Auto-CFD: Efficiently Parallelizing CFD Applications on Clusters *

Li Xiao[1], Xiaodong Zhang[2], Zhengqian Kuang[3], Baiming Feng[4], and Jichang Kang[3]

[1]Department of Computer Science and Engineering, Michigan State University, U.S.A., lxiao@cse.msu.edu

[2]Department of Computer Science, College of William and Mary, U.S.A., zhang@cs.wm.edu

[3]Department of Computer Science and Engineering, Northwestern Polytechnical University, P. R. China, {kuangzq, kangjc}@nwpu.edu.cn

[4]Institute of Computing Technology, Chinese Academy of Sciences, P. R. China, fengbm@ict.ac.cn

Abstract

Computational Fluid Dynamics (CFD) applications are highly demanding for parallel computing. Many such applications have been shifted from expensive MPP boxes to cost-effective clusters. Auto-CFD is a pre-compiler which transforms Fortran CFD sequential programs to efficient message-passing parallel programs running on clusters. Our work has the following three unique contributions. First, this pre-compiler is highly automatic, requiring a minimum number of user directives for parallelization. Second, we have applied a dependency analysis technique for the CFD applications, called analysis after partitioning. We propose a mirror-image decomposition technique to parallelize self-dependent field loops that are hard to parallelize by existing methods. Finally, traditional optimizations of communication focus on eliminating redundant synchronizations. We have developed an optimization scheme which combines all the non-redundant synchronizations in CFD programs to further reduce the communication overhead. The Auto-CFD has been implemented on clusters and has been successfully used for automatically parallelizing structured CFD application programs. Our experiments show its effectiveness and scalability for parallelizing large CFD applications.

1. Introduction

In Computational Fluid Dynamics (CFD), people numerically simulate the complex flow of various types of fluids under different ranges of speeds on high performance computers. CFD has become one of the most important pillars in the fields of aviation and aerospace. The CFD numerical models and simulations are highly complex and computationally demanding for both CPU and memory storage. For example, a sample problem size for an external flow simulation element given by the Powered-Lift Group of the Applied Computational Fluids Branch at the NASA-Ames Research Center consists of 5,000,000 grid points, 50,000 iterations, and 5,000 floating point operations per point per iteration—10^{15} operations per problem [18].

It would be ideal for computational scientists to have both MPP machines and general-purpose parallelizing compilers for solving their increasingly large and complex application problems. However, on the system architecture side, many such applications have been shifted from MPP machines to clusters for both economic and technical reasons. On the system software side, parallelizing compilers are reasonably mature only for certain application programming models [23]. Parallelizing compiler development for important and specific classes of large problems is highly necessary. Auto-CFD is a pre-compiler which transforms Fortran CFD sequential programs to efficient message-passing parallel programs running on clusters. Our pre-compiler addresses the need of both system architecture and system software by targeting the programming models of CFD applications on clusters. Our work has the following three unique contributions. First, this pre-compiler is highly automatic, requiring a minimum number of user directives for parallelization. Second, we have applied a dependency analysis technique for the CFD applications, called *analysis after partitioning*. We propose a *mirror-image decomposition* technique to parallelize self-dependent field loops that are hard to parallelize by existing methods. Finally, traditional optimizations of communication focus on eliminating redundant synchronizations. We have developed an optimization scheme which combines all the non-redundant synchronizations in CFD programs to further reduce the communication overhead. The Auto-CFD has been implemented on clusters and has been successfully used for automatically parallelizing structured CFD programs. Our experiments show its effectiveness and scalability for parallelizing large CFD applications.

* This work was supported in part by the China National Aerospace Science Foundation on High Performance Computing Initiatives, and by the U.S. National Science Foundation under grants CCR-9812187, EIA-9977030, and CCR-0098055, and by Sun Microsystems under grant EDUE-NAFO-980405.

