ASYNC Loop Constructs for Relaxed Synchronization

Russell Meyers and Zhiyuan Li

Department of Computer Science
Purdue University, West Lafayette IN 47906, USA,
{rmeyers,li}@cs.purdue.edu

Abstract. Conventional iterative solvers for partial differential equations impose strict data dependencies between each solution point and its neighbors. When implemented in OpenMP, they repeatedly execute barrier synchronization in each iterative step to ensure that data dependencies are strictly satisfied. We propose new parallel annotations to support an *asynchronous computation model* for iterative solvers. ASYNC_DO annotates a loop whose iterations can be executed by multiple processors, as OpenMP parallel `DO` loops in Fortran (or parallel `for` loops in C), but it does not require barrier synchronization. ASYNC_REDUCTION annotates a loop which performs parallel reduction operations but uses a relaxed tree barrier, instead of the conventional barrier, to synchronize the processors. When a number of ASYNC_DO and ASYNC_REDUCTION loops are embedded in an iterative loop annotated by ASYNC_REGION, the iterative solver allows each data point to be updated using the value of its neighbors which may not be the most current, instead of forcing the processor to wait for the new value to arrive. We discuss how the compiler can transform an ASYNC_REGION (with embedded ASYNC_DO and ASYNC_REDUCTION) into an OpenMP parallel section with relaxed synchronization. We present experimental results to show the benefit of using ASYNC loop constructs in 2D and 3D multigrid methods as well as an SOR-preconditioned conjugate gradient linear system solver.

1 Introduction

Many important applications use iterative solvers to solve partial differential equations (PDE's). It has been found for quite some time that there exist a class of iterative solvers which are allowed to follow a loose data dependence relationship between each data point and its neighbors [9, 3, 4]. Under such an *asynchronous computation model*, the update of a data point does not need to strictly depend on the most updated values of its neighbors. Instead, Some older values of its neighbors can be used before the newest values become available. It may take more iterations for an asynchronous algorithm to converge or to achieve the same numerical accuracy as its synchronous counterpart. However, when implemented on parallel systems, especially those of a large size, the asynchronous algorithms suffer less from the interconnect latency than their conventional counterparts.

Unfortunately, current parallel languages and language extensions (such as OpenMP [7]) do not effectively support the asynchronous computation model. When implemented with OpenMP parallel annotations, for example, an iterative solver typically has a sequential outermost loop containing a number of parallel inner loops. Each parallel loop annotation implies two barrier synchronization points, at the beginning and the end of the loop, where all processors must meet before simultaneously proceeding to the next statement. Such barrier synchronization, executed repeatedly in each iterative step, dictates the conventional strict synchronous computation model, at the expense of performance penalty due to the interconnect latency. Barriers also severely limit the compiler's ability to generate efficient machine code.

In this paper, we propose three new loop annotations, called *ASYNC_DO*, *ASYNC_REDUCTION*, and *ASYNC_REGION*, respectively. ASYNC_DO annotates a loop whose iterations can be executed by multiple processors, but unlike an OpenMP parallel DO (or parallel for) loop, it does not require barrier synchronization. ASYNC_REDUCTION annotates a loop which performs parallel reduction but uses a relaxed tree barrier, instead of the conventional barrier, to synchronize the processors. When a number of ASYNC_DO and ASYNC_REDUCTION loops are embedded in an iterative loop annotated by ASYNC_REGION, the iterative solver in effect follows the *asynchronous computation model* mentioned above. We present experimental results to show the benefit of using ASYNC loop constructs in 2D and 3D multigrid methods as well as an SOR-preconditioned conjugate gradient linear system solver.

The rest of this paper is organized as follows. In section 2 we present the concept and semantics of the proposed ASYNC loop constructs, and we discuss how the compiler can transform ASYNC_REGION (with embedded ASYNC_DO and ASYNC_REDUCTION loops) into an OpenMP parallel section with relaxed synchronization that supports the asynchronous computation model. In section 3, we introduce the benchmarks used in our experiments and discuss their numerical background. We present experimental results in section 4, which is followed by a review of related work (Section 5) and a conclusion (Section 6).

2 ASYNC Loop Constructs

ASYNC_DO Loops ASYNC_DO annotates a DO loop whose iterations can be executed in parallel by multiple processors without barrier synchronization. Its syntax, analogous to that of an OpenMP parallel DO loop, is in the form of `!$ASYNC_DO` *parallel clause*, where *parallel clause* takes the same form and meaning as its counterpart in OpenMP *sans* the reduction clause [7]. Figure 1 shows an example of annotating parallel loops by ASYNC_DO. When the shown ASYNC_DO is embedded in an ASYNC_REGION, it will be transformed by the compiler into an iteration-partitioned loop shown in Figure 2. Notice the absence of barrier synchronization.

```
!$ASYNC_DO default(shared)
!$         private(i1,i2,i3,u1,u2)
      do i3=2,n3-1
        do i2=2,n2-1
          do i1=1,n1
            u1(i1) = u(i1,i2-1,i3)
     >          + u(i1,i2+1,i3)
     >          + u(i1,i2,i3-1)
     >          + u(i1,i2,i3+1)
            u2(i1) = u(i1,i2-1,i3-1)
     >          + u(i1,i2+1,i3-1)
     >          + u(i1,i2-1,i3+1)
     >          + u(i1,i2+1,i3+1)
          enddo
          do i1=2,n1-1
            r(i1,i2,i3) = v(i1,i2,i3)
     >           - a(0) * u(i1,i2,i3)
     >           - a(1) * (u(i1-1,i2,i3)
     >                  + u(i1+1,i2,i3)
     >                  + u1(i1))
     >           - a(2) * (u2(i1)
     >                   + u1(i1-1)
     >                   + u1(i1+1))
     >           - a(3) * (u2(i1-1)
     >                   + u2(i1+1))
          enddo
        enddo
      enddo
```

