Improving the Performance of Runtime Parallelization

Shun-Tak Leung and John Zahorjan

Department of Computer Science and Engineering
University of Washington

Technical Report 92-10-05
Revised February 1993

(This paper will appear in the Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.)

Improving the Performance of Runtime Parallelization

Shun-tak Leung and John Zahorjan *
Department of Computer Science & Engineering
University of Washington

Abstract

When the inter-iteration dependency pattern of the iterations of a loop cannot be determined statically, compile time parallelization of the loop is not possible. In these cases, runtime parallelization [8] is the only alternative. The idea is to transform the loop into two code fragments: the *inspector* and the *executor*. When the program is run, the inspector examines the iteration dependencies and constructs a parallel schedule. The executor subsequently uses that schedule to carry out the actual computation in parallel.

In this paper, we show how to reduce the overhead of running the inspector through its parallel execution. We describe two related approaches. The first, which emphasizes inspector efficiency, achieves nearly linear speedup relative to a sequential execution of the inspector, but produces a schedule that may be less efficient for the executor. The second technique, which emphasizes executor efficiency, does not in general achieve linear speedup of the inspector, but is guaranteed to produce the best achievable schedule. We present these techniques, show that they are correct, and compare their performance to existing techniques using a set of experiments.

Because in this paper we are optimizing inspector time, but leaving the executor unchanged, the techniques we present have most dramatic effect when the inspector must be run for each invocation of the source loop. In a companion paper [3], we explore techniques that build upon those developed here to also improve executor performance.

1 Introduction

Parallelizing compilers offer the parallel machine programmer the best of both worlds: the programming model is nearly identical to that of the familiar sequential programming language, while program execution makes use of the availability of multiple processors, resulting in good parallel performance. To achieve this goal, the compiler relies on a static dependency analysis in order to compute a valid parallel schedule for iteration execution. However, programs with complicated or data dependent array indexing expressions are not amenable to such static analysis, and so cannot be parallelized in this way. For these programs, runtime parallelization is an attractive alternative.

In this section, we introduce the basic mechanisms used to support runtime parallelization, as developed by Saltz and his colleagues [6, 7, 8]. We begin by presenting the basic scheme, and then examine a technique they have proposed for increasing its efficiency. Our improvements to these techniques are presented in subsequent sections.

1.1 The Basic Inspector/Executor Scheme of Saltz et al.

We call the loop being parallelized the *source loop*. A simplified representation of the form of the source

*Support for this work was provided in part by the National Science Foundation (Grants CCR-8619663, CCR-9123308, and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation (Systems Research Center and External Research Program). Authors' addresses: Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195; *shuntak@cs.washington.edu, zahorjan@cs.washington.edu.*

loops on which we focus is shown in Figure 1. In it, each array element $a[i]$ in order is assigned a value that depends on other elements of a. Naturally, the loop body may contain other statements which do not lead to inter-iteration dependencies on a, or may be nested inside other sequential loops. These additional considerations are not shown for simplicity.

```
do i = 1 to n
  a[i] = F(a[g(i)], a[h(i)], ...)
enddo
```

Figure 1: Sequential Program Source Loop

The basic idea of runtime parallelization, as defined by Saltz et al. [6, 7, 8], is for the compiler to produce two pieces of customized code for each source loop . The first, called the *inspector*, calculates the values of the index functions used in each iteration of the loop, and from these values determines the inter-iteration dependencies. From these dependencies the inspector can produce a parallel execution schedule for the iterations. The second piece of code, called the *executor*, uses this schedule to achieve a parallel execution of the source loop.

The parallel schedule computed by the inspector is based on partitioning the set of iterations into subsets, called *wavefronts*. All iterations within the same wavefront can be executed concurrently. The wavefronts themselves are processed sequentially.