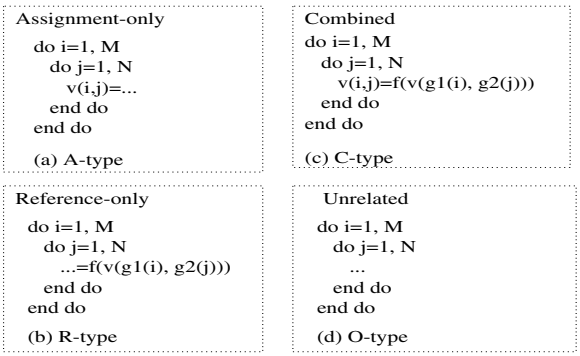

Figure 1: **Types of field loop.**

2. Computation Model of CFD Applications

Fluid movement patterns are expressed by Euler, Navier-Stokes and other equations [18]. Solving these equations by iterative numerical methods, CFD scientists characterize and study fluid movement phenomena for a wide range of applications.

In CFD applications, the first step is to generate a grid in an irregular physical flow field. The grid is then transformed into a computational grid in a regular shape, such as a rectangular grid. Each point in a grid corresponds to a group of data describing the flow field status in that area, such as velocities on different dimensions, densities, pressures, and temperatures. We use arrays to indicate this data for the whole grid, which are called *status arrays*. The second step is to initialize the arrays to describe the initial status of the flow field. A CFD simulation will eventually reach a stable flow field status after many iterations of computation. Each iteration may include several loops. Each iteration calculates all grid points in the field once. We define such an iteration as a *frame*. The iteration will not stop until a convergence condition is satisfied (the maximum data error of all grid points between the current iteration and the previous iteration should be less than a given error ϵ).

Jacobi and Gauss-Seidel iterations and their variants are commonly used iterative methods in CFD. Each grid point is calculated based on neighbor grid points. Kernel computations are based on five-point stencil and nine-point stencil models [18]. For accessing each status array, denoted as v, we have defined four types of loops in CFD programs shown in Figure 1. The four types are: A-type (Assignment-only), R-type (Reference-only), C-type (Combined assignment and reference), and O-type (Unrelated loop operations)

3. Software Structure of Auto-CFD

Figure 2 presents the basic structure of our pre-compiler and how it is connected in a parallel execution environment. The input of the pre-compiler is a CFD sequential source program. The output of the pre-compiler is a parallel CFD source program in SPMD model with communication statements (PVM/MPI calls). The pre-compiler first partitions the grid subject to communication optimizations. Each subgrid is assigned to a parallel subtask. The pre-compiler analyzes the dependency among subtasks to preliminarily determine the synchronization points. After optimizing synchronizations by eliminating redundant synchronization points and combining non-redundant synchronizations, the pre-compiler determines the final synchronization points and determines data items for message-passing in each synchronization point. Applying the analysis results, the pre-compiler finally restructures the sequential source code into optimized parallel source code. The restructuring procedure consists of inserting communication statements, modifying loop indices, redefining the sizes of arrays, modifying read file statements, and other related operations. The reference on the implementations of the restructuring procedure is given in [24].

4. Dependency Analysis

4.1. Grid Partitioning

We first briefly introduce grid partitioning techniques that are necessary technical background for dependency analysis. Grid partitioning serves for two purposes: (1) to balance the computation among the subtasks; and (2) to minimize communication during the computation among the subtasks. For tori grids, all the partitioned subgrids should be sized as equally as possible for load balancing. In addition, we have rigorously proved that the amount of communication is minimized if the grid is partitioned by finding the equal number of grid points (or as close to equal as possible) for all demarcation lines of a partitioned subgrid.

4.2. Dependency Analysis

After partitioning the grid, the pre-compiler starts to identify the communication data sets through dependency analysis. For a CFD application, the program structure and operations in field loops are complicated. We have considered the following 5 cases in the implementation of Auto-CFD:

(1) Multiple status arrays will be referenced in a Field-Loop. All the status arrays are analyzed in order to find the intersection sets made up of arrays to be assigned and arrays to be referenced. The dependency will be determined following this analysis.

(2) References in some field loops may not be a regular five-point stencil or a nine-point stencil. Some references in these field loops occur only on a certain dimension or in a certain direction.

