency that impair (to a proportional degree) the performance of the parallel computation in comparison with the performance of the sequential computation.

These charts show that for all applications except Barnes-Hut, almost all of the computation takes place in loops that QuickStep parallelizes. For Barnes-Hut, 93 percent of the executed instructions take place in the loop that QuickStep parallelizes. The charts also show that for Barnes-Hut there is almost no parallel idle time, indicating that the application is almost perfectly load balanced, and very little increase in the parallel compute time, indicating that hardware resource effects (such as communication and caching overhead) have little effect on the performance. The limiting factor on the parallel performance of Barnes-Hut on this machine is therefore the remaining 7 percent of the instructions that execute serially on one thread while the other threads are idle (as reflected in the increase in the serial idle time as the number of threads increases). For the first input the primary limiting factor on the performance of String is the sequential idle time. This sequential idle time also affects the performance of String on the second input, but an increase in parallel idle time shows that application is not always load balanced.

Search has almost no sequential work time, indicating that almost all of the computation takes place in loops that QuickStep parallelizes. The charts indicate that the primary limiting factor on the parallel peformance is an unbalanced load. As the number of threads increases, all of the time components stay constant except the parallel idle time. The limiting factor on the parallel performance of this application is therefore the load imbalance in the parallelized loop. For Water the limiting factor is the increase in parallel work caused by hardware resource effects.

6.4 Distortions

Table 2 presents the mean and variance for the observed distortions for the applications running with eight threads. Each entry is of the form $x(y)$, where x is the mean distortion and y is the variance. For Barnes-Hut and Search, the iterations of the parallelized loops are independent (no iteration reads or writes a value that another iteration writes), the generated parallel program always generates the same result as the sequential program, and the distortion is always zero. For String and Water, the iterations of the parallelized loops have dependences (each iteration accesses values written by other iterations), the output varies depending on the order in which the loop iterations execute, and the generated parallel program contains data races. Both generated parallel programs nondeterministically produce different outputs on different executions. Both the mean and variance of their distortions is nonzero.

On all executions the distortion for String is always less than 0.1, which is acceptably accurate for this application. Moreover, the vast majority (over 95 percent) of the distortions are between 0.03 and 0.06, and always less than 0.07. The mean distortion for Water is approximately 0.009. Moreover, on all runs the distortion for Water is between 0.006 and 0.013. For both String and Water all of the observed distortions appear within a narrow band, indicating that the perturbations introduced by the data races are relatively consistent across executions.

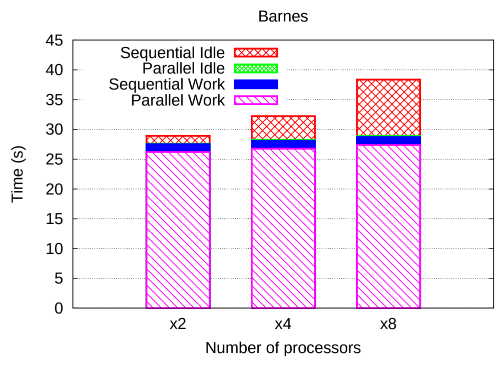

Figure 3: Barnes (First Input)

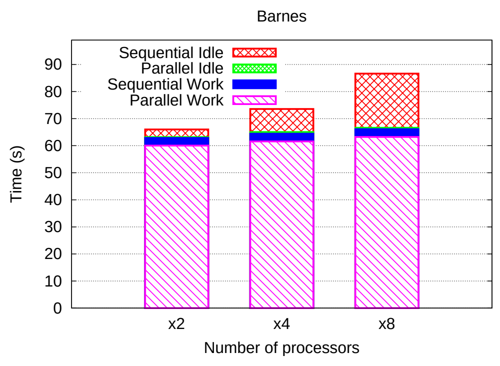

Figure 4: Barnes (Second Input)