A schedule's *depth* is the number of wavefronts it has. There are many valid schedules for the same source loop, some with more wavefronts than others. As a rule of thumb, we prefer a schedule with minimal depth, as such a schedule is likely (but not guaranteed) to lead to minimal parallel execution time.

```
/* wf is initially all zeros */
do i = 1 to n
  wf[i] = max(wf[g(i)],wf[h(i)],...) + 1
enddo
```

Figure 2: Basic Inspector Structure

Figure 2 shows the structure of the basic Saltz et al. inspector, which is responsible for producing a schedule that respects flow (read-after-write) dependencies. If iteration l reads a value produced by iteration k, then because of the restrictions on the form of the source loop, l must be greater than k. Thus, when running the inspector, iteration k must have already been placed in a wavefront by the time iteration l is considered. The inspector looks up k's wavefront from wf and places l in a later wavefront, leading to a schedule that is correct with respect to flow dependencies.

On the other hand, because the inspector disregards both anti- (write-after-read) and output (write-after-write) dependencies, these must be enforced by the executor. The executor shown in Figure 3, first proposed by Saltz et al. [6, 8] and adopted for all work presented here, is sufficient to do this for the class of source loops under consideration. (Improvements to both executor efficiency and domain of applicability are developed in a companion paper [3].)

```
do w = 1 to depth
  pardo all i such that wf[i] = w
    if (g(i) < i) then a1 = anew[g(i)]
                  else a1 = aold[g(i)]
    endif
    ...
    anew[i] = F(a1, a2, ...)
  enddo
enddo
aold = anew
```

Figure 3: Basic Executor Structure

As written in Figure 2, the inspector is sequential. Because it processes as many iterations as the source loop itself, it requires time commensurate with that of a sequential source loop execution, defeating the purpose of runtime parallelization.

Saltz et al. have identified two techniques for reducing the overhead associated with running the inspector. The first is applicable when the source loop is contained inside one or more sequential loops, and when its dependency pattern does not change across iterations of these outer loops. In this case, a single execution of the inspector produces a parallel schedule that can be applied repeatedly, thus amortizing inspector overhead across repeated source loop executions.

The second technique used to reduce inspector overhead is to parallelize its execution. Saltz et al. do this using an alternative general runtime loop parallelization procedure, which they call the preprocessed doacross [7]. We explain this technique briefly in the following subsection.

1.2 Doacross Parallelization

Doacross parallelization [7] is a general runtime parallelization technique that is not in the class of inspector/executor approaches. Doacross is based on the use of an auxiliary array that specifies whether or not each data array element has already been written in the current invocation of the source loop. Processors are assigned iterations in a wrapped manner, and each spin waits until all operands necessary for its execution have been produced.

The advantages of doacross parallelization are that no inspector step is required for loops like the source loop we are considering, reducing overhead, and that a somewhat more general class of loop can be handled by introducing a preprocessing step. The disadvantages of doacross, though, are that it applies to only a restricted class of loop, and that it has potentially significantly suboptimal parallel efficiency. In fact, Saltz et al. find that while the doacross parallelization is preferable to sequential execution [7], it is not a viable alternative to the inspector/executor approach for loops amenable to the latter technique [8]. Thus, we will not consider doacross parallelization further for general runtime parallelization.

However, there is one case in which doacross parallelization can be applied usefully: parallelizing the computation of wavefronts in the inspector. As shown in Figure 2, the inspector is sequential, and so is very expensive. In [8] this expense is reduced by parallelizing the sequential inspector loop using the doacross technique.

There are in fact two variants of the parallel inspector used in [8]. Each employs doacross to parallelize the computation of the wavefronts. They differ, however, in how the iterations in each wavefront are assigned to processors for execution by the executor. The first method, *doacross-global*, appears to be inherently sequential [8], and so is expensive. However, it produces an iteration assignment that is as balanced as possible. In contrast, the second approach, *doacross-local*, is easily parallelized, but makes no guarantees about how balanced the resulting schedule will be. In the worst case, doacross-local can produce a schedule that is sequential in a situation where doacross-global would produce a schedule with perfect speedup.

1.3 Paper Goals and Organization

While the doacross technique does parallelize the inspector to some degree, performance may still be poor [8]. Saltz et al. reports an experiment in which, using sixteen processors, the inspector took 100 ms. while the executor required only 23 ms. In comparison, the source loop took 241 ms. when run sequentially, indicating that the efficiency of the inspector was less than 15%. While parallelizing the source loop in this case does significantly reduce the time to perform the computation (from 241 ms. to 123 ms.), the poor efficiency achieved by the inspector, which accounted for over 80% of the parallel execution time, indicates that there is much room for improvement. Our aim in this paper is to develop alternative inspector parallelization techniques that come as close as possible to the goal of perfect inspector speedup.

