Using Early Phase Termination To Eliminate Load Imbalances At Barrier Synchronization Points

Martin Rinard
Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 021139
rinard@csail.mit.edu

Abstract

We present a new technique, *early phase termination*, for eliminating idle processors in parallel computations that use barrier synchronization. This technique simply terminates each parallel phase as soon as there are too few remaining tasks to keep all of the processors busy.

Although this technique completely eliminates the idling that would otherwise occur at barrier synchronization points, it may also change the computation and therefore the result that the computation produces. We address this issue by providing *probabilistic distortion models* that characterize how the use of early phase termination distorts the result that the computation produces. Our experimental results show that for our set of benchmark applications, 1) early phase termination can improve the performance of the parallel computation, 2) the distortion is small (or can be made to be small with the use of an appropriate compensation technique) and 3) the distortion models provide accurate and tight distortion bounds. These bounds can enable users to evaluate the effect of early phase termination and confidently accept results from parallel computations that use this technique if they find the distortion bounds to be acceptable.

Finally, we identify a general computational pattern that works well with early phase termination and explain why computations that exhibit this pattern can tolerate the early termination of parallel tasks without producing unacceptable results.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Concurrent Programming;
D.2.5 [*Software Engineering*]: Testing and Debugging;
D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Performance, Reliability, Design, Languages

Keywords Barrier Synchronization, Parallel Computing, Probabilistic Distortion Models, Early Phase Termination

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

1. Introduction

Many parallel programs exhibit a parallelism pattern consisting of alternating parallel and serial phases. Each parallel phase consists of a set of tasks that can execute in parallel. When all of the tasks finish, a serial phase consisting of a single task continues the computation until the start of the next parallel phase. Each serial phase typically accesses data from all of the tasks in the preceding parallel phase. These tasks must therefore finish before the subsequent serial phase can begin. The synchronization that accomplishes this temporal separation is called *barrier synchronization* because it imposes a temporal barrier that separates the two phases.

A well-known issue with barrier synchronization is *barrier idling*, which occurs at the end of the parallel phase when there are few tasks left to execute. If there is a mismatch in processor speeds or task sizes, processors may be left idle as they wait for the remaining tasks to finish. This idling can limit the performance of the parallel computation.

We propose a simple solution, *early phase termination*, to barrier idling: when there are too few tasks to keep all of the processors busy at the end of the parallel phase, simply terminate the remaining tasks and proceed immediately to the subsequent serial phase. This solution completely eliminates any barrier idling. It may, however, change the computation and therefore the result that the computation produces. There are two potential questions: 1) how much does the result change, and 2) how predictable is the change? If the change is sufficiently small and predictable, users may be willing to accept the perturbed result in exchange for the increased performance associated with the elimination of barrier synchronization.

1.1 Distortion Models

To enable the user to evaluate the potential impact of early phase termination on the results of their computations, we build a statistical *distortion model* that characterizes the effect of terminating tasks [20]. We obtain and use this model as follows:

- **Task Decomposition:** The developer specifies the task decomposition by identifying *task blocks* in the program. Each task block consists of a block of code whose execution corresponds to a task in the computation. Note that a given task block may execute many times during the course of a computation and may therefore generate many tasks into the computation.
- **Phase Identification:** The developer identifies the parallel and serial phases in the computation.
- **Baseline:** We obtain several sample inputs for which the program is known to generate correct output. We run the program on these inputs, executing every task to completion, and record the outputs that it generates.
- **Criticality Testing:** We configure the execution platform to randomly skip executions of selected tasks at target skip rates. We then select each task block in the program in turn, skip executions of that task block at a targeted rate, and observe the resulting output distortion. If the task skips produce unacceptable distortion or cause the computation to fail, we mark the task block as *critical*, otherwise we mark the task block as *skippable*.
- **Distortion Model:** Given a set of skippable task blocks, we run repeated trials in which we randomly select a target task skip rate for each skippable task block, execute the computation, then record both the observed task skip rates and the resulting output distortion. We then use regression [8] to obtain a probabilistic model that estimates the distortion as a function of the task skip rates.
- **Production Runs:** We use early phase termination in the production runs to eliminate barrier idling. We use the resulting observed proportion of early terminated tasks and the distortion model to obtain an estimated distortion and confidence interval around the distortion. The estimated distortion and confidence interval allow the user to determine whether or not to accept the result.

We have applied this approach to several benchmark applications selected from the Jade benchmark suite [17]. Our results show that the models are quite accurate (they have good statistical properties and usually explain almost all of the observed variation in the distortion data) and that the estimated distortion and confidence intervals are small enough to provide useful distortion bounds in practice. And it is possible to use models developed on one set of inputs to accurately predict distortion properties for other inputs. Finally, our results indicate that the elimination of barrier idling can reduce the overall execution time and increase the performance of our benchmark applications.

1.2 Computation Pattern

Based on our experience with our benchmark applications, we have identified a general computation pattern that interacts well with our approach. Specifically, the parallel phases in our applications generate many contributions to a final result, then combine the contributions to obtain the result. If each parallel task either generates or combines contributions, the net effect of terminating tasks early is simply to discard some of the contributions. Our results indicate that the distortion associated with discarding these contributions is often quite small (or that it is possible to accurately compensate for the discarded contributions).

Identifying this pattern provides a conceptual framework that can help users better understand the effect of early phase termination on their computations. One of our benchmark applications, for example, performs a Monte Carlo simulation in which each set of trials comprises a contribution. The net effect of early phase termination is simply to drop some of the trials. Another application traces a set of rays through a medium to compute the density of different parts of the medium. The net effect of early phase termination is simply to drop some of the traced rays.

We anticipate that this kind of understanding, which translates the effect of early phase termination back into concepts from the underlying application domain, can help users determine if they are comfortable using the technique. In the two examples above, it is clear that, in practice, both computations have enough redundancy (either from other trials or other traced rays) to easily tolerate the loss of some of the contributions without substantially impairing the utility of the final result. Indeed, users may find that understanding the effects of early phase termination at the level of the application domain may provide a more compelling case for its use than the (in comparison relatively opaque) probabilistic distortion models.

1.3 A Broader Perspective

To place early phase termination in a broader perspective, consider that almost all scientific computations are inherently inaccurate in that they are designed to produce an approximation to an ideal result rather than the ideal result itself. Many computational chemistry programs, for example, use classical or semi-empirical methods instead of *ab initio* quantum mechanics (which is computationally intractable for larger molecules) [10]. Many scientific computations discretize conceptually continuous fields to enable the representation and computation of approximate solutions on digital computers, with the granularity of the discretization determined, in part, by factors such as the amount of available memory and computational power [15]. As these examples illustrate, accuracy versus performance tradeoffs strongly influence the form and basic approach of almost all scientific computations.

The key question for such computations, then, is not correctness or incorrectness, but accuracy and pragmatic feasibility — does the computation produce a result that is close enough to the ideal result within an acceptable time frame, and, if so, can the user determine that the result is acceptably close the ideal result? Viewed from this perspective, early phase termination is simply yet another technique for obtaining acceptable performance at the potential cost of some accuracy. And the probabilistic distortion models enable the user to evaluate the effect of the technique and determine whether or not the final result is acceptably accurate.

It is worth considering how these issues might play out in different application domains. Many information retrieval computations, for example, map a subcomputation over a set of discrete items, then combine the results, with the overall computation able to tolerate the loss of some of the subcomputations [7]. Because of the ability of the human sensory system to tolerate noise and other processing artifacts, computer graphics and other sensory processing computations can often drop subcomputations without unacceptably degrading the final output [14, 19]. As these examples illustrate, early phase termination may be feasible in practice for a wide range of computations, including computations for which probabilistic distortion models may not be readily available.

1.4 Contributions

This paper makes the following contributions:

- **Elimination of Barrier Idling:** It introduces the concept of eliminating barrier idling by terminating tasks at the end of parallel phases.
- **Distortion Models:** It introduces the use of probabilistic distortion models for characterizing the effect of early phase termination on the result that the program generates. These models provide an estimate of the distortion and probabilistic accuracy bounds around this distortion estimate.
- **Explanation:** It identifies a computation pattern that works well with early phase termination and explains why computations that exhibit this pattern can terminate tasks before they complete and still produce acceptable results.
- **Experimental Results:** It provides experimental results for our set of benchmark applications. These results indicate that the distortion and accuracy bounds are small enough for practical use and that eliminating barrier idling can improve the overall parallel performance.