Fig. 1. An ASYNC_DO loop inside MG residual calculation subroutine

```
      z_low = (my_id * bz(k)) + 1
      z_high = (my_id + 1) * bz(k)
      if(my_id .eq. 0) z_low = 2
      if(my_id .eq. (total_threads - 1))
         z_high = n3 - 1
      do i3 = z_low, z_high
        do i2=2,n2-1
          do i1=1,n1
            u1(i1) = u(i1,i2-1,i3)
     >          + u(i1,i2+1,i3)
     >          + u(i1,i2,i3-1)
     >          + u(i1,i2,i3+1)
            u2(i1) = u(i1,i2-1,i3-1)
     >          + u(i1,i2+1,i3-1)
     >          + u(i1,i2-1,i3+1)
     >          + u(i1,i2+1,i3+1)
          enddo
          do i1=2,n1-1
            r(i1,i2,i3) = v(i1,i2,i3)
     >           - a(0) * u(i1,i2,i3)
     >           - a(1) * (u(i1-1,i2,i3)
     >                   + u(i1+1,i2,i3)
     >                   + u1(i1))
     >           - a(2) * (u2(i1)
     >                   + u1(i1-1)
     >                   + u1(i1+1))
     >           - a(3) * (u2(i1-1)
     >                   + u2(i1+1))
          enddo
        enddo
      enddo
```

Fig. 2. A synchronization-relaxed loop generated from ASYNC_DO annotation

ASYNC_REDUCTION The ASYNC_REDUCTION is supported by a relaxed barrier tree structure which allows a thread, depending on its thread ID, to deposit its partial term of the reduction result in the tree and continue its execution without obtaining the newly computed value. Figure 3 shows an example with eight threads. Threads are numbered 0 through 7 and every dot represents a lock structure. Once a pair of threads arrive at the appropriate lock (if one gets there first, it waits for the other), the left-sibling thread proceeds up the tree with the new information deposited by both threads, and the other thread is released and allowed to continue execution. Using this structure, we can greatly reduce the amount of blocking time of threads.

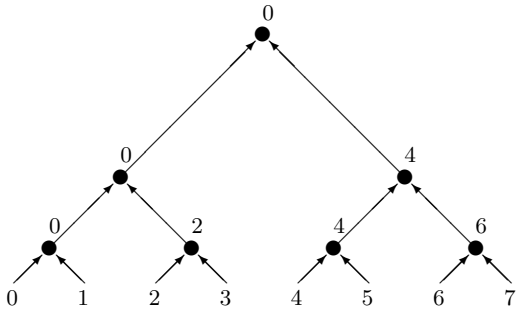

Fig. 3. A relaxed barrier tree structure

The ASYNC_REDUCTION annotation is analogous to an OpenMP parallel DO loop with a reduction clause, but with the relaxed barrier tree replacing the strict barrier. Figure 4 shows an example of a loop annotated by ASYNC_REDUCTION. When embedded in an ASYNC_REGION, this loop will be transformed into an iteration-partitioned loop with a call to a runtime routine `logbarrier` which implements the relaxed barrier tree, as shown in Figure 5.

```
!$ASYNC_REDUCTION(+:d)
      do j=1, lastcol-firstcol+1
        d = d + p(j)*q(j)
      enddo
```

Fig. 4. An ASYNC_REDUCTION loop

```
temp = 0.d0
do j = low_limit, high_limit
  temp = temp + p(j)*q(j)
enddo
call logbarrier(my_id, temp, d, 0)
```

Fig. 5. A loop converted from ASYNC_REDUCTION

ASYNC_REGION The following code skeleton shows a general template for implementing an iterative solver with an asynchronous region. In this template, the notation !$*a parallel loop header* could mean !$ASYNC_DO, !$ASYNC_REDUCTION, or any conventional OpenMP parallel construct.

Algorithm 1 A General Template of Using ASYNC Loop Constructs

```
1: !$ASYNC_REGION
2: DO ITER = 1, number_iter
3: !$a parallel loop header
4: a parallel loop body
5: ...
6: !$a parallel loop header
7: a parallel loop body
8: ...
9: !$a parallel loop header
10: a parallel loop body
11: ...
12: END DO ITER
```

The iterative loop annotated by ASYNC_REGION will be transformed by the compiler into an OpenMP parallel section (!$OMP parallel) to be executed by a number of parallel threads, each executing the identical code but may access and compute different sections of the same arrays depending on their thread IDs. Within the asynchronous region, an ASYNC_DO loop is transformed by the compiler into a DO loop whose iteration ranges are determined by the thread ID, as illustrated previously in Figure 2. No barrier synchronization is inserted. An ASYNC_REDUCTION loop is transformed into a DO loop that computes a partial reduction before invoking the relaxed barrier synchronization routine to add the partial term to the final result, as illustrated in Figure 5. A conventional OpenMP parallel do or reduction loop will be transformed into a DO loop as defined by the OpenMP standard, for which conventional barrier synchronization is inserted. Sequential loops and statements will be enclosed in a segment annotated by `!$OMP master` to make sure they are executed by the master thread only.