(3) For grid points in boundaries, the program usually includes a special code section for those points.

(4) The number of dimensions of status arrays may be larger than the number of dimensions of the flow field. In some CFD applications, programmers may pack multiple status arrays into one high dimensional array. The extended dimensions due to the packing are not related to the grid partitioning. We should be able to

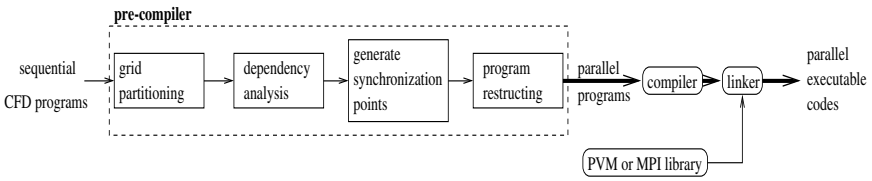

Figure 2: **Basic software structure.**

```
do i=1, M
  do j=1, N
    v(i,j)=f(v(i-1, j-1))
  end do
end do
```

(a)

```
do i=1, M
  do j=1, N
    v(i,j)=f(v(i-1, j), v(i+1, j), v(i, j-1), v(i, j+1))
  end do
end do
```

(b)

Figure 3: **Examples of self-dependent field loops.**

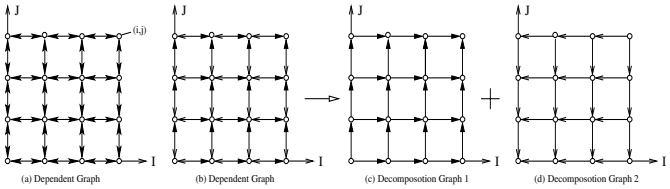

Figure 4: **Mirror-image decomposition.**

distinguish the extended dimensions from the original dimensions that are called *status dimensions*.

(5) We must identify the dependency distance (the number of grid points referenced in one direction of one dimension for one calculation) since in some CFD applications such as multiple-grids, it is likely that the dependency distance is larger than 1.

Our dependency test algorithm generates a set of field loop dependency pairs, called S_{LDP}. Each element in this set records a pair of dependent field loops and records other related information, such as dependent status arrays and dependency distances. Scanning the entire program section, the algorithm first identifies the loop structures and the statements referencing status arrays. The algorithm then searches for all pairs of assignment statements and reference statements, and further identifies the field loops including these statements. If the loop pair has been included in set S_{LDP}, additional related information for the loop pair is recorded. Otherwise the loop pair is added into the set. Dependent pairs in S_{LDP} consist of the complete dependent information. The information in S_{LDP} is sufficient for synchronizations, communications and program restructuring.

When a pair of dependent field loops (an A-type and an R-type) happens to be the same loop, the loop is called as *self-dependent field loop*. Figure 3(a) and (b) give two represented examples. Loops in Figure 3(a) which do not have dependences opposite to the lexicographic order can be parallelized using a wavefront method or a loop skewing technique [2, 22]. The dependence graph of the loops in Figure 3(b) is shown in Figure 4(a). This graph has both dependences in the lexicographic order and dependences opposite to the lexicographic order. The dependence distance is 1. Loops in Figure 3(b) are not parallelizable by traditional methods[2]. We have developed a method called *mirror-image decomposition* of the dependence graph to parallelize the self-dependent loops in Figure 3(b). The method first decomposes a dependency graph of a program (e.g. Figure 4(b)) into subgraphs (e.g.Figure 4(c) and (d)) based on the access direction of status arrays. Then traditional techniques of wavefront, or pipelining are applied to subgraphs. The detailed descriptions of the dependency test algorithm and mirror-image decomposition can be found in [24].