The remainder of the paper is structured as follows. In Section 2 we provide an example that illustrates our programming model. Section 3 presents the methodology we use to obtain the distortion models. Section 4 presents our experience applying our technique to a set of scientific computations. We present related work in Section 5 and conclude in Section 6.

2. Programming Model

Figure 1 presents a simple procedure, the `add` procedure, that we use to illustrate the programming model. This procedure is written in the Jade programming language [17]; it computes and returns the sum of `n` numbers. The Jade `phase` construct on line `4` indicates that the loop from lines `5` through `9` comprise a parallel phase. All of the tasks in this phase must complete before the program continues past line `10`.

The task block on lines `7` and `8` uses the Jade `withonly` construct to specify that the computation of each number is a task. Each task computes a number `f(i)` and adds the number into the corresponding partial sum `fs[i]`. The `withonly` access specification on line `7` uses the access specification operations `rd(p)` and `wr(p)` to specify that the code in the body of the task may, when it executes, read and/or write the shared object which `p` references.

The code starting at line `11` following the parallel phase is the subsequent sequential phase. The loop on lines `12` through `16` adds up the partial sums into the final sum `sum`. The `add` procedure returns `sum` at line `17`.

```
 1: int add(int n, int shared* shared* fs) {
 2:   int i, sum;
 3:   int shared *p;
 4:   phase {
 5:     for (i = 0; i < n; i++) {
 6:       p = fs[i];
 7:       withonly { rd(p); wr(p); }
 8:       do (p,i) { *p += f(i); }
 9:     }
10:   }
11:   sum = 0;
12:   for (i = 0; i < n; i++) {
13:     p = fs[i];
14:     with { rd(p); } cont;
15:     sum += *p;
16:   }
17:   return sum;
18: }
```

Figure 1. Example Jade Program

When it executes this program, the Jade implementation creates a new task for every execution of the task block on lines 7 and 8. It dynamically analyzes the data dependences between tasks to exploit any available concurrency; Jade implementations exist for a variety of parallel computing platforms [17].

To support the development of the distortion models, we modified the Jade implementation to accept a target task skip rate for each task block. When the program runs, the Jade implementation randomly skips the corresponding tasks at the specified rates.

To construct the distortion model, our technique first runs the program on several inputs for which it is known to produce correct output. It records the output (in this case the value of `sum` that `add` returns), then runs a sequence of trials that randomly skip executions of each task block at a randomly selected skip rate. For each trial it records the output of the computation and the actual task skip rate for each task block.

To quantify the impact of the task skips on the output, the technique computes the *distortion* associated with each trial. In our example the distortion is simply $|(o - \hat{o})/o|$, where o is the correct output and $\hat{o}$ is the observed output from the trial with task skips. Dividing the difference $o - \hat{o}$ by o makes it possible to meaningfully compare distortions from executions with different correct outputs.

The result of the sampling phase is a set of observations x, d, where x is the actual task skip rate for the task block in the program and d is the observed distortion. Our technique takes this set of observations and uses regression [8] to obtain a linear model $\hat{d}(x) = c_0[\pm e_0] + c_1[\pm e_1]x$ of the distortion. Here c_0 and c_1 are the regression coefficients and e_0 and e_1 provide the confidence bounds for these coefficients. For the program in our example, the regression produces the following distortion model.

$$\hat{d}(x) = 0[\pm 0.0002] + 1.0[\pm 0.0007]x$$

In this model $c_0 = 0$, which correctly estimates that there should be no distortion if there are no skipped tasks. The coefficient $c_1 = 1$, which indicates that every increase in the task skip rate produces the same increase in the distortion. So, for example, an increase of 10% in the task skip rate would produce an increase in the distortion of 0.10.[1]

In addition to the model, the regression algorithm provides a variety of statistics that evaluate how well the model fits the data. In our example R^2 is 1, which means that the model perfectly explains the variation in the data. Finally, given a task skip rate x, the regression can provide a confidence bound e around the estimated distortion $c_0 + c_1 x$. In our example the maximum 95% confidence bound over all of the 2064 sampled task skip rate points is 0.0025, which provides a tight confidence interval around the estimated distortion.

We anticipate the following usage scenario. The user has obtained the model and now runs the program in parallel with early phase termination enabled. The program comes back with an output $\hat{o}$. It also informs the user that it terminated several tasks early at the end of parallel phases. The user plugs the early task terminate rate into the model to get an estimated distortion $c_0 + c_1 x$ and confidence bound e. The user then evaluates the distortion and confidence bound to determine if, with high enough likelihood, the distortion is acceptable.

[1] It turns out that for this example it is possible to apply the bias compensation technique discussed in Section 3.8 to obtain an estimator with an expected distortion of 0.0 for task failure rates within the sampled range of 0.0 to 0.75. This technique enables the program to acceptably tolerate much higher task skip rates.

3. Distortion Models

We obtain and evaluate the distortion model for each program as follows.

3.1 Standard Executions

Our approach applies to programs that produce an output of the form $o_1, \ldots, o_m$, where each output component o_i is a number. We obtain several representative test inputs for which the program is known to execute without failures, run the program on these inputs, and record the correct output $o_1, \ldots, o_m$ for each input.

3.2 Distortion Definition

Given a correct output $o_1, \ldots, o_m$ and an observed output $\hat{o}_1, \ldots, \hat{o}_m$, the following quantity d, which we call the *distortion*, measures the accuracy of the observed output:

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

The closer the distortion is to zero, the less distorted is the observed output.

Note that because each difference $o_i - \hat{o}_i$ is scaled by the corresponding correct output component o_i 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 inputs cause the program to produce outputs with different numbers of components m and different correct component values o_i.

Note that the distortion equation weights each output value o_i equally. It is possible to use a set of weights w_i to generalize the distortion equation for programs whose outputs are not all equally important. Each weight w_i would capture the importance of the corresponding output o_i:

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

where the w_i satisfy $m = \sum_{i=1}^{m} w_i$.

3.3 Criticality Testing

It turns out that some programs have task blocks that must always execute completely for the program to produce acceptably accurate output. We experimentally detect these *critical* task blocks as follows. We first configure the underlying execution engine (in our case the Jade runtime system) to randomly skip executions of a selected task block at a specified rate (our criticality testing executions skip 10% of the executions of the selected task block). We then select each task block in turn and run the program at least ten times with the execution engine randomly skipping that task block at the specified rate. If any of these runs does not produce any output at all (typically because the program failed) or if the mean distortion from all of the runs is larger than the specified acceptable distortion for criticality testing (we use an acceptable distortion of 0.10), we identify the task block as a critical task block that must execute completely. Otherwise, we consider the task block to be a *skippable* task block. The purpose of the remaining steps is to characterize how skipping tasks from skippable task blocks affects the distortion of the result that the program produces.

3.4 Distortion Sampling Runs

We next run a set of trials in which we randomly select a target skip rate for each skippable task block, run the program on each of the representative inputs with the execution engine randomly skipping executions of each task block at its target skip rate, then record the distortion and actual skip rate for each skippable task block for that run. If we have n skippable task blocks, the result is a set of observations $x_1^i, \ldots, x_n^i, d^i$, where d^i is the distortion and $x_1^i, \ldots, x_n^i$ are the actual skip rates for the n skippable task blocks in the i'th trial.

Our motivation for selecting target skip rates randomly (instead of using some more systematic approach) was simply to distribute the trials more or less evenly across the skip rate space with no biases. That said, we anticipate that almost any approach that distributes the trials reasonably well across the skip rate space should produce a reasonably accurate model.