In Section 2 we present *sectioning*, a technique for parallelizing the inspector that achieves nearly linear speedup. In Section 3 we present *bootstrapping*, which is not quite as efficient but is guaranteed to produce a minimal depth schedule. In Section 4 we evaluate the performance of these techniques, and compare them to the doacross executions of the inspector. Finally, Section 5 summarizes our conclusions.

2 Parallelizing the Inspector: Sectioning

2.1 The Sectioning Algorithm

One way of parallelizing the inspector is by *sectioning*. The basic idea is simple. The entire range of iterations is partitioned into consecutive sub-ranges, which we call *sections*. Each section is assigned to a different processor. Each processor computes a valid parallel schedule for the iterations in its section by applying the sequential inspector algorithm and ignoring any dependencies on iterations outside of its section.

After all processors have finished, we have a schedule for each section. Every such schedule, which we call a *sub-schedule*, is a mapping from the iterations in the corresponding section to the wavefronts of that section. The overall schedule is formed by concatenating the sub-schedules in the order of the sections. In other words, in the overall schedule, the last wavefront of section s is immediately followed by the first wavefront of section $s + 1$.

Figure 4 illustrates how fifteen iterations are divided into three sections and how the three sub-schedules are concatenated to form the overall schedule. (For simplicity, we have not shown the data dependencies that induce the particular sub-schedules shown.) This schedule is valid but, like all schedules computed by sectioning, it does not necessarily have the fewest possible wavefronts.

It is easy to show that sectioning produces a valid schedule. In particular, a valid schedule must satisfy the following condition: if there is a flow dependence from iteration k to iteration l (remember that the inspector needs to consider only flow dependencies), then l's wavefront is after k's. There are two cases to consider. If iterations k and l belong to different sections, the concatenation of sub-schedules guarantees this condition by putting all the wavefronts of l's section behind those of k's section. If the two iterations are in the same section, they are handled by a single processor running the sequential inspector algorithm. The algorithm ensures that the condition is satisfied.

Given the mapping from iterations to wavefronts, the final step is to assign iterations to processors in a way that respects these wavefronts. This assignment is done in the inspector. Each processor is responsible for scheduling its own section, thus yielding good parallelism. The iterations of each wavefront are assigned in a wrapped manner among the processors, thus giving good executor load balance.

2.2 Comparison with the Doacross Inspector

There are two criteria by which we can compare the different inspectors: execution time and quality of the schedule produced.

In terms of execution time, the key consideration is how efficiently each approach is able to make use of multiple processors. In this respect, both of the doacross-based techniques are at a disadvantage. To begin with, the efficiency of the basic doacross scheme is data dependent. While it is probably fairly good in many cases, there are certain to be a large number of others in which it is quite poor. Additionally, doacross-global has a large, basically sequential component in forming the schedule for the executor from the wavefront information.

In contrast, our parallel inspector should exhibit nearly perfect speedups[1]: each processor works on a subproblem in a way that is completely independent of the others. There is no need for any kind of locking as no shared data structures are modified. Nor is there any need for synchronization, other than a barrier to detect when all the processors have finished.

In terms of the quality of the schedule produced, the situation is less clear. Intuitively, a good schedule is one that has minimum depth and is load balanced. Of the three inspectors we are considering, only doacross-global has this property: doacross-local produces a minimum depth schedule, but it may be very load imbalanced, while sectioning produces a load balanced schedule, but its depth may be somewhat greater (up to a factor of the number of sections) than the minimum possible.

In Section 4, we use experimental results to assess the impact of schedule depth on performance. In the next section, we show how to compute a minimal depth schedule in a highly parallelizable way, but at the cost of a second pass over the iterations.