5. Synchronization and Communication Optimizations

Using the dependency analysis results, the pre-compiler determines where to do synchronization and communications and what data are needed for communications for each pair of dependent field loops. We focus on synchronization and communication optimizations between neighbor subtasks in this paper. A *synchronization point* refers to a position (or a line number) in a program, where the dependent data of a variable have been updated so that the update of this variable can proceed. The dependency analysis generates these synchronization points to ensure the correctness of a parallel program, but it does not consider its efficiency. The communication in a synchronization point refers to message passing of the dependent data between two neighbor computing nodes. We have developed optimization techniques to improve execution efficiency by reducing the number of synchronization points and by combining the communications.

5.1. Optimizations from combining synchronizations

We use $L =< index, S >$ to denote a loop, where $index$ is a loop variable, and S is the ordered set of statements making up a loop body. We also define an extended loop body denoted as $S^* = \{S_s\} \cup S \cup \{S_e\}$, where S_s is the do statement of the loop and S_e is the $end\ do$ statement. We give the following definitions:

Definition 6.1. *Inner Loop* and *Outer Loop*: For loop $L_1 =< I_1, S_1 >$ and $L_2 =< I_2, S_2 >$, if $S_2^* \subset S_1^*$, then L_2 is a inner loop of L_1, and L_1 is a outer loop of L_2, denoted as $L_2 \subset L_1$.

Definition 6.2. *Direct Inner Loop* and *Direct Outer Loop*: If $L_2 \subset L_1$, and there is no loop L_3 such that $L_2 \subset L_3 \subset L_1$, then L_2 is a direct inner loop of L_1, and L_1 is a direct outer loop of L_2, denoted as $L_1 \vdash L_2$.

Definition 6.3. *Adjacent Loops*: If both loop L_1 and loop L_2 do not have a outer loop, or both L_1 and L_2 have the

same direct outer loop, loops L_1 and L_2 are adjacent loops, denoted as $L_1 \parallel L_2$.

Definition 6.4. *Simple Loop*: Loop L is a simple loop if and only if $\{(L_1, L_2) | L_1 \subset L, L_2 \subset L, L_1 \parallel L_2\} = \Phi$.

5.1.1. Generating upper-bound regions for synchronization points

A synchronization and a communication are needed between each dependent field loop pair, $L^A \rightarrow L^R$ where L^A is an A-type field loop and L^R is an R-type field loop. The correctness of the program is ensured as long as the synchronization point is inserted at any position after L^A and before L^R. This is a legal region for the synchronization point. We call this region a *synchronization region*. The starting point of the synchronization is placed right after loop L^A, and the ending point of the synchronization is placed right before loop L^R. However, we may find areas, such as the areas inside inner loops, in a synchronization region where placing the synchronization point could cause redundant synchronizations. We define the synchronization region excluding such areas as an *upper-bound synchronization region*. We aim at identifying an upper-bound synchronization region for each synchronization point, and determining the minimum number of intersections of these upper-bound synchronization regions so that the numbers of synchronizations and communications can be minimized.

We have developed an algorithm to generate upper-bound synchronization regions. The idea of the algorithm consists of two parts: to move out the starting point of a synchronization region from loops as far as possible and then to determine its synchronization region. The algorithm first analyzes the direct inner loops in an non-simple loop as follows. The starting point can be moved out to the next level loop if there is no R-type loop. If there exists an R-type loop, the synchronization region has to be kept in this non-simple loop. The algorithm then determines the synchronization region as follows: if an R-type loop is found between the starting point and the end of the loop, the synchronization region is identified to be between the starting point and the R-type loop; otherwise the region is identified to be between the starting point and the end of the non-simple loop.

Figure 5 shows how our algorithm generates upper-bound synchronization regions. In Figure 5(a), the direct outer loop of the A-type loop is L_3 that does not include the R-type loop. So, the starting point of the synchronization region at the end of the A-type loop can be moved to the end of next level loop L_3. For the same reason, the starting point can be moved again, to the end of the L_2 loop. The direct outer loop of L_2 is L_1 that includes the R-type loop. Thus, the starting point cannot be further moved out. Now a synchronization region needs to be determined. Figure 5(b) shows two typical cases. In case 1, the R-type loop is included between the starting point and its direct outer loop L_1. The synchronization region starts from the starting point and ends in the beginning of the R-type loop, but excludes unrelated loops. In case 2, the R-type loop is not included between the starting point and its direct outer loop L_1. The synchronization region starts from the starting point, and ends at the end of the direct outer loop L_1, but excludes unrelated loops.