It is important for the compiler to *align* the iteration ranges of the DO loops converted from the inner parallel loops (ASYNC or OpenMP) such that no two threads write into the same array sections. This requires the compiler to perform array data flow analysis which has been studied quite extensively by previous work [12, 15, 16] and will be omitted in this paper due to space limitation. If the compiler is unable to perform the necessary array dataflow analysis for a particular asynchronous region, the ASYNC_REGION annotation will be ignored. An ASYNC_DO loop will be treated as an ordinary OpenMP parallel DO loop and an ASYNC_REDUCTION loop will be treated as an ordinary OpenMP reduction loop.

As one can see, transformation of an asynchronous region into an OpenMP parallel section with relaxed synchronization is a rather straightforward task for the compiler (except for the array dataflow analysis whose difficulty depends on the complexity of the symbolic expressions in the loop body). On the other hand, the ASYNC annotation makes it significantly easier for the programmer to experiment with relaxed synchronization in an iterative solver (whose iterative

loop body may be long and may contain many parallel loops). We will show a concrete example below when we present the benchmarks for this study.

3 Benchmark Study

In this paper, we are interested in using ASYNC_DO loops in two primary classes of iterative methods. The first are *relaxation methods*, which are typically used to solve discretized PDE's [18]. Previous studies have found this class of iterative methods to be particularly suitable for applying the asynchronous computation model. The second class of iterative methods are *projection methods*, of which *Krylov subspace methods* are a subset [18]. These can be used to solve large, sparse linear systems in matrix form, and they rely on the orthogonality of the (Krylov) subspace vectors that are generated during the algorithm for convergence. It is difficult to apply the asynchronous computation model to this class of iterative methods due to the need for frequent synchronization to maintain the orthogonality. Nonetheless, the Krylov subspace methods are known to benefit greatly from *preconditioning*, in terms of both numerical accuracy and computation efficiency. Relaxation methods such as SOR have been found to be effective preconditioners, and they, on the other hand, are good candidates for using the relaxed data propagation model.

In this section, we introduce two benchmarks, namely MG and preconditioned CG, as representatives of these two classes of iterative methods mentioned above. We discuss the numerical background for these benchmarks first and then consider how the asynchronous computation model can be applied to relax the data dependencies. Both MG and CG are from the NAS OpenMP parallel benchmarks [1] (version 3.2) which were released in 2003 [13]. We wrote a simple SOR preconditioner for the CG.

3.1 Multigrid Methods and the MG benchmark

As a representative from the class of relaxation methods, we discuss multigrid methods for solving systems arising from discretized PDE's. In particular, the V-cycle mutigrid method will be examined in this paper. The general purpose for using such methods over a Krylov subspace method is that their convergence rates are, in theory, independent of the mesh size (and hence the linear system size).

In a general algorithm for a multigrid V-cycle [18], the grid is coarsened until a sufficiently coarse grid is reached such that the resulting system can be solved directly. Such coarsening is called *restriction*. After this point, the grids are interpolated successively to the immediately finer grid, and smoothed. This movement to a finer grid is called *prolongation*. The smoothing is done by a preconditioner, such as an approximate inverse preconditioner. Depending on the boundary conditions, special care may need to be taken in communicating the boundary values across the grid if necessary, as with periodic boundary conditions, for example. For better accuracy, the above algorithm can be restarted

multiple times successively on the same mesh until a desired residual precision is met.

MG Benchmark The MG program from the NAS OpenMP benchmarks [13] implements a multigrid method to solve the Poisson problem $\nabla u^2 = v$ with periodic boundary conditions. The benchmark places -1 and +1 values at twenty random grid points each, and zeros elsewhere. Each iteration consists of a full V-cycle as described earlier. We also derived a two-dimensional version of MG for a 2D grid size, but the algorithm remained the same. The high level organization of MG in OpenMP can be illustrated by the code skeleton in Algorithm2. In the actual OpenMP code, the `!$OMP` annotations are within the subroutines shown in the algorithm such that within each subroutine call, one or more !$OMP parallel DO loops are executed, forcing a strict data flow.