3.5 Distortion Model

Given the results from the distortion sampling runs, we use multiple linear regression [8] to compute a linear least-squares distortion model of the form

$$\hat{d}(x_1, \ldots, x_n) = c_0[\pm e_0] + \sum_{i=1}^{n} c_i[\pm e_i]x_i$$

where the c_i are the least-squares coefficients for the regression and the e_i provide the confidence intervals for these coefficients (we use 95% confidence intervals in this paper). Given skip rates x_i for all of the skippable task blocks, this model produces a distortion estimate $\hat{d}(x_1, \ldots, x_n)$ of the expected accuracy of the result produced by a computation with task block skip rates $x_1, \ldots, x_n$.

The regression also produces an F value that assesses how well the model predicts the data and an R^2 value that indicates how much of the variation in the data the model accounts for. Moreover, given a specific point $x_1, \ldots, x_n$ in the skip rate space, the regression can produce a confidence interval around the distortion estimate $\hat{d}(x_1, \ldots, x_n)$ at that point. It is possible to obtain confidence intervals for whatever alpha level one desires; in this paper we use an alpha level of 0.05, which produces a 95% confidence interval. We use the SAS system to compute the regression [8].

3.6 Number of Distortion Sampling Runs

In general, there is a trade-off between the accuracy of the distortion model and the number of trial runs. More trial runs produce more samples, which in turn produce a more accurate model. But performing more trial runs takes more time. So one potential question is how many trial runs does it take to obtain an acceptably accurate model?

In principle, one could use the confidence intervals to answer this question — as the number of samples increases, the sizes of the confidence intervals should decrease. One could therefore periodically recalculate the distortion model and resulting confidence intervals during the sampling phase to stop sampling when the confidence intervals either converge or become acceptably small.

In practice, we found it unnecessary to control the sampling phase this closely. Specifically, we simply let the script that performed the sampling runs execute for about a day for each program before collecting the data and computing the distortion model. For our set of benchmark programs, we found that this approach delivered acceptably accurate distortion models.

3.7 Using the Model

One goal of the distortion model is to allow a user running the program to obtain an estimate of how any early task terminations in that execution affected the accuracy of the result that the program produced. In particular, the user would take the observed early task termination rates $x_1, \ldots, x_n$, apply the distortion model to obtain an estimated distortion $\hat{d}(x_1, \ldots, x_n)$ along with its associated confidence interval to evaluate whether the terminations were likely to have unacceptably distorted the result. In this scenario, several issues are likely to be of interest:

- **Distortion:** How quickly does the distortion grow as a function of the task block skip rates? We evaluate this issue by examining the model coefficients c_i. The smaller the model coefficients, the less the results are affected by early terminations of the corresponding tasks.
- **Bounds:** How small are the confidence intervals? We evaluate this issue by computing the minimum and maximum sizes of the upper confidence intervals at all of the $x_1, \ldots x_n$ points observed during the distortion sampling runs. For a user to accept a distorted result, both the distortion estimate and the upper confidence interval must be small enough to make the likelihood that an unacceptably large actual distortion has occurred remote enough for the user to accept. The upper confidence interval provides the appropriate bound for this purpose — the distortion is inherently bounded below by zero and becomes more acceptable the closer it gets to this bound.

- **Predictive Power:** We use our test inputs to obtain the regression model. We anticipate that users will apply the model to executions of the program running on other inputs. The issue is whether a model derived from executions on one set of inputs can accurately predict the distortion for other inputs. Our experimental results show that, for our set of benchmark applications, our models can accurately predict distortions for unseen inputs [20].
- **Sampling Time:** It is possible for the time required to perform the trial runs to exceed the time gained by the application of early phase termination during production runs. We anticipate usage scenarios in which the overhead of the trial runs is profitably amortized over the production runs, either because the production runs take substantially longer to execute than the trial runs or because the program will be used for many more production runs than trial runs. It may also be feasible to perform the trial runs during a lead time between development and deployment.

3.8 Bias Definition and Use

The distortion measures the absolute error induced by skipping a set of tasks. It is also sometimes useful to consider whether there is any systematic direction to the error. The following quantity b measures the *bias* of the outputs:

$$b = \frac{1}{m} \sum_{i=1}^{m} \frac{o_i - \hat{o}_i}{o_i}$$

Note that this is the same formula as the distortion with the exception that it preserves the sign of the summands. Errors with different signs may therefore cancel each other out in the computation of the bias instead of accumulating as for the distortion.

If there is a systematic bias, it may be possible to compensate for the bias to obtain a more accurate result. Consider, for example, the special case of a program with a single output component o. If we know that the bias at a certain point is b, we can simply divide the observed output $\hat{o}$ by $(1 - b)$ to obtain an estimate of the correct output whose expected distortion is 0.

The reasoning in this example generalizes to handle programs with multiple output components $o_1, \ldots, o_m$ — the key is to generalize our methodology to obtain a separate distortion and bias model for each different output component o_i. It is then possible to correct each output component individually to eliminate the bias for that component. If the output components do, in fact, exhibit a systematic bias in the face of skipped task blocks, the primary obstacle to applying this technique is the number of output components. For programs with large numbers of output components it may be difficult to perform the number of trials required to obtain a useful model of the distortion and bias for each individual output component.

4. Experimental Results

We apply early phase termination to three scientific computations:

- **String:** String uses seismic travel-time inversion to construct a discrete velocity model of the geological medium between two oil wells[12]. 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.
- **Search:** Search is an application from the Stanford Electrical Engineering department [5, 6]. 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 electrons 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.

In addition to these computations, the Jade benchmark suite contains Panel Cholesky, which performs a Cholesky factorization of a sparse matrix; Volume Rendering, which generates a sequence of images of a set of volume data; and Ocean, whose core computation uses an iterative method to solve a set of spatial partial differential equations [17]. Panel Cholesky is not a candidate for early phase termination because it has an irregular concurrency pattern with no easily identifiable serial and parallel phases. Volume Rendering, on the other hand, is a perfect candidate for early phase termination — it uses ray casting to produce two-dimensional projections of a three-dimensional data set [16]. Applying early phase termination to this computation would result in the computation casting somewhat fewer rays, which we expect would not substantially affect the quality of the final image. Unfortunately, we were unable to run Volume Rendering on our current computational platform because of incompatibilities associated with the input and output data file formats.

While it is possible to apply early phase termination to Ocean, it does not improve the performance. Although early phase termination eliminates the overhead associated with barrier idling at the end of each parallel phase, it also causes the phase to drop some relaxation computations, which in turn causes the computation to take more iterations to converge. The net effect is a decrease in performance [20].

4.1 Methodology

For each of our benchmark applications, we perform the following steps to evaluate the effect of early phase termination on the parallel performance and the distortion.

4.1.1 Inputs and Outputs

For each application we choose a set of inputs (the goal is to obtain a set of inputs that reflect how the program will be used in practice). In general, an application may produce multiple outputs, only some of which are important for the user. We therefore select a set of outputs to use in the calculation of the distortion metric (the goal is to select important outputs that motivated the development of the application in the first place).

4.1.2 Distortion Model

We next apply the procedure outlined in Section 3 to obtain a distortion model for each of our programs. In our experiments the target skip rates for our distortion sampling runs range from 0 (skip no execution of the task block) to 0.75 (skip three out of every four executions). For each program we run the script that performed the distortion sampling runs for about a day. The number of sampling runs for each test input varied between approximately 500 and 5000 depending on the program.

4.1.3 Performance

We calculate the parallel performance of each application as follows. We first instrument the application and the Jade implementation to emit a log of events in the computation. These events include the beginning and end of each parallel and serial phase and each task.

We next run the application sequentially on each input. We use the resulting event logs to compute the running time of the parallel computation running on p processors as follows. We first scan the log to find the parallel phases and compute the execution time of each task in each phase. We also compute the time required to perform each serial phase.

We next use an event-driven simulation to compute the amount of time required to execute the computation in parallel on p processors. Specifically, we use a standard dynamic load-balancing algorithm to schedule tasks onto processors — during each parallel phase, the algorithm maintains a list of waiting tasks. When a processor finishes a task, the algorithm selects the next task in the list and assigns it to the newly ready processor for execution. This algorithm has the property that no processor will become idle as long as there are tasks that are waiting to execute. We work with two versions of this scheduling algorithm:

- **Natural:** In most programs there is a natural task execution order. This order corresponds to the order in which a sequential execution of the program would execute the corresponding computations. This order is usually determined by the order in which the sequential computation traverses the data structures that the parallel tasks access. In the Natural scheduler, tasks appear the list of waiting tasks in the natural execution order.