3 Parallelizing the Inspector: Bootstrapping

When we parallelize the inspector by sectioning, we achieve perfect speedup at the cost of a schedule that is deeper than necessary. In this subsection we

[1]Technically, speedup is an inappropriate term as the output of our inspector, like that of doacross-local, depends on the number of processors assigned to the problem. However, the intuitive notion of speedup captures the running time considerations important here.

section	1	2	3
iterations	1 2 3 4 5	6 7 8 9 10	11 12 13 14 15

(a) Partitioning iterations into sections

section	1		2		3		
wavefront number in sub-schedule	1	2	1	2	1	2	3
wavefront number in overall schedule	1	2	3	4	5	6	7
iterations in wavefront	1 2 5	3 4	6 7 9	8 10	11 14	12	13 15

(b) Concatenating sub-schedules

Figure 4: Example of Parallelizing the Inspector by Sectioning

present a method of parallelizing the inspector that costs more inspector execution time but that guarantees that the schedule generated is identical to the minimal depth schedule produced by the sequential inspector. We call this method *bootstrapping*.

Bootstrapping is based on the observation that the sequential inspector (Figure 2) is itself a loop, and that it has the same inter-iteration dependency pattern as the source loop (Figure 1). Comparing the two loops, we see that the inspector loop is simply the source loop with the general function $F(.)$ replaced by $max(.) + 1$ and a by wf. It therefore follows that any parallel schedule valid for the source loop is also valid for the inspector loop. This observation forms the basis of the boostrapping technique.

To bootstrap, we first generate a valid source loop schedule using sectioning. Then we use this schedule to run the sequential inspector loop in a wavefront-by-wavefront (i.e., parallel) manner. We call this second step *refinement*. The final schedule thus computed is the same as what a sequential inspector would compute, for the same reason that the executor yields the same results as the sequential source loop. Thus, bootstrapping must produce the same minimum depth schedule as the sequential inspector.

Having computed the minimum depth schedule, the inspector must assign iterations to processors for execution in the executor. This is done in two steps. First, when a processor running the inspector assigns an iteration to a wavefront, it tentatively assigns execution of that iteration to itself. The schedule resulting from this procedure could be imbalanced. Therefore, in the second step, overloaded processors reassign iterations to underloaded ones. This too is done in parallel.

4 Performance Comparison

In this section, we compare the speeds of the various inspector algorithms and their performance impacts on the executor. We use successive over-relaxation as a concrete example.

4.1 Successive Over-Relaxation

Successive over-relaxation (SOR) [1, 5] is a numerical technique that can be used to obtain the solution of a sparse linear system. Given a non-singular $n \times n$ matrix A and an n-dimensional vector b, we want to solve the system of linear equations $Ax = b$ for the unknown n-dimensional vector x. The SOR algorithm starts with an initial guess of the solution vector, x^0, and repeatedly computes a new estimate x^{t+1} from the previous estimate x^t using the formula:

$$x_i^{t+1} = (1-\gamma)x_i^t - \frac{\gamma}{A_{ii}}\left(\sum_{j<i} A_{ij}x_j^{t+1} + \sum_{j>i} A_{ij}x_j^t - b_i\right)$$

where the subscripts represent indexing and where γ is a manually chosen non-zero scalar called the relaxation parameter. The algorithm terminates when the estimates converge or after some predetermined number of iterations have been performed.

Figure 5 shows a sequential implementation of SOR when the matrix A is sparse. (For concreteness, it is shown in actual C code rather than pseudo-code.) Because A is sparse, only non-zero coefficients are stored: $nzCoeffs$ holds the coefficients used by the algorithm (*not* elements of A), while $nzColumns$ records the columns to which they belong. Element $firstNz[i]$ indicates where the information for row i starts in $nzCoeffs$ and $nzColumns$.

For this program, the outer loop is our source loop. Each iteration reads and writes elements of x. Only iteration i writes $x[i]$ and hence there are no inter-iteration output dependencies. The flow dependencies and anti-dependencies cannot be determined at compile time because the contents of $nzColumns$ depend on input data and are not known until runtime. However, the dependencies do not depend on computation performed within the source loop, and so runtime parallelization is possible.

In our experiments, the source loop was manually transformed into its corresponding inspector and executor because we did not have compiler support for this conversion.

4.2 Results

We applied the SOR technique to solve the global balance equations [2] corresponding to a queueing network model with blocking [4]. The model itself consists of a number of service centers in series, each with a finite capacity queue. This sort of model is commonly used to evaluate the performance of communication networks built from switches with finite buffer space.