Algorithm 2 Multigrid V-Cycle with Full Synchronization

1: DO iter = 1, number_iter
2: **for** $i = h_{max}...h_0, i = i/2$ **do**
3: !$OMP parallel do
4: Coarsen residual: $r^i = I^i_{i/2} r^{i/2}$
5: **end for**
6: !$OMP parallel do
7: Zero: $u^{h_0} = 0$
8: !$OMP parallel do
9: Smooth: $u^{h_0} = u^{h_0} + Sr^{h_0}$
10: **for** $i = 2...h_{max}/2, i = 2i$ **do**
11: !$OMP parallel do
12: Zero: $u^i = 0$
13: !$OMP parallel do
14: Prolongate: $u^i = I^{i/2}_i u^{i/2}$
15: !$OMP parallel do
16: Calculate Residual: $r^i = r^i - Au_i$
17: !$OMP parallel do
18: Smooth: $u^i = u^i + Sr^i$
19: **end for**
20: !$OMP parallel do
21: Prolongate: $u^{h_{max}} = I^{h_{max}/2}_{h_{max}} u^{h_{max}/2}$
22: !$OMP parallel do
23: Calculate Residual: $r^{h_{max}} = r^{h_{max}} - Au^{h_{max}}$
24: !$OMP parallel do
25: Smooth: $u^{h_{max}} = u^{h_{max}} + Sr^{h_{max}}$
26: END DO ITER

MG Using ASYNC_DO To relax the data flow constraints, we annotate the entire iterative solver by ASYNC_REGION (which will then be converted

to an OpenMP parallel section as discussed previously). A careful analysis of the numerical property of the solver suggests that all but two of the synchronization points can be safely removed without severely slowing the convergence. The necessary synchronization points occur immediately after the prolongation operation, but before the residual calculation. Hence, we change all embedded OpenMP parallel DO loops to ASYNC_DO loops. As discussed previously, the ASYNC_DO loops will result in a partition of the parallel loop iterations, but without implied barrier synchronization. We reinsert barriers immediately before the residual calculation. Figures 1 and 2 (in Section 2) show details for the residual calculation loop. The high-level organization of the ASYNC_REGION is shown in the code skeleton in Algorithm 3 below.

Algorithm 3 Multigrid V-Cycle with ASYNC_DO loops

1: !$ASYNC_REGION
2: DO iter = 1, number_iter
3: **for** $i = h_{max}...h_0, i = i/2$ **do**
4: !$ASYNC_DO
5: Coarsen residual: $r^i = I^i_{i/2} r^{i/2}$
6: **end for**
7: !$ASYNC_DO
8: Zero: $u^{h_0} = 0$
9: !$ASYNC_DO
10: Smooth: $u^{h_0} = u^{h_0} + Sr^{h_0}$
11: **for** $i = 2...h_{max}/2, i = 2i$ **do**
12: !$ASYNC_DO
13: Zero: $u^i = 0$
14: !$ASYNC_DO
15: Prolongate: $u^i = I_i^{i/2} u^{i/2}$
16: !$Barrier
17: !$ASYNC_DO
18: Calculate Residual: $r^i = r^i - Au_i$
19: !$ASYNC_DO
20: Smooth: $u^i = u^i + Sr^i$
21: **end for**
22: !$ASYNC_DO
23: Prolongate: $u^{h_{max}} = I_{h_{max}}^{h_{max}/2} u^{h_{max}/2}$
24: !$Barrier
25: !$ASYNC_DO
26: Calculate Residual: $r^{h_{max}} = r^{h_{max}} - Au^{h_{max}}$
27: !$ASYNC_DO
28: Smooth: $u^{h_{max}} = u^{h_{max}} + Sr^{h_{max}}$
29: END DO ITER

The reinserted synchronization points are necessary because, if stale values are used from the interpolated grid, the residual will undoubtedly be higher. Once this higher residual is applied to correct the grid, the difference (in norm)

of the current grid to the previous one will also be larger. This error will propagate through each V-cycle iteration, so that the residual will continue to grow indefinitely after a few iterations. The smoothing operation exists to even out sharp differences in the residual, but the effects of a larger magnitude of residual values in general will still exist. Our experiments without the reinserted barriers have turned out poor numerical accuracies and hence confirmed the necessity of these synchronization points.

3.2 SOR-preconditioned Conjugate Gradient Methods

Krylov subspace methods are designed as general purpose methods for solving large and sparse linear systems, and they are based on projection processes. Among these, the conjugate gradient (CG) method is one of the best known for solving sparse symmetric positive definite linear systems. We use CG in this paper as a representative of Krylov subspace methods.

A *preconditioned* conjugate gradient method is given in Algorithm 4 below [18]. Preconditioning a system of linear equations is a way to transform the original system into one that is likely to be easier to solve with an iterative solver. Both the efficiency and robustness of iterative techniques can be improved by a good preconditioner. The number of iterations to execute CG are expected to be reduced after preconditioning. The preconditoner M (in steps 1 and 6) is chosen such that M^{-1} is a good approximation of A^{-1}. Also, the system $Mz = r$ needs to be much easier to solve than the original system $Ax = b$, for example, using Jacobi, Gauss-Seidel, or Successive Overrelaxation. Here, we apply the preconditioner M to the system $Ax = b$ from the left, i.e. $M^{-1}Ax = M^{-1}b$. Notice in statement 6 of algorithm 4, we compute $z = M^{-1}r = M^{-1}(b - Ax) = M^{-1}b - M^{-1}Ax$. Thus z represents the residual of the transformed system.