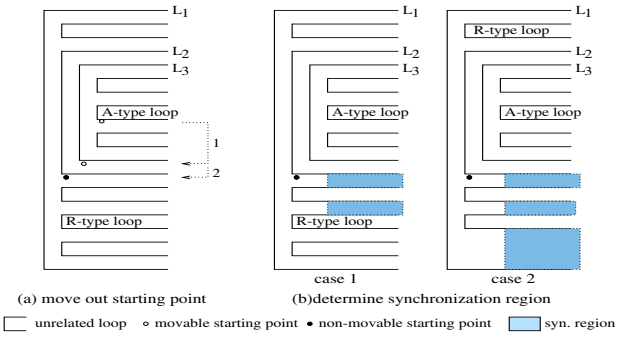

Figure 5: **Starting point movement and synchronization region identification in a non-simple loop.**

5.1.2. Combining synchronization regions for loops

As described above, one upper-bound synchronization region is generated for each synchronization point. Upper-bound synchronization regions for some synchronization points could be overlapped. So synchronization points of the overlapped regions can be merged as one single synchronization as long as the merged synchronization point is placed in the overlapped region. The optimal solution is to find the minimum number of regions so that the total number of synchronizations is minimized. We have developed and implemented an algorithm in Auto-CFD to achieve this optimization. The basic idea is as follows. All the upper-bound synchronization regions are sorted by the program line number of the first statement. An example of this sorting is shown in Figure 6(a) where 6 synchronization points are needed in the program. The 6 upper-bound synchronization regions have been created for the 6 synchronization points. Intersected regions are generated in the sorted order. A new intersection will not be generated until the currently sequenced region does not intersect with the existing intersections. Thus, the minimum number of intersections of the regions is found. Using this algorithm, the 6 upper-bound synchronization regions in Figure 6(a) have been combined into 2 regions, which is the minimum number of the intersections for block synchronizations (see Figure 6(b)). After optimization, the first three synchronizations are combined into one synchronization and corresponding communications are aggregated. Same thing happens to the last three synchronizations. If the regions are not intersected in this way, we may obtain more regions than the minimum. Figure 6(c) gives such an example in which the six regions are combined into 3 regions. The proof of the correctness and the minimization of the algorithm is given in [24].

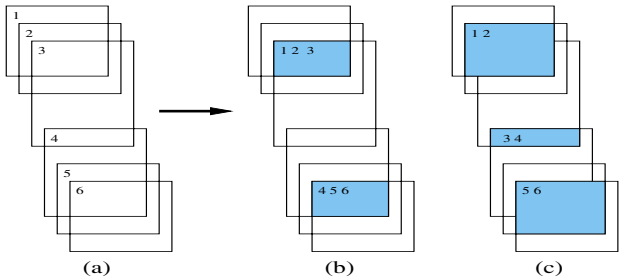

Figure 6: **Two different strategies (b) and (c) to combine synchronization points. Each square represents the upper bound synchronization region generated for a synchronization point. Shadow areas are intersections of upper-bound synchronization regions.**

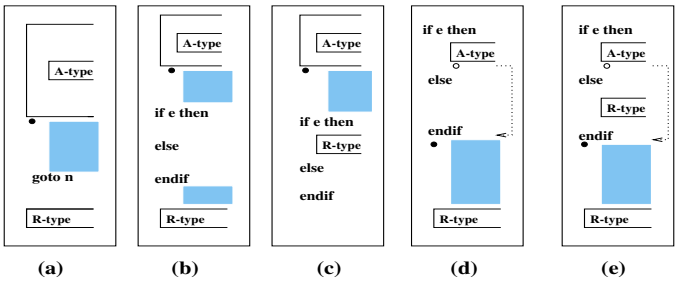

Figure 7: **Examples of determining upper-bound synchronization regions in branch structures. A circle represents a movable starting point, while a dot represents a non-movable starting point. Shadow areas represent upper-bound synchronization regions.**