Measurements were taken on a Sequent Symmetry shared memory multiprocessor running Mach 3.0. All programs were written in C using Cthreads and compiled with the *gcc* compiler. In each case studied, we made several measurements and observed only insignificant variations among the measured times.

We tried a number of different parameterizations of the queueing model, leading to different sets of linear equations to be solved. We present two examples here. Problem A has a 17577×17577 coefficient matrix with about 100000 non-zero elements. A sequential execution of a single iteration of SOR for this problem takes 1.58 seconds. Problem B has a 45676×45676 matrix with about 250000 non-zero elements. A sequential execution of a single iteration of SOR for this problem takes 4.03 seconds.

Figure 6 shows the achieved inspector and executor efficiencies for the two problems. For the inspector, efficiency is measured against the sequential inspector of Figure 2. For the executor, efficiency is measured against a true sequential execution of the loop.

Considering the inspectors first, we see that only the sectioning approach achieves good efficiency[2]. For the doacross inspectors, efficiency drops off rapidly with number of processors. For bootstrapping, efficiency is more nearly constant, but is half that of sectioning, reflecting the two pass nature of this technique.

Examining executor efficiency, we note first they are uniformly much lower than would be desired (although they still represent a considerable improvement over the sequential execution that would be the only option if compile time parallelization were employed). This is due in part to the overheads involved in the executor (Figure 3) relative to the sequential loop, as evidenced by the poor executor efficiency when running on a single processor. Because our focus in this paper is on improving the inspector phase, we have not attempted to address these executor inefficiencies here, but leave them for a companion paper [3].

Since we ran identical executor code with each inspector, any differences observed in their efficiencies must stem from differences in the schedules the inspectors produce. We make two observations in this regard. First, the doacross-local technique distinguishes itself with slightly lower efficiency than the others. This is a reflection of that policy's imbalanced assignment of iterations to processors within each wavefront.

[2]Because the various inspectors in fact compute different results than the sequential inspector, the term "efficiency" does not strictly apply. However, the efficiencies as we have computed them do convey the proper sense of reduction in elapsed time due to parallelization.

```
for (i = 0; i < n; i++) {
  sum = constant[i];
  for (nz = firstNz[i]; nz < firstNz[i+1]; nz++) {
    sum += nzCoeffs[nz] * x[nzColumns[nz]];
  }
  x[i] = sum;
}
```

Figure 5: Sequential Implementation of SOR for Sparse Linear Systems

Second, we note that, perhaps surprisingly, the increase in schedule depth caused by sectioning does not necessarily have a great impact on performance. Table 1 shows the number of wavefronts produced by the sectioning inspector. The minimum possible depth schedule (which is produced by the other three inspectors) corresponds to the sectioning result when run on a single processor. Examining Figure 6, we observe that a doubling of the schedule depth has little or no effect on executor performance.

Number of	Number of Wavefronts	
Processors	Problem A	Problem B
1	9	12
2	14	18
4	15	18
8	16	20
16	21	24