Algorithm 4 Preconditioned Conjugate Gradient

1: Compute $r_0 = b - Ax_0$, $z_0 = M^{-1}r_0$, $p_0 = z_0$
2: **for** $j = 0, 1, ...$ until convergence **do**
3: $\quad \alpha_j = (r_j, z_j)/(Ap_j, p_j)$
4: $\quad x_{j+1} = x_j + \alpha_j p_j$
5: $\quad r_{j+1} = r_j - \alpha_j Ap_j$
6: $\quad z_{j+1} = M^{-1}r_{j+1}$
7: $\quad \beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$
8: $\quad p_{j+1} = z_{j+1} + \beta_j p_j$
9: **end for**

CG Benchmark The CG benchmark from NAS estimates the smallest eigenvalue in magnitude of a matrix A using the inverse power method with shifts. During each iteration, a solution of a system of the form $Ax = b$ is obtained by calling a CG subroutine. Because here the CG method is being used as a

solver inside of another iterative method, it is invoked a fixed number of times. The input matrix is of dimension 14000, and the conjugate gradient method is stopped when the residual norm falls below 10^{-8}. We implemented a point successive overrelaxation (SOR) scheme to serve as a preconditioner for CG. Because the structure of the input matrix is symmetric positive definite but random, the point SOR solver is an appropriate preconditioner. The preconditioner is stopped when the residual norm is less than 10^{-6}.

4 Experimental Results

We performed experiments by running the chosen benchmarks on a Sun E10000 which has 54 Ultrasparc II processors (each clocked at 400 Megahertz) and 56 GB of memory. For each data set, tests were run on a differing number of processors, ranging from 1 to 32 in powers of two. With the exception of the residual vs. the number of iterations, all performance data reported here is an average over 10 runs. As we are in an early stage of exploring the potential of ASYNC loop constructs, the conversion from such constructs to OpenMP codes is not yet implemented in a compiler, but is performed manually instead.

3D Multigrid We first use a 256 x 256 x 256 grid size with 4 iterations for the MG benchmark, in order to be consistent with the original benchmark specification for class A problems. Figures 6(a)-6(c) compare the performance and residual after executing four iterations, while figure 6(d) shows that both versions approach convergence at the same rate in the number of iterations shown.

After just four iterations, a satisfactory residual has been reached. We see that the speedup behavior of the relaxed version is superior to that of the original OpenMP version, in particular with more than eight threads. In addition, although the final residual values are slightly higher than that of the original version, the residuals for the relaxed version are approximately of the same order of magnitude in comparison.

Next, we reduce the grid size to see what would result from using the relaxed benchmark on a smaller data set. Figure 2 shows the results from using a grid size of 128 x 256 x 256, and also with 4 iterations used in figures 7(a)-7(c). In figure 7(d), more iterations are given to show that when trying to reach a more satisfactory residual, both versions approach convergence at the same rate.

Clearly the time taken for this grid is half as much as the previous case, and yet the speedup behavior is approximately the same. We also note that the residual for the relaxed version again remains within an order of magnitude of the original version. Given our assumption at the beginning of this section, the relaxed version converges in exactly the same number of iterations as the original for either sized grid.

2D Multigrid To further study the performance, we use a 512 x 512 grid size for the two dimensional problem. Figures 8(a)-8(c) compare the performance and