  Note that with the Natural scheduler, early phase termination will tend to terminate sets of tasks that 1) occur next to each other in the natural execution order, and 2) occur near the end of the order. Note that this could be a significant source of bias, since such sets of tasks may tend to operate on 1) related pieces of data, and 2) extreme parts of the computational domain (which may differ significantly from the computational domain as a whole).

  Recall that the distortion sampling runs select the tasks to skip randomly. The assumption behind the resulting probabilistic distortion model is that there is no systematic bias in the set of tasks that early phase selection terminates. One potential problem with the Natural scheduler is the possibility that the bias it exhibits in the terminated tasks could invalidate the distortion model.

- **Random:** In the Random scheduler, tasks appear in the list of waiting tasks in a random order. The goal is to arrange to have the early phase termination algorithm terminate a randomly selected set of tasks at the end of the parallel phase. Note that the Random scheduler may not completely achieve this goal. In particular, longer tasks are more likely to be executing at the end of the parallel phase and are therefore more likely to be terminated by the early phase termination algorithm. For our set of applications this phenomenon does not cause the actual distortion to diverge significantly from the predicted distortion. When applying early phase termination to a new application it may nevertheless be prudent to recognize that this phenomenon could affect the ability of the distortion model to accurately predict the actual distortion.

Without early phase termination, we compute the end time of each parallel phase as the time when the last processor finishes its last task at the end of the parallel phase. With early phase termination there are two possibilities. If the parallel phase contains tasks from critical task blocks, we compute the end time of the parallel phase as described above for the version without early phase termination. If the parallel phase does not contain tasks from critical task blocks, we compute the end time as the time when the first processor to become idle at the end of the parallel phase completes its last task. The time required to perform each parallel phase is then simply the end time of the phase minus the start time of the phase (the time when the previous serial phase finished). Given a time for each parallel phase, we compute the overall parallel execution time as the sum of the times required to complete all of the parallel and serial phases. We then divide the total sequential execution time (the sum of the times required to complete all of the tasks and serial phases) by the overall parallel execution time to obtain the speedup. We calculate three different parallel speedup measures:

- **Optimized Speedup:** The speedup with early phase termination.
- **Standard Speedup:** The speedup without early phase termination (all tasks in parallel phases execute to completion).
- **Balanced Speedup:** The hypothetical speedup of a computation that executes all tasks to completion with perfect load balancing in parallel phases. The overall parallel execution time is the sum of the parallel task execution times divided by the number of processors p plus the time spent in serial phases. The speedup is the total sequential execution time divided by this quantity.

To appreciate the value of the balanced speedup measure, understand that there are two distinct ways in which early phase termination can improve the performance. First, it can improve the overall load balance by eliminating idle time at the end of parallel phases. Second, it can cause the computation to perform less work by eliminating computation in terminated tasks. The balanced speedup measure separates these two effects by presenting the speedup that would result if the load balance was perfect. Any remaining performance increase is caused by a reduction in the amount of work that the computation performs.

This simulation assumes that it takes no time to manage the parallel computation — to create, schedule, and synchronize the tasks. While these activities clearly must take some time, in practice we expect the associated overhead to be amortized away to negligible levels given reasonable task sizes (which our benchmarks have). A potentially more problematic assumption is that the parallel machine provides a low-overhead mechanism that one processor (specifically, the first processor to go idle at the end of the parallel phase) can use to terminate tasks running on other processors. While most parallel machines provide such a mechanism, it is typically available only via a relatively high overhead interaction with the operating system. Exposing such mechanisms more directly could reduce the overhead of implementing early phase termination. One factor that should simplify the development of an effective task termination mechanism is that the computation should be relatively insensitive to the latency of the mechanism. Specifically, it should be feasible to overlap the execution of the remaining tasks from one parallel phase with the execution of the next serial phase while the mechanism terminates the remaining parallel tasks.

4.1.4 Distortion

We calculate the distortion for each application as follows. We instrument the event-driven simulation to print, at the end of each parallel phase, the identifiers of all of the tasks that are still executing when the first processor to become idle at the end of the phase completes its last task (these are the tasks that early phase termination would terminate). We then rerun the computation, but configure the Jade run-time system to use these identifiers to skip all of the corresponding tasks. We then use the results from the original sequential execution (which executed all tasks to completion) to compute the resulting distortion caused by skipping terminated tasks.

This distortion calculation assumes that terminated tasks have no effect whatsoever on the overall computation. On message-passing machines one could easily achieve this effect by simply discarding the updates from terminated tasks (these updates would all be resident in the memory of the machine executing the task) [21]. On shared-memory machines, one could use transactional memory [13, 1, 2] to achieve this effect — simply make each task execute as a transaction.

We anticipate, however, that for many applications it should be viable to simply let each task's effects become immediately visible. This approach would include the effects of more of the computation into the final result and should therefore improve the accuracy of this result. One issue is the potential for data structure corruption if the system happens to terminate a task in the middle of an update. It is possible to address this issue by enforcing a minimal acceptable atomicity granularity for each data structure update, with either the developer or a program analysis algorithm identifying an appropriate granularity [18]. For our set of benchmark programs, this issue does not arise at all — it is possible to terminate any parallel task at any point without corrupting the data structures or unacceptably distorting the final result (the only distortion would be the anticipated distortion that comes from discarding the remaining computation in the task).

Note that including more of the computation into the final result can affect the accuracy of any applied bias compensation. Because this technique attempts to compensate for any missing computation, including the effects of additional computation into the starting point can result in overcompensation. Transactional task execution eliminates this issue. It may also be possible to base the bias compensation not on the number of tasks that execute to completion, but on an estimate of the percentage of the total computation whose effects were included into the starting point for the compensation calculation. This percentage would include, of course, those parts of each terminated task that executed before the system terminated the task at the end of the parallel phase.

4.2 String

String repeatedly traces a set of rays from one well to the other. The velocity model between the two wells determines both the path and the simulated travel time of each ray. The computation records the difference between the simulated and measured travel times. It backprojects the difference linearly along the path of the ray. At the end of the phase the computation averages the backprojected differences to construct an improved velocity model. The process continues for a specified number of iterations.

We ran String on four input files: A.pro, B.pro, C.pro, and D.pro. A.pro and B.pro have the same seismic data and starting velocity models but different ray tracing parameters (these parameters control the density and orientation of the set of traced rays). Similarly, C.pro and D.pro also have the same seismic data and starting velocity models (which are different from the seismic data and starting velocity models for A.pro and B.pro) but different ray tracing parameters. The output is the new velocity model for the geology between the oil wells. This output is a large matrix of numbers; the size of this matrix varies depending on the size of the starting velocity model.

4.2.1 Task Blocks and Criticality Testing

String has five task blocks. Executions of the first task block shoot rays through the current velocity model, storing their intermediate results into blocks of storage allocated for that purpose. Executions of the second task block combine these results into a single block of storage that executions of the third task block use to compute a new velocity model. The fourth task block creates data structures used in the remaining computation; the fifth task block deallocates these data structures at the end of the computation.

Our criticality testing runs revealed that the first two task blocks are skippable, while the third and fourth task blocks are critical. We attribute the criticality of the third task block to the fact that skipping its executions leaves some of the values of the old velocity model in place, which causes significant distortion. The fourth task block is critical because skipping its tasks leaves the remaining computation without a place to store some of its results. Skipping tasks from the fifth task block has no effect at all on the result, but leaves the data structures allocated at the end of the computation.