5.2. Synchronization optimizations for branch structures

We have focused on optimizations for loops in the last section. We consider synchronizations in branch structures in this section. The upper-bound synchronization regions are determined in branch structures as follows:

1. If there is a *goto* statement in a synchronization region, the region will be ended before the *goto* statement.
2. When there is an $if-else$ block in a synchronization region, and if there is an R-type loop in the $if-else$ block, the synchronization region will be ended before the $if-else$ block; otherwise the region only needs to exclude the $if-else$ block.
3. If the starting point is in an $if-else$ block, it can be moved out of the block as long as there is not an R-type loop in the $if-else$ block.

We give examples to explain how synchronization regions are determined in Figure 7, where (a) is an example for case 1, (b) and (c) are examples for case 2, and (d) and (e) are examples for case 3. In Figure 7(e), there is an R-type loop in the $else$ block, but the R-type loop and the A-type loop in the $if-then$ block cannot be executed at the same time. Thus, the starting point after the A-type loop can be moved out of the $if-else$ block. Further optimization for the case in Figure 7(c) and for *while* loops can be found in [24]

5.3. Combining the synchronizations from multiple subroutines

An application program usually consists of many subroutines. We have investigated the possibility of combining synchronizations from multiple subroutines. Here is the outline of our work. When a subroutine call is met in the process of locating the synchronization region, the pre-compiler checks if there is an R-type loop in the subroutine. If so, a synchronization is installed before the subroutine call. Otherwise the synchronization region excludes this call statement. If there is a synchronization region in the end

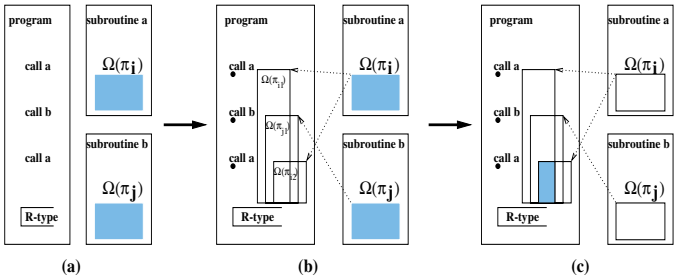

Figure 8: **Combining the synchronizations from multiple subroutines. A dot represents a starting point. Shadow areas are upper-bound synchronization regions for subroutines in (a) and (b). The shadow area in (c) is the combined synchronization region.**

of the subroutine, this region can be moved out of the subroutine, which could be combined with other upper-bound synchronization regions.

Figure 8 gives an example of optimizing synchronizations among multiple subroutines, where the main program calls subroutine a twice and subroutine b once (see Figure 8(a)). Without optimization in Figure 8(a), the execution needs three synchronizations in which two of them happen in subroutine a and one happens in subroutine b. As shown in Figure 8(b), there is a synchronization region, $\Omega(\pi_i)$, in the end of subroutine a. This region can be moved out of subroutine a to the main program. This region is further derived into a new region $\Omega(\pi_{i1})$ in the main program, which is after the first "call a" and before the R-type loop. Similarly, we have derived synchronization regions $\Omega(\pi_{j1})$ and $\Omega(\pi_{i2})$ for the "call b" statement and the second "call a" statement, respectively. The pre-compiler further combines $\Omega(\pi_{i1})$, $\Omega(\pi_{j1})$ and $\Omega(\pi_{i2})$ into a single region (see the shadow area in Figure 8(c)). After the optimization, only one synchronization is needed and communications are aggregated.