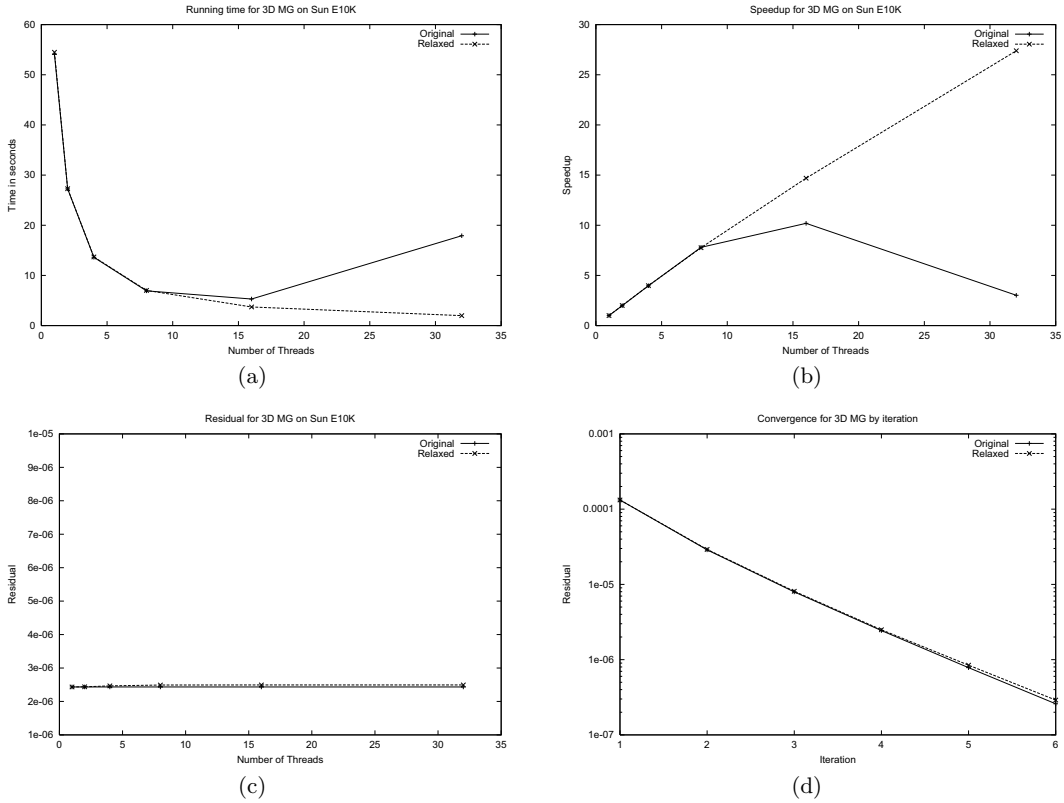

Fig. 6. Comparison between strict and relaxed data flow models with MG. (a) Running time using standard 3D grid (b) Parallel speedups (c) Final residual norms (d) Convergence Rates

residual after executing 30 iterations, and figure 8(d) shows that both versions approach convergence at the same rate in the number of iterations given. The results also show that the original version performs quite poorly in terms of parallel speedup. The relaxed version clearly performs better.

SOR-preconditioned CG We tested the use of SOR as a preconditioner embedded in the basic conjugate gradient method (see Algorithm 4). The SOR preconditioner was run both in its original OpenMP version with strict data flow and in the ASYNC_DO version with relaxed data flow. Figures 9(a) and 9(b) show that the parallel speedups of the relaxed version are much improved over the original version. It is not yet clear why the original version experiences a jump in the running time for four threads. One might observe that the relaxed data flow causes the preconditioner to converge in a greater number of iterations,

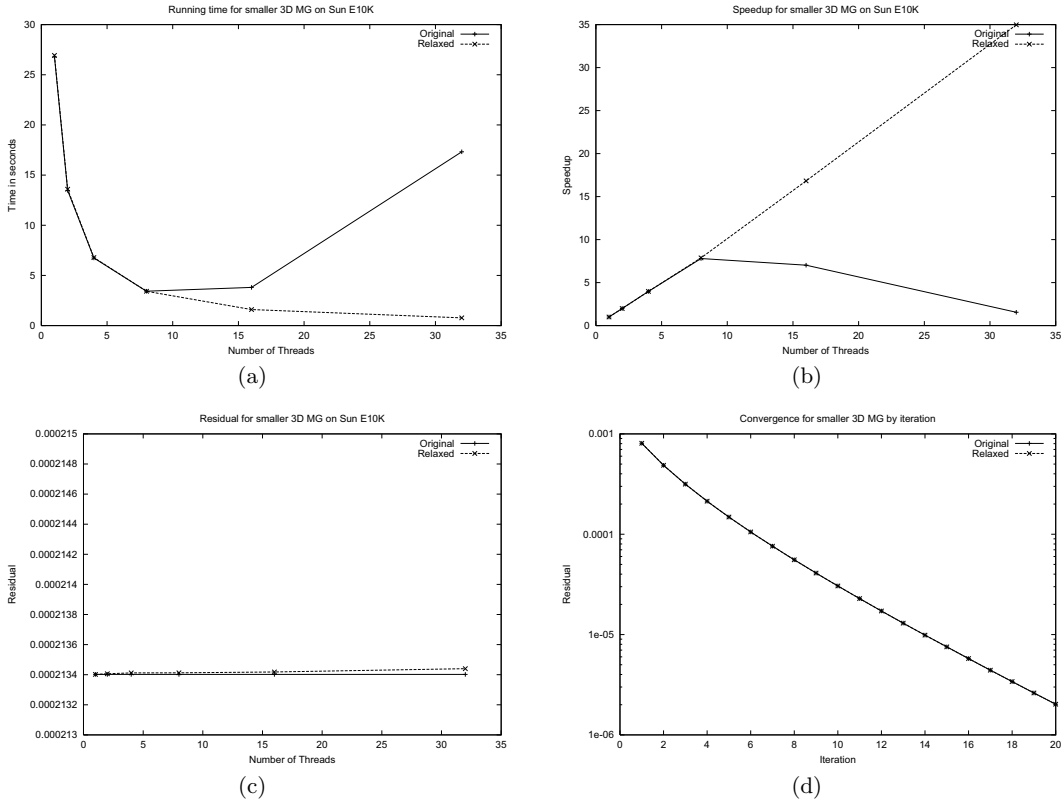

Fig. 7. Comparison of two data flow models with MG using a smaller 3D grid. (a) Running time (b) Parallel speedups (c) Final residual norms (d) Convergence rates

as figure 9(c) demonstrates. Even as such, the improvement in efficiency over the original version is quite significant.

5 Related Work

In [17], Saad presents a survey of work on Krylov subspace methods for parallel and vector computers. Meier and Eigenmann show performance enhancements to different conjugate gradient schemes using both manual and automatic parallelization on the Cedar architecture [14]. Methods to enhance the performance of relaxation schemes have also been studied in the past. Diniz and Yang [11] discuss efficient algorithms for parallelizing relaxation methods for banded systems. In [10], Chow, Falgout, Hu, Tuminaro, and Tang present a survey of parallelization techniques for multigrid solvers. These previous techniques, however, follow strict data flow in iterative steps, unlike our relaxed data flow model.