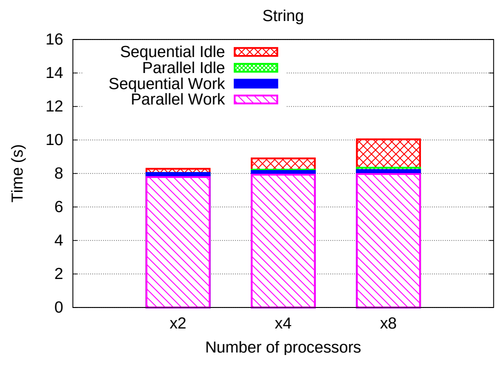

Figure 5: String (First Input)

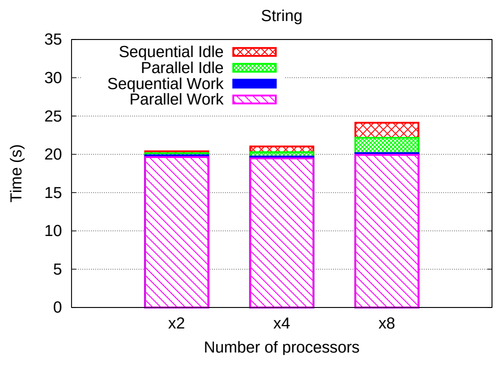

Figure 6: String (Second Input)

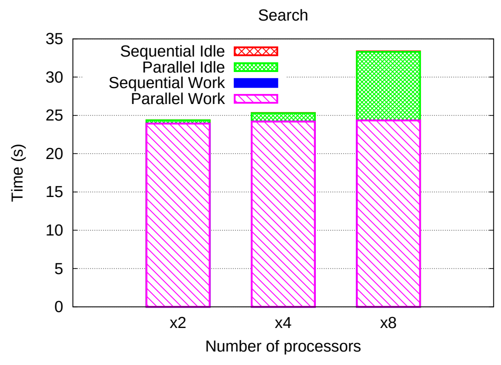

Figure 7: Search (First Input)

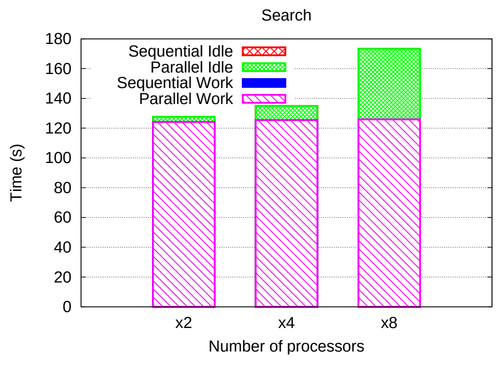

Figure 8: Search (Second Input)

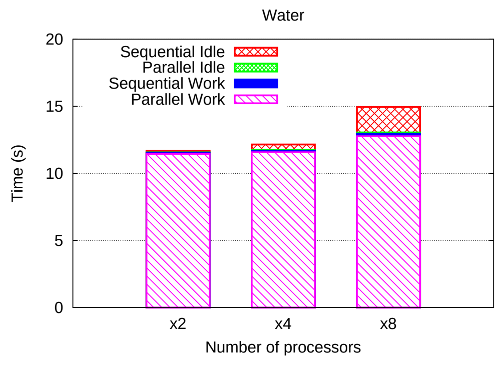

Figure 9: Water (First Input)

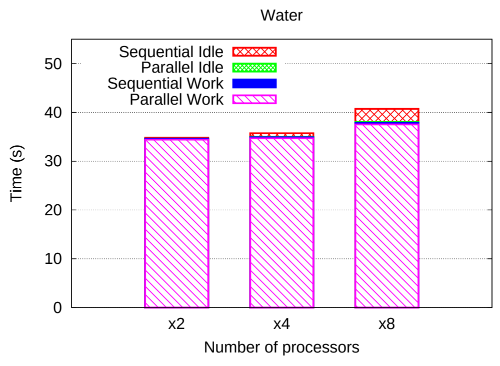

Figure 10: Water (Second Input)

Application	Distortion (First Input)	Distortion (Second Input)
Barnes-Hut	0(0)	0(0)
Search	0(0)	0(0)
String	0.048 (0.0038)	0.049 (0.0037)
Water	0.009 (0.001)	0.011 (0.0008)