4.2.2 Performance Results

Table 1 presents the performance results for String running with the Random and Natural schedulers. The table has several columns:

- **Application:** The name of the application — String, Water, or Search.
- **Processors:** The number of processors executing the computation.
- **Input:** The input for the computation.
- **Optimized Speedup, Random Scheduler:** The speedup for the computation using the Random scheduler and early phase termination.
- **Standard Speedup, Random Scheduler:** The speedup for the computation using the Random scheduler when every parallel task executes to completion.
- **Optimized Speedup, Natural Scheduler:** The speedup for the computation using the Natural scheduler and early phase termination.
- **Standard Speedup, Natural Scheduler:** The speedup for the computation using the Natural scheduler when every parallel task executes to completion.
- **Balanced Speedup:** The speedup for the hypothetical perfectly load balanced computation in which every task would execute to completion with no idle time in parallel phases.

The performance results show that String is inherently well suited for parallel execution — the vast majority of the execution time is spent in parallel phases and the load is reasonably well balanced. Under these circumstances early phase termination is useful primarily for larger numbers of processors.

We note that the speedups are close to identical for both schedulers. The Balanced Speedup numbers indicate that the performance increase for early phase termination is roughly evenly split between an improvement in the load balance and reduction in the amount of work that the computation performs. Several runs have superlinear speedup; this is possible because early phase termination can cause the parallel version to perform less work than the sequential version.

4.2.3 Distortion Model

Figure 2 presents the distortion model for String. We present each regression coefficient in the form $c[\pm e]$, where c is the coefficient itself and $[\pm e]$ provides the 95% confidence bounds for that coefficient. The F value for this model is in the thousands, the corresponding p value is less than 0.0001, and the R^2 value is around 44%, which reflects the fact that the representative inputs used to obtain this model have somewhat different distortion properties. The maximum upper 95% confidence bound for any sampled task block skip point x_1, x_2 is 0.0229. An appropriate probabilistic bound on the distortion at, for example, a 50% task skip rate for executions of task blocks 1 and 2 is therefore approximately 0.063. The coefficients c_1 and c_2 are very small, which indicates that the distortion is hardly affected at all by skipping tasks. It is therefore possible to skip a large proportion of the tasks without incurring substantial distortion.

4.2.4 Distortion Results

Table 2 presents the distortion results for String running with the Random and Natural schedulers. In addition to the Application, Processors, and Input columns, which have the same meaning as in the performance tables, the table has the following columns:

- **Actual Distortion, Random Scheduler:** The distortion for the computation using the Random scheduler and early phase termination.

Application	Processors	Input	Optimized Speedup, Random Scheduler	Standard Speedup, Random Scheduler	Optimized Speedup, Natural Scheduler	Standard Speedup, Natural Scheduler	Balanced Speedup
String	60	A.pro	60.34	53.25	60.33	53.16	56.66
String	80	A.pro	82.30	69.60	82.33	69.51	74.14
String	100	A.pro	100.90	82.42	100.76	82.21	90.99
String	120	A.pro	129.67	100.63	129.60	100.42	107.24
String	60	B.pro	61.37	54.02	61.36	54.07	57.52
String	80	B.pro	84.30	70.99	84.27	71.10	75.63
String	100	B.pro	103.78	84.31	103.70	84.48	93.24
String	120	B.pro	134.64	103.58	134.62	103.86	110.39
String	60	C.pro	53.94	48.21	53.94	48.27	51.00
String	80	C.pro	70.86	61.27	70.85	61.37	64.70
String	100	C.pro	84.11	70.92	84.09	71.05	77.14
String	120	C.pro	103.24	83.96	103.21	84.24	88.49
String	60	D.pro	56.53	50.24	56.53	50.29	53.27
String	80	D.pro	75.37	64.54	75.41	64.70	68.43
String	100	D.pro	90.67	75.42	90.64	75.58	82.51
String	120	D.pro	113.00	90.23	113.25	90.70	95.63

Table 1. Performance Results for String

$$\hat{d}(x_1, x_2)=0.0004[\pm 0.0005] + 0.033[\pm 0.003]x_1 + 0.033[\pm 0.003]x_2$$

Figure 2. Distortion Model for String

Application	Processors	Input	Actual Distortion, Random Scheduler	Actual Distortion, Natural Scheduler	Predicted Distortion
String	60	A.pro	0.0058	0.0291	0.0091[± 0.0237]
String	80	A.pro	0.0074	0.0396	0.0120[± 0.0237]
String	100	A.pro	0.0133	0.0421	0.0149[± 0.0237]
String	120	A.pro	0.0134	0.0437	0.0179[± 0.0237]
String	60	B.pro	0.0086	0.0175	0.0091[± 0.0237]
String	80	B.pro	0.0094	0.0259	0.0120[± 0.0237]
String	100	B.pro	0.0101	0.0293	0.0149[± 0.0237]
String	120	B.pro	0.0103	0.0331	0.0179[± 0.0237]
String	60	C.pro	0.0014	0.0026	0.0091[± 0.0237]
String	80	C.pro	0.0017	0.0030	0.0120[± 0.0237]
String	100	C.pro	0.0021	0.0027	0.0149[± 0.0237]
String	120	C.pro	0.0020	0.0028	0.0179[± 0.0237]
String	60	D.pro	0.0008	0.0019	0.0091[± 0.0237]
String	80	D.pro	0.0015	0.0026	0.0120[± 0.0237]
String	100	D.pro	0.0018	0.0036	0.0149[± 0.0237]
String	120	D.pro	0.0015	0.0045	0.0179[± 0.0237]

Table 2.
Distortion Results for String

- **Actual Distortion, Natural Scheduler:** The distortion for the computation using the Natural scheduler and early phase termination.
- **Predicted Distortion:** The predicted distortion of the computation from the distortion model for the application. We present the predicted distortion in the form $d[\pm e]$, where d is the predicted distortion itself and $[\pm e]$ provides the confidence bounds for that distortion.

With both schedulers, the actual distortions for inputs A.pro and B.pro are always larger than the corresponding distortions for inputs C.pro and D.pro. We attribute this difference to differences in the input files (A.pro and B.pro share an input file; C.pro and D.pro share a different input file). These data underscore the need to use representative inputs for the distortion sampling runs if one is to obtain an accurate distortion model.

With the Random scheduler, the actual distortions for inputs A.pro and B.pro track the predicted distortions. The actual distortions for inputs C.pro and B.pro are always smaller than the predicted distortions. The accuracy bounds are reasonably tight and the distortion is always small. It is possible to use early phase termination for this application for all numbers of processors in our experiments.

We note that with the Natural scheduler, the actual distortions for inputs A.pro and B.pro appear to be somewhat larger than the model predicts. This phenomenon may be due to a correlation between the terminated tasks in the runs that use the Natural scheduler. Specifically, the Natural scheduler tends to leave a set of tasks that trace adjacent rays executing together at the end of the corresponding parallel phases. This correlation may tend to undermine the redundancy that keeps the overall distortion small in the distortion sampling runs and the runs that use the Random scheduler.

4.3 Water

We ran Water on four different inputs; the inputs vary in the number of molecules they cause Water to simulate. Specifically, the inputs produce simulations of 343, 512, 729, and 1000 molecules. Water calculates several values of potential interest, including the total energy, kinetic energy, and potential energy. We choose to measure the distortion of the total energy (in part because it includes contributions from all of the other partial energy calculations); it would be possible to extend this measure to include the different partial energy values explicitly.

4.3.1 Task Blocks and Criticality Testing

Water has four task blocks. Executions of the first task block compute the intermolecular forces between pairs of molecules, storing their intermediate results into blocks of storage allocated for that purpose. Executions of the second task block sum up all of the intermediate results to produce the final intermolecular forces. Similarly, executions of the third task block compute the intermolecular contributions to the potential energy, storing intermediate results into blocks of storage allocated for that purpose. Executions of the fourth task block sum up the intermediate results to produce the final intermolecular potential energy.

Our criticality testing experiments revealed no critical task blocks — all task blocks produce a mean distortion of less than 0.1 when skipped at the target 10% rate during our criticality testing runs.

4.3.2 Performance Results

Table 3 presents the performance results for Water running with the Random and Natural schedulers. As for String, the speedups are comparable for both schedulers. Note that the speedup increases as the input size increases, indicating that the computation spends relatively more time in parallel phases as the number of molecules increases.

The Balanced Speedup numbers indicate that the performance increase for early phase termination is split more or less evenly between an improvement in the load balance and reduction in the amount of work that the computation performs. Unlike String, the reduction in work does not cause any of the runs to exhibit superlinear speedup.