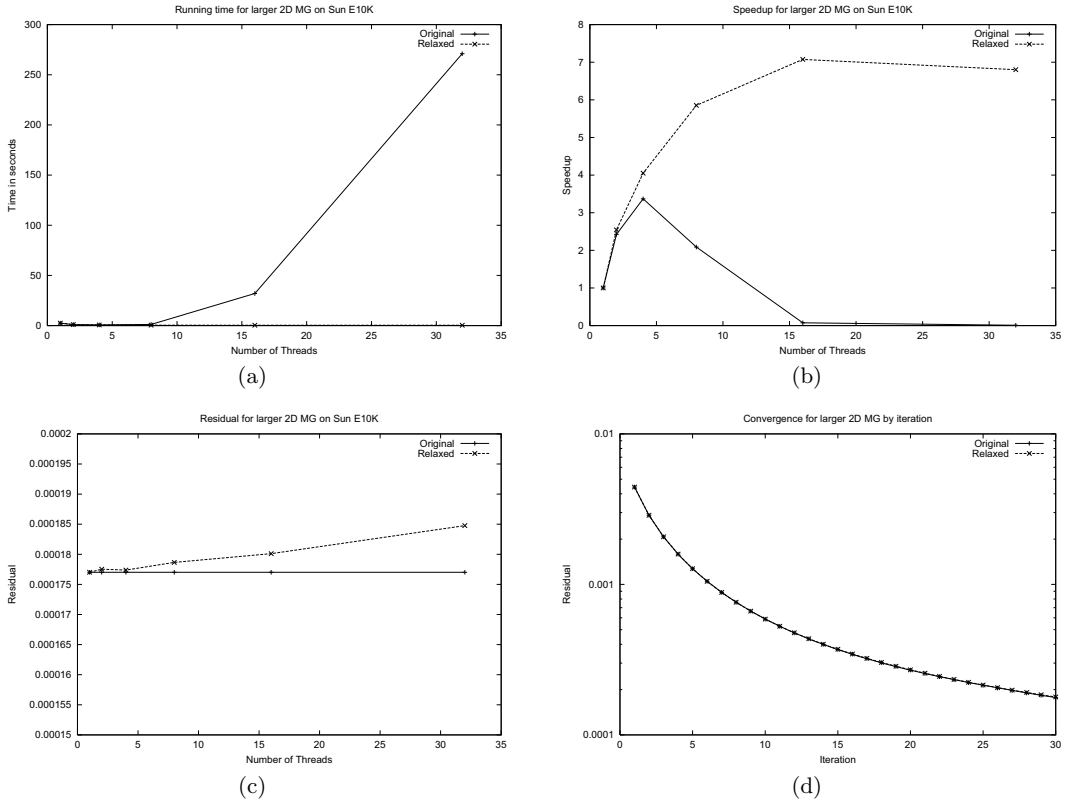

Fig. 8. (a) Running time of MG using a 2D grid (b) Parallel speedup (c) Final residual norms (d) Convergence rates

A considerable amount of prior work has been conducted on theories of asynchronous iterative algorithms in the past decades [9, 3, 4]. Most publications seem to have focused on developing general convergence criteria and tightening convergence conditions, although several have devoted themselves to specific iterative methods such as Jacobi, Gauss-Seidel and SOR [3, 2]. Applying the general theory of asynchronous computation model to a concrete iterative numerical method remains a nontrivial problem. Prior work has also investigated two-stage methods using the block-Jacobi iterative scheme to solve a system of linear equations [8, 5, 6]. Bru, Migallon, Penades, and Szyld study a comparison between parallel synchronous and asynchronous two-stage multisplitting algorithms [8]. Blathras, Szyld, and Shi [6] also investigate the stopping criteria of the inner method for the two-stage scheme using an asynchronous model.