Table 2: Distortion Means and Variances for Executions with Eight Threads

6.5 Barnes-Hut

Almost all of the computation in the Barnes-Hut N-body solver takes in the force computation, which iterates over all the bodies, using the space subdivision tree to compute the force acting on that body. There are no cross-iteration dependences in force computation loop. QuickStep's parallelization of this loop therefore produces a parallel program that deterministically produces the same result as the original sequential program. The resulting parallel computation exhibits good performance on the target hardware platform. The first input for Barnes-Hut simulates $2^{18} = 262,144$ bodies; the second input simulates $2^{19} = 524,288$ bodies. The bodies are placed at psuedo-random locations at the start of the simulation. Our manual evaluation of the source code of the force computation loop confirms the lack of dependences between parallel iterations and the overall acceptability of the parallelization.

6.6 Search

Almost all of the computation in Search takes place inside two loops, one of which is nested inside the other. The outermost loop iterates over a set of points. The innermost loop traces a fixed number of particles for each point to compute the number of traced particles that exit the front of the simulated material. QuickStep parallelizes the outermost loop only. An inspection of the code indicates that this loop has no cross-iteration dependences and that the resulting parallel computation always produces an identical output as the sequential computation. Note that the starting sequential program allocates separate psuedo-random number generator for each point.

6.7 String

Almost all of the computation in String takes place inside two loops, one of which is nested inside the other. The loops trace rays through a discretized model of the geology between two oil wells. The outermost loop iterates over a set of sources. The innermost loop traces a collection of rays for each source. It computes, for each ray, the time the ray takes to travel from one oil well to the other as it propagates through the geometry model. For each ray, String compares the traced travel time to an experimentally measured travel time and backprojects the difference along the path of the ray to update the model of the geology. QuickStep parallelizes the outermost loop only. The resulting automatically generated parallel computation may contain data races — it is possible for two threads to update the same location in the model of the geology at the same time. And in fact, the noise in the output from the parallel version indicates that at least some races occur during the parallel executions. The results also indicate, however, that the noise is small and that the parallel executions produce an output that satisfies the statistical accuracy goal. Our manual evaluation of the source code confirmed the potential presence of (infrequent) data races and the overall acceptability of the parallelization.

The two inputs for String vary in the number of rays they shoot, the angles at which it shoots the rays, the angle between successive rays, and the number of iterations over the geology model that the computation performs.

The accuracy test measures the mean distortion over all of the elements of the discretized geology model that the program produces as output. We set the acceptable distortion bound for this application to be 0.1.

6.8 Water

Almost all of the computation in Water takes place in two loop nests. One of the loop nests computes the forces acting on each water molecule. The other loop nest computes the potential energy of the system of water molecules. Both loop nests have two loops. The outermost iterates over all molecules in the system; the innermost iterates over the remaining molecules to compute all pairs of interactions. Both loop nests accumulate contributions into a single object. With the standard sequential implementation of this scalar accumulator, parallel executions of these loops produce unacceptably inaccurate outputs — pervasive race conditions associated with the accumulator object cause the computation to lose many of the updates to this object. Linking the program against an implementation of the accumulator object designed to execute in a parallel environment eliminates the race conditions for the accumulator object. QuickStep then parallelizes both outer loops in both loop nests.

With this parallelization, there are still some remaining data races. These races exist when two threads update the force acting on a given molecule at the same time. These data races occur infrequently enough so that the resulting noise in the output leaves the output acceptably accurate. Our evaluation of the source code confirms the presence of (infrequent) data races and the acceptability of the parallelization.

The acceptability test for Water is based on the mean distortion of several values that the simulation produces as output. Water produces a set of outputs related to 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 first input for Water simulates 1000 water molecules; the second input simulates 1728 water molecules. At the start of the simulation the water molecules are placed in a regular lattice with psuedo-randomly assigned momenta.

6.9 Panel Cholesky

Panel Cholesky 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. ClearView finds three candidate parallel loops in this computation. Together, these three loops account for over 59 percent of the instructions in the computation.