4.3.3 Distortion Model

Figure 3 presents the distortion model for Water. The F value for this model is in the tens of thousands, the corresponding p value is less than 0.0001, and the R^2 value is above 98%, which indicates that the model explains over 98% of the variation in the data. The maximum upper 95% confidence bound for any sampled task block skip point $x_1, \ldots, x_4$ is 0.586. Given a task block skip point $x_1, \ldots, x_4$, the quantity $\hat{d}(x_1, \ldots, x_4) + 0.0586$ provides an appropriate upper bound for the actual distortion.

The model coefficients are relatively large, which indicates that the distortion increases relatively quickly when the computation skips tasks. For example, the coefficient for x_3 is 0.56. So, for example, a 10% skip rate for task block 3 results in a 0.056 increase in the estimated distortion. If the skip rate is 50% for all task blocks, the estimated distortion is approximately 0.6 — in other words, the observed output with skipped tasks is estimated to be more than a factor of two different from the correct output! The models would therefore appear to indicate that the vast majority of the tasks in Water must execute to completion for the computation to produce an acceptable output. Note, however, that Section 4.3.5 describes a way to compensate for the bias in the output to obtain a new output with an estimated distortion of zero. With this bias compensation, Water can tolerate much higher task block skip rates.

4.3.4 Distortion Results

Table 4 presents the distortion results for Water running with the Random and Natural schedulers. With the Random scheduler, the actual distortions for all inputs closely track the predicted distortions and the accuracy bounds are reason-

Application	Processors	Input	Optimized Speedup, Random Scheduler	Standard Speedup, Random Scheduler	Optimized Speedup, Natural Scheduler	Standard Speedup, Natural Scheduler	Balanced Speedup
Water	60	343	40.09	33.40	41.94	33.61	37.37
Water	80	343	47.99	38.54	49.53	38.08	44.17
Water	100	343	54.45	42.52	56.16	41.58	49.60
Water	120	343	60.06	45.71	63.68	45.25	54.01
Water	60	512	44.12	38.42	45.57	38.90	41.84
Water	80	512	53.84	45.50	56.17	46.29	50.59
Water	100	512	62.19	51.14	64.53	51.98	57.86
Water	120	512	69.37	55.76	72.54	56.60	63.98
Water	60	729	46.33	41.89	47.20	42.08	44.64
Water	80	729	57.35	50.57	58.25	50.53	54.76
Water	100	729	66.93	57.83	68.73	58.42	63.39
Water	120	729	75.36	63.87	77.62	64.49	70.83
Water	60	1000	49.63	45.73	50.36	45.17	48.14
Water	80	1000	62.51	55.73	63.56	55.11	60.15
Water	100	1000	73.95	64.15	74.93	63.02	70.74
Water	120	1000	84.39	71.25	86.54	70.48	80.16

Table 3. Performance Results for Water

$$\hat{d}(x_1, \ldots, x_4) = 0.011[\pm 0.0006] + 0.052[\pm 0.0035]x_1 + 0.11[\pm 0.0035]x_2 + 0.56[\pm 0.0035]x_3 + 0.54[\pm 0.0035]x_4$$

Figure 3. Distortion Model for Water

Application	Processors	Input	Actual Distortion, Random Scheduler	Actual Distortion, Natural Scheduler	Predicted Distortion
Water	60	343	0.2471	0.0629	0.2281[± 0.0581]
Water	80	343	0.3088	0.0927	0.3017[± 0.0581]
Water	100	343	0.3441	0.1203	0.3753[± 0.0581]
Water	120	343	0.4068	0.1631	0.4488[± 0.0581]
Water	60	512	0.1491	0.0334	0.1564[± 0.0581]
Water	80	512	0.1997	0.0616	0.2057[± 0.0581]
Water	100	512	0.2512	0.0816	0.2550[± 0.0581]
Water	120	512	0.2859	0.0876	0.3043[± 0.0581]
Water	60	729	0.0918	0.0235	0.1131[± 0.0581]
Water	80	729	0.1283	0.0155	0.1478[± 0.0581]
Water	100	729	0.1549	0.0357	0.1824[± 0.0581]
Water	120	729	0.1913	0.0453	0.2170[± 0.0581]
Water	60	1000	0.0741	0.0223	0.0855[± 0.0581]
Water	80	1000	0.1011	0.0247	0.1107[± 0.0581]
Water	100	1000	0.1268	0.0155	0.1359[± 0.0581]
Water	120	1000	0.1557	0.0355	0.1612[± 0.0581]

Table 4.
Distortion Results for Water

Application	Processors	Input	Actual Distortion After Bias Compensation, Random Scheduler	Actual Distortion After Bias Compensation, Natural Scheduler
Water	60	343	0.0256	0.2127
Water	80	343	0.0113	0.2978
Water	100	343	0.0486	0.4064
Water	120	343	0.0751	0.5167
Water	60	512	0.0077	0.1447
Water	80	512	0.0065	0.1802
Water	100	512	0.0040	0.2314
Water	120	512	0.0254	0.3101
Water	60	729	0.0230	0.1000
Water	80	729	0.0217	0.1539
Water	100	729	0.0325	0.1781
Water	120	729	0.0318	0.2180
Water	60	1000	0.0114	0.0680
Water	80	1000	0.0097	0.0955
Water	100	1000	0.0095	0.1382
Water	120	1000	0.0054	0.1487

Table 5.
Actual Distortion for Water After Bias Compensation

ably tight. Because the distortions are large for large numbers of processors, this application requires the use of bias compensation (see Section 4.3.5) to make early phase termination practical as the number of processors grows.

With the Natural scheduler, however, the actual distortions differ substantially from the predicted distortions. In fact, they are substantially lower than the predicted distortions. While this property makes the results more accurate than expected, it also causes the bias compensation mechanism to overcompensate, resulting in less accurate results after bias compensation (see Section 4.3.5).

We attribute this deviation from the predicted distortion to the fact that the Natural scheduler tends to cause the run-time system to terminate a set of tasks that compute values for molecules in extreme points in the computational domain. These molecules apparently make smaller contributions to the total energy than molecules that are closer to the center of the domain. The net result is that the total energy changes less than the distortion model (which is based on eliminating computations from randomly selected molecules) predicts.

4.3.5 Bias Compensation

It turns out that, for every execution of Water, the bias is the same as the distortion (which implies that the bias estimator $\hat{b}(x_1, \ldots, x_4)$ equals the distortion estimator $\hat{d}(x_1, \ldots, x_4)$). Because Water has a single output, it is possible to compensate for the bias by simply dividing the observed output by $(1 - \hat{d}(x_1, \ldots, x_4))$ to obtain an output estimator with an expected distortion of zero and a confidence interval of the same size as the confidence interval of the distortion.

Table 5 presents the distortion results for Water with the Random scheduler and Natural schedulers after bias compensation. In addition to the Application, Processors, and Input columns, which have the same meaning as in the performance and distortion tables, the table has the following columns:

- **Actual Distortion After Bias Compensation, Random Scheduler:** The distortion for the computation after bias compensation using the Random scheduler and early phase termination.
- **Actual Distortion After Bias Compensation, Natural Scheduler:** The distortion for the computation after bias compensation using the Natural scheduler and early phase termination.

The numbers show that, with the Random scheduler, bias compensation eliminates much of the distortion, enabling the use of early phase termination for this application for large numbers of processors. With the Natural scheduler, however, the deviation from the predicted distortion makes bias compensation impractical.

4.4 Search

Search simulates the interaction of solids and electron beams. It uses a Monte-Carlo approach to trace the paths that a set of electrons take through the solid. It counts the number of electrons that emerge back out of the solid and (implicitly) the number that remain trapped inside.