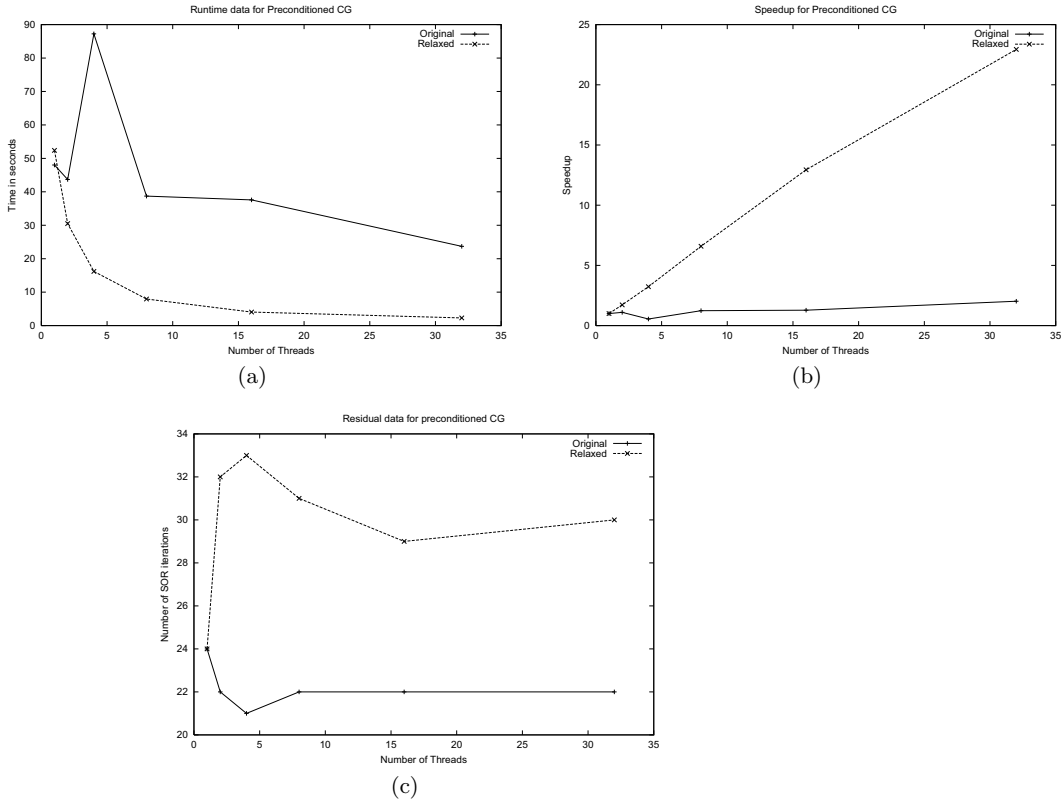

Fig. 9. Comparison of methods with SOR-preconditioned CG (a) Running times (b) Parallel speedups (c) Number of iterations of first invocation of SOR preconditioner

6 Conclusion

The number of processors in parallel computers have been steadily increased in recent years. The largest computational clusters now boast over ten thousands of processors. Interprocessor data communication is therefore becoming a more serious performance bottleneck. We have proposed three kinds of ASYNC loop constructs to support the asynchronous computation model for iterative solvers, which enables relaxed synchronization with significantly reduced performance penalty due to synchronization overhead.

The experimental results with 3D and 2D MG benchmarks and with SOR-preconditioned CG benchmark show excellent improvement of the ASYNC versions over the conventional OpenMP versions of these parallel programs in terms of the parallel execution efficiency. Moreover, the convergence rate has remained approximately the same. While these results are highly encouraging, we observe that deciding which synchronization points to remove remain a nontrivial task

which involves careful consideration of the numerical properties of the given iterative solver. We believe that the proposed new loop constructs make it easier for programmers to implement and fine tune asynchronous algorithms.

For our future work, we will further investigate other asynchronous algorithms. We hope to see more successes with various asynchronous algorithms, which will greatly motivate us to engage in a full implementation of the proposed ASYNC loop constructs in a parallelizing compiler.

References

1. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Fredrickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weerantunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA, March 1994.
2. R. H. Barlow and D. J. Evans. Parallel algorithms for the iterative solution to linear systems. *The Computer Journal*, 25(1):56–60, 1982.
3. Gérard M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25(2):226–244, 1978.
4. Dimitri P. Bertsekas and John N. Tsitsiklis. Convergence rate and termination of asynchronous iterative algorithms. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 461–470, New York, NY, USA, 1989. ACM.
5. K. Blathras, D. Szyld, and Y. Shi. Parallel processing of linear systems using asynchronous methods. Preprint, Temple University, Philadelphia, PA, April 1997, 1997.
6. Kostas Blathras, Daniel B. Szyld, and Yuan Shi. Timing models and local stopping criteria for asynchronous iterative algorithms. *Journal of Parallel and Disributed Computing*, 58:446–465, 1999.
7. OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2.5 edition, May 1990.
8. Rafael Bru, Violeta Migallón, José Penadés, and Daniel B. Szyld. Parallel, Synchronous and Asynchronous Two-stage Multisplitting Methods. *Electronic Transactions on Numerical Analysis*, 3:24–38, 1995.
9. D. Chazan and W. L. Miranker. Chaotic relaxation. *Linear Algebra and Its Application*, 2:199–222, 1969.
10. Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike Meier Yang. A Survey of Parallelization Techniques for Multigrid Solvers, 2006. SIAM Book Series: Frontiers of Parallel Processing for Scientific Computing.
11. Pedro C. Diniz and Tao Yang. Efficient Parallelization of Relaxation Iterative Methods for Solving Banded Linear Systems on Multiprocessors. In *PPSC*, pages 490–491, 1995.
12. J. Gu and Z. Li. Efficient interprocedural array data-flow analysis for automatic program parallelization. *IEEE Trans. on Software Engineering*, 26:244–261, 2000.
13. H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NASA, October 1999.
14. U. Meier and R. Eigenmann. Parallelization and performance of conjugate gradient algorithms on the Cedar hierarchical-memory multiprocessor. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 178–188, Williamsburg, VA, April 1991.

15. Sungdo Moon and Mary W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. *SIGPLAN Not.*, 34(8):84–95, 1999.
16. Sungdo Moon, Mary W. Hall, and Brian R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *International Conference on Supercomputing*, pages 204–211, 1998.
17. Yousef Saad. Krylov Subspace Methods on Supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, 1989.
18. Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Siam, second edition, 2003.