The accuracy test for this application uses the factored matrix to solve for a vector with a known correct result. If the norm of the solution differs from the known correct result by more than a given tolerance, the factored matrix fails the acceptability test.

Parallelizing the first or second loop produces a generated parallel program that always fails the accuracy test. The remaining third loop accounts for approximately 18 percent of the instructions in the computation. Parallelizing this loop produces a parallel program that always passes the accuracy test, but does not execute significantly faster than the original sequential program.

Several characteristics of this application place it well beyond the reach of QuickStep (or, for that matter, any existing or envisioned parallelizing compiler [29]). 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 perform an inspector/executor approach that extracts the nonzero pattern from the matrix, analyzes the nonzero pattern to build a schedule of the legal parallel execution, then executes this schedule to execute the computation in parallel.

This application illustrates an important limitation of the QuickStep approach (and the approach behind virtually all other parallelizing compilers). QuickStep is designed to exploit concurrency available in loops whose iterations can execute 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.

7. Related Work

We discuss related work in parallelizing compilers, interactive profile-driven parallelization, and unsound transformations.

7.1 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 [12, 5, 18]. 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 a statistical 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. It can also effectively parallelize a wider range of programs.

Commutativity analysis [27, 1] 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, statistically analyze the output of the parallelized program rather than program itself with the goal of producing a parallel program with outputs that are statistically close to, rather than necessarily identical to, the results that the sequential program produces. Once again, our approach produces a dramatically simpler compiler and analysis 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 [33, 20, 6, 22]. 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 runtime instrumentation that extracts additional information (available only at run time) that may enable the parallel execution of the program [31, 22, 21, 11]. 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 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).

7.2 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 [34, 30, 10]. These potential parallelizations are then typically presented to the developer for approval.

It is possible to use QuickStep in a similar way, to explore potential parallelizations that are then certified as acceptable by the programmer. And because QuickStep can generate successful parallelizations that violate the data dependences, it can generate a broader range of parallelizations for programmer approval. Note that this broader range may make a significant difference in the performance of the produced parallel computation. The successful parallelizations of two of the applications in our benchmark set (String and Water) inherently violate the underlying data dependences, which places these parallelizations beyond the reach of any technique that attempts to preserve these dependences.

7.3 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 [24]. 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 [25]. 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 Hoeffding bound 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 trial runs to obtain tight statistical distortion bounds.

7.4 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 [26, 3], code injection attacks [26, 19], data structure corruption errors [8, 9], memory leaks [17], infinite loops [17], improve performance [14], respond to the clock frequency changes or core failures [14], and discover additional optimization opportunities [16]. 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.

8. Conclusion

Automatic parallelization of sequential programs is a research area with a long history. The field has demonstrated successes within specific computation domains, but many computations remain well beyond the reach of traditional approaches (which analyze the sequential program to provide a guarantee that the parallel and sequential programs will always produce identical outputs). The difficulty of building compilers that use these approaches and the large classes 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. QuickStep's statistical accuracy approach provides both the simplicity and wider applicability that the field requires. The results indicate that this approach enables QuickStep to automatically generate parallel computations that perform well and produce outputs that have tight statistical accuracy guarantees.

References

[1] ALEEN, F., AND CLARK, N. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, Mar. 2009).

[2] BARNES, J., AND HUT, P. A hierarchical O(NlogN) force calculation algorithm. *Nature 324*, 4 (Dec. 1986), 446–449.

[3] BERGER, E., AND ZORN, B. DieHard: probabilistic memory safety for unsafe languages. In *PLDI* (June 2006).

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

[5] BLUME, W., EIGENMANN, R., FAIGIN, K., GROUT, J., HOEFLINGER, J., PADUA, D., PETERSEN, P., POTTENGER, W., RAUGHWERGER, L., TU, P., AND WEATHERFORD, S. Effective automatic parallelization with Polaris. In *International Journal of Parallel Programming* (May 1995).