We ran Search on four different inputs; the inputs vary in the backscattering parameters, in particular in the num-

Application	Processors	Input	Optimized Speedup, Random Scheduler	Standard Speedup, Random Scheduler	Optimized Speedup, Natural Scheduler	Standard Speedup, Natural Scheduler	Balanced Speedup
Search	60	1	60.21	52.36	54.19	47.64	56.49
Search	80	1	81.09	67.43	71.56	60.22	73.85
Search	100	1	100.06	79.84	85.28	70.52	90.55
Search	120	1	125.09	94.69	104.37	81.91	106.62
Search	60	2	58.38	51.25	58.55	51.01	54.78
Search	80	2	76.99	65.06	78.87	65.69	70.95
Search	100	2	95.18	77.52	96.18	78.38	86.21
Search	120	2	116.69	90.66	122.16	92.93	100.65
Search	60	3	59.36	51.89	59.71	51.66	55.81
Search	80	3	79.06	66.80	81.01	66.88	72.69
Search	100	3	98.70	79.24	99.84	79.49	88.81
Search	120	3	121.22	92.79	127.09	95.48	104.21
Search	60	4	62.29	53.59	62.65	53.65	58.22
Search	80	4	84.30	69.60	86.04	70.01	76.85
Search	100	4	105.62	83.26	108.05	84.61	95.12
Search	120	4	132.24	98.39	139.53	101.46	113.03

Table 6. Performance Results for Search

$$\hat{d}(x_1)=0.005[\pm 0.0002] + 0.11[\pm 0.0007]x_1$$

Figure 4. Distortion Model for Search

Application	Processors	Input	Actual Distortion, Random Scheduler	Actual Distortion, Natural Scheduler	Predicted Distortion
Search	60	1	0.0271	0.0253	0.0194[± 0.013]
Search	80	1	0.0314	0.0340	0.0243[± 0.013]
Search	100	1	0.0349	0.0353	0.0292[± 0.013]
Search	120	1	0.0389	0.0393	0.0341[± 0.013]
Search	60	2	0.0259	0.0260	0.0194[± 0.013]
Search	80	2	0.0267	0.0328	0.0243[± 0.013]
Search	100	2	0.0352	0.0348	0.0292[± 0.013]
Search	120	2	0.0399	0.0419	0.0341[± 0.013]
Search	60	3	0.0248	0.0293	0.0194[± 0.013]
Search	80	3	0.0263	0.0299	0.0243[± 0.013]
Search	100	3	0.0350	0.0373	0.0292[± 0.013]
Search	120	3	0.0395	0.0451	0.0341[± 0.013]
Search	60	4	0.0165	0.0178	0.0194[± 0.013]
Search	80	4	0.0192	0.0210	0.0243[± 0.013]
Search	100	4	0.0217	0.0267	0.0292[± 0.013]
Search	120	4	0.0263	0.0252	0.0341[± 0.013]

Table 7.
Distortion Results for Search

ber of electron paths that they simulate. The application calculates and outputs a backscattering coefficient for 51 different solid/energy level pairs; this coefficient indicates the percentage of the electrons that escape back out of the solid. We take the resulting sequence of 51 backscattering coefficients as the output of the application.

4.4.1 Task Blocks and Criticality Testing

Search has one task block which uses a Monte-Carlo simulation to trace the paths of the electrons through the solid. Our criticality testing experiments revealed that this task block is skippable since it produced a mean distortion of less than 0.1 when skipped at the target 10% rate during our criticality testing runs.

4.4.2 Performance Results

Table 6 presents the performance results for Search running with the Random and Natural schedulers, respectively. As for String and Water, the speedups are comparable for both schedulers. The results show that Search, like many Monte-Carlo simulations, can benefit substantially from parallel execution.

The Balanced Speedup numbers indicate that the performance increase for early phase termination is split more or less evenly between an improvement in the load balance and reduction in the amount of work that the computation performs. Note that the reduction in work associated with early task termination causes some of the runs to exhibit superlinear speedup.

4.4.3 Distortion Model

Figure 4 presents the distortion model for Search. The F value for this model is in the tens of thousands, the corresponding p value is less than 0.0001, and the R^2 value is above 96%, which indicates that the model explains over 96% of the variation in the data. The maximum upper 95% confidence bound for any sampled task block skip point x_1 is 0.0136. An appropriate bound on the distortion at, for example, a 25% task skip rate is therefore approximately 0.039.

The coefficient c_1 is relatively small, which indicates that the distortion increases relatively slowly as the computation skips more tasks. In particular, a 10% skip rate for task block 1 results in approximately 0.01 increase in the estimated distortion. The application is therefore relatively resilient to skipping tasks.

4.4.4 Distortion Results

Table 7 presents the distortion results for Search running with the Random and Natural schedulers. Unlike String and Water, the choice of scheduler has no impact on the distortion. This makes sense since the tasks differ only in the random numbers that they use to drive their part of the Monte-Carlo simulation. One would therefore expect there to be no correlation between the position of the task in the natural execution order and the effect of the task on the computation. The experimental results are consistent with this expectation.

The actual distortions closely track the predicted distortions, the accuracy bounds are reasonably tight, and the distortion is always small. In particular, it is always smaller than the distortion of approximately 0.08 that results from simply changing the random number seed for the Monte-Carlo simulation. It is possible to use early phase termination for this application for all numbers of processors in our experiments.

4.5 Discussion

In retrospect, it is possible to reconstruct the reasons why early task terminations cause the applications in our study to behave the way they do. The skippable tasks in Water, Search, and String all either compute a set of contributions that are combined to obtain a final result, or combine these contributions to produce the final result. Although the applications themselves perform complex, detailed computations, it is possible to come up with a relatively simple high-level characterization of the behavior of each application that explains the observed results.

At a high level, the tasks in Search essentially sample a population of electron paths. The net effect of skipping a task is to discard the samples that the task would have performed. The net effect of performing fewer samples is that the resulting estimate of the property of interest in the population may be somewhat less accurate. The coefficient c_1 in Search's distortion model indicates that skipping half the tasks (in effect, sampling half of the points in the sample space) can cause a distortion of around 0.06. To place this loss of accuracy in perspective, consider that simply changing the random number seed that drives the Monte Carlo simulation in Search can cause a distortion of 0.08 in the standard computation.

At a high level, the skippable tasks in String either sample a population of rays projected through the velocity model of the geology between two oil wells or combine the results of this sampling process. Skipping tasks therefore has the effect of discarding projected rays. Because the combination operator averages the contributions, there is no bias and early task terminations have little effect on the accuracy of the final computation.

At a high level, Water essentially computes sums of positive numbers. The net effect of terminating tasks early is to remove some of the positive numbers from the sums. On average, the resulting relative reduction in the values of the sums will be roughly proportional to the percentage of numbers removed from the sums. The coefficients in the distortion model capture the relative contribution of each partial sum to the final output total energy of the system of water molecules. Because all of the summed numbers are positive, it is possible to model the bias as a linear function of the task skip rates and apply that bias to correct the output. One can view the resulting computation as selecting a subset of the numbers to sum, computing the sum of that subset, then using the size of the subset to extrapolate the partial sum to obtain an estimate of the sum of all of the numbers.

4.5.1 Redundancy

One of the reasons that our computations work well with early phase termination is that they have some inherent redundancy. In String, the redundancy comes from the multiple rays that it traces through the velocity model — if the computation loses some rays, the remaining rays usually cover enough of the velocity model to deliver an acceptable result. If Search loses some traced electrons, the remaining electrons usually cover enough of the possible interactions with the solid for the computation to deliver an acceptable result.

Water has a different kind of redundancy — different molecules have correlated interactions. This correlation makes it possible to sample some of the interactions, then extrapolate to obtain an accurate estimate of the final result. The distortion results from the Natural scheduler highlight the importance of choosing an appropriate unbiased set of sampled interactions.

In all of these applications, redundancy comes with increased computation — String traces more rays, Water computes more interactions, and Search traces more electrons than they need to generate acceptable results. Note that early phase termination, by dropping some of these subcomputations, consumes some of this redundancy. One may very well wonder if this is the best way to exploit this redundancy. One could instead, for example, simply change the program or its input to reduce the amount of computation up front. An advantage of early phase termination is that it dynamically tailors its consumption of redundancy to the specific characteristics of the executing parallel computation. It can therefore maximize the delivered performance benefit — in addition to reducing the amount of computation, it also eliminates the idle time associated with load imbalances in the underlying computation. Our results show that, for our set of benchmarks, the phenomena have comparable effects on the performance. And early phase termination has the additional benefit that it can eliminate the possibility of incurring very substantial performance decreases if anomalies such as slow processors or unequal assignments of computation to tasks cause drastic load imbalances.