Table 1: Number of Wavefronts in Schedule Produced By Sectioning

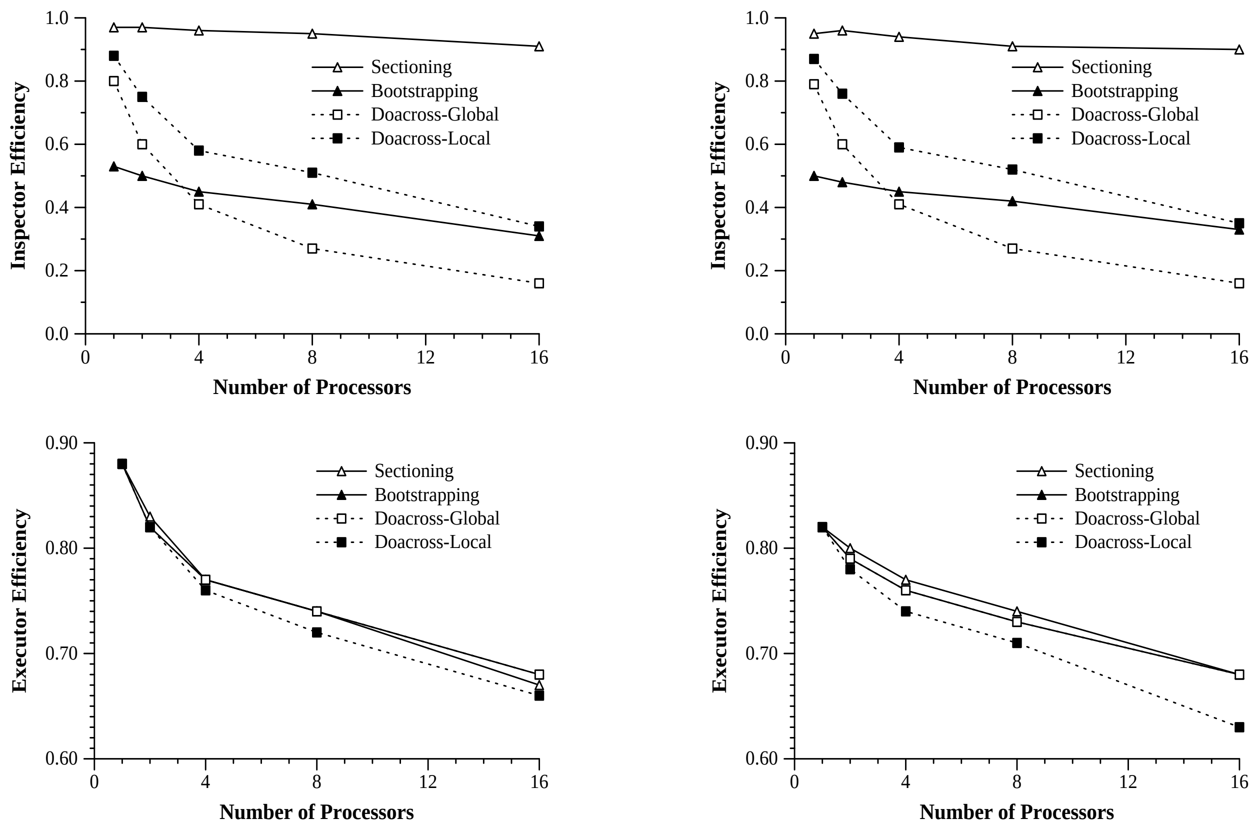
A look at the parallel schedules tells us why sectioning does not necessarily slow down the executor. Take problem B (the larger problem) as an example. The sequential inspector generates a schedule with 12 wavefronts, giving an average of about 3800 iterations per wavefront. In contrast, the schedule computed using 16 sections has 24 wavefronts, for an average of 1900 iterations each. Although the schedule is deepened by a factor of two, there is more than enough work in each wavefront to keep all 16 processors busy. Thus, the difference in schedule depth does not translate into a difference in execution time. Clearly, if the minimal depth schedule had been much deeper because of the dependencies, doubling the schedule depth could have meant under-utilized processors and added synchronization overhead, leading to degraded executor performance.

As a final test of the effect that a deeper schedule obtained by sectioning has on performance, we examined a single queueing model and varied the number of customers in it, producing a set of related systems of linear equations of varying sizes. In Figure 7 we plot the number of invocations of the executor required for sectioning and bootstrapping to have identical total execution times against the average number of iterations per processor per sectioning wavefront. If SOR takes more than the plotted number of iterations to converge, the greater efficiency of the executor under the bootstrapping schedule more than compensates for the added expense of the refinement step required to produce that schedule; below the line, sectioning is cheaper overall.

We see from Figure 7 that when the average number of iterations executed by each processor per sectioning wavefront is small, bootstrapping is advantageous even for very small numbers of invocations of the executor. However, this breakeven point grows quite rapidly, and in general it appears that sectioning is the method of choice for loops where the number of iterations per wavefront greatly exceeds the number of processors.

4.3 Combining Bootstrapping with Sectioning

The relationship of schedule depth to executor execution time motivates the following heuristic technique. We first generate a schedule by sectioning. Given this schedule, we can easily determine how many iterations there are on average in each wavefront. If this number is *not* much greater than the



Model A (17577 × 17577) Model B (45676 × 45676)

Figure 6: Observed Inspector and Executor Efficiencies

number of processors that the executor can use, it is likely that the processors will be under-utilized. In this case, the refinement step is performed to obtain a schedule with a minimum number of wavefronts. Otherwise, there is no need to do so. How many times the executor will be run can also be taken into account, if it is known or can be reasonably estimated: the larger the estimated number of executor invocations, the greater the tendency to choose boostrapping over sectioning.