[6] BRIDGES, M., VACHHARAJANI, N., ZHANG, Y., JABLIN, T., AND AUGUST, D. Revisiting the sequential programming model for multicore. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Chicago, IL, Dec. 2007).

[7] BROWNING, R., LI, T., CHUI, B., YE, J., PEASE, R., CZYZEWSKI, Z., AND JOY, D. Low-energy electron/atom elastic scattering cross sections for 0.1-30keV. *Scanning 17*, 4 (July/August 1995), 250–253.

[8] DEMSKY, B., ERNST, M., GUO, P., MCCAMANT, S., PERKINS, J., AND RINARD, M. Inference and enforcement of data structure consistency specifications. In *ISSTA '06*.

[9] DEMSKY, B., AND RINARD, M. Data structure repair using goal-directed reasoning. In *ICSE '05* (2005).

[10] DING, C., SHEN, X., KELSEY, K., TICE, C., HUANG, R., AND ZHANG, C. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (2007), ACM New York, NY, USA.

[11] DING, Y., AND LI, Z. An Adaptive Scheme for Dynamic Parallelization. *Lecture notes in computer science* (2003), 274–289.

[12] HALL, M., ANDERSON, J., AMARASINGHE, S., MURPHY, B., LIAO, S., BUGNION, E., AND LAM, M. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Comput.* (Dec. 1996).

[13] HARRIS, J., LAZARATOS, S., AND MICHELENA, R. Tomographic string inversion. In *Proceedings of the 60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts* (1990), pp. 82–85.

[14] HOFFMANN, H., MISAILOVIC, S., SIDIROGLOU, S., AGARWAL, A., AND RINARD, M. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Tech. Rep. MIT-CSAIL-TR-2009-042, MIT, Sept. 2009.

[15] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[16] MISAILOVIC, S., SIDIROGLOU, S., HOFFMANN, H., AND RINARD, M. Quality of service profiling. In *ICSE '10* (2010).

[17] NGUYEN, H., AND RINARD, M. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM '07*.

[18] OPEN64. Open64: Open research compiler. `www.open64.net`.

[19] PERKINS, J., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., ET AL. Automatically patching errors in deployed software. In *SOSP '09*.

[20] PRABHU, M., AND OLUKOTUN, K. Exposing speculative thread parallelism in spec2000. In *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[21] RAUCHWERGER, L., AMATO, N., AND PADUA, D. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing* (1995), ACM.

[22] RAUCHWERGER, L., AND PADUA, D. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation* (June 1995).

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

[24] RINARD, M. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 2006 ACM International Conference on Supercomputing* (Cairns, Australia, June 2006).

[25] RINARD, M. Using early phase termination to eliminate load imbalancess at barrier synchronization points. In *Proceedings of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Canada, Oct. 2007).

[26] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND WILLIAM S. BEEBEE, J. Enhancing Server Availability and

Security Through Failure-Oblivious Computing. In *OSDI* (December 2004).

[27] RINARD, M., AND DINIZ, P. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 19*, 6 (Nov. 1997).

[28] RINARD, M., AND LAM, M. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems 20*, 3 (May 1998).

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

[30] RUL, S., VANDIERENDONCK, H., AND DE BOSSCHERE, K. A dynamic analysis tool for finding coarse-grain parallelism. In *HiPEAC Industrial Workshop* (2008).

[31] RUS, S., PENNINGS, M., AND RAUCHWERGER, L. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the 21st annual international conference on Supercomputing* (2007), ACM New York, NY, USA, pp. 263–273.

[32] SINGH, J., AND HENNESSY, J. Finding and exploiting parallelism in an ocean simulation program: Experience, results and implications. *Journal of Parallel and Distributed Computing 15*, 1 (May 1992).

[33] TINKER, P., AND KATZ, M. Parallel execution of sequential Scheme with Paratran. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming* (Snowbird, UT, July 1988), pp. 28–39.

[34] TOURNAVITIS, G., WANG, Z., FRANKE, B., AND O'BOYLE, M. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 PLDI*, ACM New York, NY, USA.