4.5.2 User Acceptance

We anticipate that two different aspects of our approach will affect user acceptance. First, the presence of the probabilistic distortion model will enable users to quantitatively evaluate the impact of early phase termination on the results that their parallel computations produce. Most scientific computations are designed to produce only an approximate solution to a complex set of equations; users typically understand and accept the imprecision inherent in the underlying computational model and method. We anticipate that the distortion model will allow users to quantitatively evaluate the magnitude of any additional imprecision, and that in many cases the result will remain acceptably accurate.

Second, we anticipate that many users will develop a qualitative understanding of the effect of early phase termination on their computation, and that this understanding will make it easier for them to accept the result that the computation produces. Both String and Search, for example, have an inherent precision versus execution time tradeoff. For String, tracing more rays through the geological medium between the oil wells takes more time but delivers a more precise result. For Search, simulating more electron paths requires more time but, once again, delivers a more precise result. Both of these applications accept configuration files that allow users to control either the number of traced rays (String) or simulated electron paths (Search). Users routinely manipulate their configuration files to manage the execution time, understanding that reducing the number of traced rays or simulated electron paths may change the overall precision of the result that the computations produce.

Note that early phase termination has basically the same overall effect on the computation as using the configuration file to change the number of traced rays or simulated electron paths. For String, terminating tasks early reduces the number of traced rays. For Search, terminating tasks early reduces the number of simulated electron paths. And the benefit (a reduction in the execution time) is the same. Moreover, early phase termination provides the additional benefit of terminating precisely those tasks that deliver the greatest improvement in the execution time while minimizing the amount of discarded computation.

Based on our experience with the developers and users of String and Search [17], we believe that the probabilistic distortion models would, by themselves, provide enough information for users to accept the use of early phase termination for their production runs. Moreover, once the users understood the overall effect of early phase termination on their computations, we believe they would actually welcome its use for the production runs.

For Water, the use of early phase termination in combination with bias compensation has the effect of converting a full computation of all interactions into a randomized computation that samples a (usually large) subset of the interactions, then uses extrapolation to compute an approximation of the final result. Given the resulting small distortions and tight accuracy bounds for this application, we anticipate that the probabilistic accuracy models would enable users to accept the results. The results also suggest that random sampling plus extrapolation may be a worthwhile general technique for reducing computation times in other computations.

4.5.3 Implications for Other Programs

In general, we anticipate that many computations will turn out to have the same general pattern as String, Water, and Search. Many computer graphics computations have this high-level pattern [14], as do information retrieval computations [7]. And of course other scientific computations share this pattern [3]. We anticipate that our technique can be applied to eliminate barrier idling in these kinds of computations as well as the scientific computations discussed in this paper.

We applied early phase termination to existing parallel programs that already had, at the cost of some development effort, been engineered to balance the load more or less evenly across modest numbers of processors. The integration of early phase termination into the initial development of other parallel programs may make it possible to obtain an acceptable load balance with less engineering effort. For both existing and new applications, another benefit is the protection early phase termination can provide against extreme imbalances caused either by an anomalously uneven distribution of work across tasks or by computing environment effects such as heterogeneous processors with different computing speeds.

5. Related Work

Asynchronous iteration [9] relaxes standard ordering constraints in iterative solvers to allow parallel processors to proceed without synchronization, typically as they operate on different regions of the problem that potentially share border elements. The lack of synchronization introduces nondeterminism into the computation as different processors asynchronously read and write the same (typically border) elements. Our technique, in contrast, completely eliminates some computations rather than running computations asynchronously at variable execution rates.

Barrier idling has been long recognized as an issue in parallel computing. Fuzzy barriers [11] address this problem by identifying additional instructions that an otherwise idle processor can execute while it is waiting at a barrier. To work well, fuzzy barriers require the availability of enough additional instructions to hide the load imbalance otherwise responsible for the barrier idling. Early phase termination differs in that it is appropriate for arbitrarily unbalanced computations and may change the result that the program produces.

Several existing systems obtain additional robustness in the face of errors by discarding problematic tasks. MapReduce discards records that cause the record processing task to fail multiple times [7]. We know of a graphics rendering algorithm that discards computations associated with problematic triangles instead of including complex special-case code that attempts to render the triangle into the scene [14]. These systems differ from ours in that 1) the motivation is primarily robustness, not performance improvements, and 2) they provide no indication of the effect of discarding computations on the resulting outputs.

We initially developed our distortion models to provide accuracy bounds for programs that tolerate hardware or software failures by discarding tasks [20]. In addition to the distortion models, we also developed timing models. Together, the distortion and timing models can help users successfully apply strategies that purposefully discard tasks to reduce the amount of work that the computation performs (and hence the amount of time it takes to perform the computation) while keeping the resulting distortion within acceptable bounds. The technique presented in this paper also accepts bounded distortion in return for better performance. But the primary purpose of the technique presented in this paper is to eliminate barrier idling, not to reduce the total amount of work that the computation performs (although it may have this effect in practice).

6. Conclusion

Barrier idling can impair the scalability and performance of parallel computations. We have presented a technique, early phase termination, that can completely eliminate barrier idling at the cost of some distortion in the result that the application produces. If this distortion is small enough and predictable enough, users may be willing to accept the distortion in return for the performance increase.

Our results with several benchmark applications show that early phase termination can improve the scalability of the computation. Moreover, when the computation interacts well with early phase termination, the resulting distortion is small and predictable — our distortion models accurately predict the distortion and provide reasonable accuracy bounds. These models therefore enable users to 1) determine if early phase termination is appropriate for their computation and satisfies their accuracy needs, and 2) if so, to confidently determine when a computation that uses early phase termination has produced an acceptable result.

Finally, we have identified a computation pattern that explains why our applications work well with early phase termination. Given that other classes of applications share this pattern, it may very well prove to be fruitful to explore the use of this technique (as well as other techniques that leverage the ability to tolerate subcomputation failures) more widely.

7. Acknowledgements

I would like to thank Gilbert Rinard for drawing my attention to the necessary presence of redundancy in computations that can sustain failures and still produce accurate outputs.

8. Support

This research was supported in part by the Singapore-MIT Alliance, DARPA Cooperative Agreement FA 8750-04-2-0254, NSF Grant CCR-0086154, NSF Grant CCR-0341620, NSF Grant CCF-0209075, and NSF Grant CCR-0325283.

References

[1] C. Ananian and M. Rinard. Efficient object-based software transactions. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct. 2005.

[2] C. S. Ananian. *Architectural and Compiler Support for Strongly Atomic Transactional Memory*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2007.

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

[4] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, Nov. 1992.

[5] R. Browning, T. Li, B. Chui, J. Ye, R. Pease, Z. Czyzewski, and D. Joy. Empirical forms for the electron/atom elastic scattering cross sections from 0.1-30keV. *J. Appl. Phys.*, 76(4):2016–2022, Aug. 1994.

[6] 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.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[8] R. Freund and R. Littell. *SAS System for Regression*. SAS Publishing, 2000.

[9] A. Frommer and D. Szyld. On asynchronous iterations.

[10] J. Goodman. *Chemical Applications of Molecular Modeling*. Royal Society of Chemistry, 2007.

[11] R. Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Apr. 1989.

[12] 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*, pages 82–85, 1990.

[13] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[14] T. Kay and J. Kajiya. Ray tracing complex scenes. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 20(4):269–78, Aug. 1986.

[15] C. Moler. *Numerical Computing with Matlab*. Society for Industrial and Applied Mathematics, 2004.

[16] 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.

[17] 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.

[18] M. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 19(4), Nov. 1999.

[19] M. Rinard. Exploring the acceptability envelope. In *2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '05 Companion) Onwards! Session*, Oct. 2005.

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

[21] D. Scales and M. S. Lam. Transparent fault tolerance for parallel applications on networks of workstations. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.