5 Conclusions

Saltz et al. [6, 8, 7] have made important contributions to the area of runtime parallelization. Our work builds on theirs. The purpose of the work described here is to improve the efficiency of the inspector phase of runtime parallelization through its parallel execution.

We have discussed two approaches to achieving this: sectioning and bootstrapping. Sectioning is a divide-and-conquer strategy that has nearly perfect speedup. On the other hand, the schedule it produces may contain more wavefronts than necessary. While it is unlikely that this would result in a noticeable degradation in executor efficiency unless the average number of iterations per wavefront per processor is quite low, this cannot be guaranteed.

Our second technique, bootstrapping, addresses this potential drawback of sectioning, trading some inspector time for a schedule that is guaranteed to have minimal depth. Bootstrapping is based on the observation that the sequential inspector loop itself can be parallelized using the same schedule as

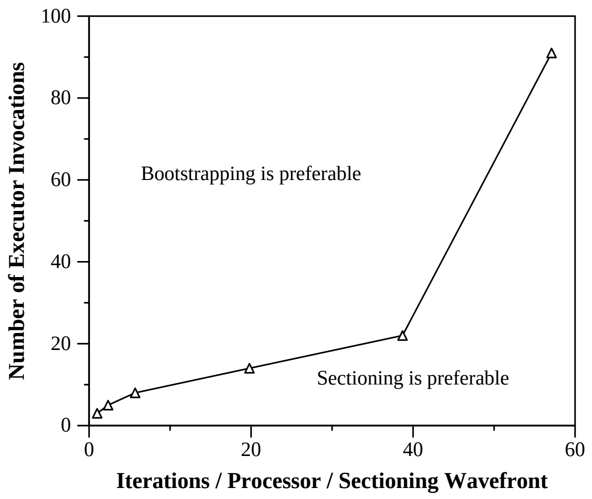


Figure 7: Number of Executor Invocations Required for Equal Sectioning and Boostrapping Total Execution Time

the source loop. Thus, we first compute a schedule by sectioning, producing a potentially suboptimal parallel schedule, and then use this schedule to execute the parallelized version of the sequential inspector, which is known to produce a minimal depth schedule.

We performed a number of experiments to compare these methods. We saw that our new techniques can significantly reduce the overhead of running the inspector. We noted that while sectioning can increase the depth of the schedule, that this has little impact on executor performance if there are enough iterations per wavefront to keep all processors busy. Finally, we noted that in cases where the number of iterations per wavefront is modest relative to the number of processors, and the executor will be executed many times with the same schedule, that the additional cost of bootstrapping can be paid for by a reduction in executor times.

The techniques presented in this paper and the old techniques represent an array of options from which the compiler can choose when it generates code for the inspector. Which one is best in terms of overall execution time (which is the ultimate bottom line) depends on the size and dependency pattern of the source loop, as well as the number of times it is executed. We believe that in many situations, sectioning and bootstrapping are appropriate choices.

Acknowledgements

Derek Eager, Ed Felten, Ed Lazowska, and Radika Thekkath provided valuable comments on the content and presentation of this material.

References

[1] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[2] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance.* Prentice Hall, Englewood Cliffs, NJ, 1984.

[3] S. Leung and J. Zahorjan. Extending the domain and improving the execution performance of runtime parallelization. Technical Report , in preparation, Department of Computer Science & Engineering, University of Washington, October 1992.

[4] Raif O. Onvural. Survey of closed queueing networks with blocking. *ACM Computing Surveys*, 22(2):83–121, June 1990.

[5] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1986.

[6] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. In *Proc. International Workshop on Compilers for Parallel Computers*, Paris, 1990.

[7] J. Saltz and R. Mirchandaney. The preprocessed doacross loop. In *Proc. 1991 International Conference on Parallel Processing*, August 1991.

[8] J. Saltz, R. Mirchandaney, and K. Crowley. Runtime parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.