6. Experimental Results and Analysis

Using the Auto-CFD, we have conducted several case studies for aerospace engineering designs. We present two

Table 1: **Improvement by synchronization optimizations.**

program	partition	# of synchronization		percentage of
		before optimization	after optimization	optimization
case study 1 aerofoil simulation	$4 \times 1 \times 1$	73	8	89.0
	$1 \times 4 \times 1$	84	10	88.1
	$1 \times 1 \times 4$	81	9	88.9
	$4 \times 4 \times 1$	148	13	91.2
	$4 \times 1 \times 4$	145	13	91.0
	$1 \times 4 \times 4$	156	14	91.0
case study 2 flow simulation of sprayer	4×1	72	7	90.3
	1×4	69	7	89.9
	4×4	141	7	95.0

Table 2: **Overall performance of case study 1.**

flow field scale	$99 \times 41 \times 13$			
number of processors	partition	execution time (s)	speedup	parallel efficiency
1		1970	-	-
2	$2 \times 1 \times 1$	1760	1.12	56%
4	$4 \times 1 \times 1$	2341	0.84	21%
6	$3 \times 2 \times 1$	1093	1.80	30%

representative case studies in this paper. Case study 1 is an aerofoil simulation with 3,600 lines of Fortran code, and case study 2 is a flow simulation of sprayers with 6,100 lines of Fortran code. The aerofoil simulation does two major calculations: the distribution of the velocity on the aerofoil surface and the parameters of the flow close to the aerofoil surface (also called boundary layer analysis). The flow simulation of sprayers studies the air velocity for variations of sprayers, such as the sprayer fan speeds and fan positions. Both simulations are time-consuming CFD programs. The results presented here are measured from a dedicated networks of 6 Pentium workstations connected by Ethernet.

6.1. Performance improvement of the synchronization optimizations

Table 1 presents the performance improvement results from the synchronization optimization on the two case studies. In the "partition" column, $x \times y \times z$ means that the flow field is partitioned into x parts in dimension X, y parts in dimension Y, and z parts in dimension Z, respectively. Each of $x \times y \times z$ subgrids is assigned to a subtask for execution in a processor. We have shown that the pre-compiler is able to reduce the total number of synchronization points for both programs by about 90%. The overall execution performance results are presented in the next subsection.

Table 3: **Overall performance of case study 2.**

flow field scale	300×100			
number of processors	partition	execution time (s)	speedup	parallel efficiency
1		362	-	-
2	2×1	254	1.43	71%
3	3×1	184	1.97	66%
4	2×2	130	2.78	70%

6.2. Overall performance and scalability

Table 2 presents the parallel execution times of case study 1. This simulation includes a large number of self-dependent field-loops that are hard to parallelize by traditional methods [2]. Auto-CFD parallelizes these loops using the mirror-image decomposition and achieves some speedups. The relatively low efficiencies presented in Table 2 reflect this feature of the simulation program. The reason causing the speedup of the program on 4 processors to be lower than its speedup on 2 processors is as follows. On 2 processors, the best way to partition the flow field is to cut the longest dimension of 99 grid points. Each processor only needs to communicate demarcation grid points with the other processor. On 4 processors, one alternative partitioning is $4 \times 1 \times 1$. Each processor holding an non-boundary subtask needs to communicate with two neighbor processors. Thus, the number of grid points to be communicated per processor is the same as that on the 2 processor system. This means that the computation in each processor is half of that in the 2 processor system, and the communication is doubled. The computation and communication could not be fully overlapped due to the usage of mirror-image decomposition. Thus, the speedup decreased on the 4 processor system. We also ran this parallel program using 4 processors with another partitioning of $2 \times 2 \times 1$, and obtained similar result. Since each subgrid is $45 \times 21 \times 13$. Each processor needs to communicate with two other processors, and the total number of communicating grid points is 1.6 times, or $(45 \times 13 + 21 \times 13)/(41 \times 13)$ times of that in each processor in the 2 processor system. The communications of one processor to other two neighbor processors are not balanced. Thus, the speedup by this partitioning alternative also decreased, compared with the two processor system. The speedup increased on 6 processors because the partitioning of $3 \times 2 \times 1$ provided more balanced communication with neighbor processors and generated an less amount of communications than that in the 4 processor system with a partitioning of $2 \times 2 \times 1$.

Tables 3 to 5 present the performance results of case study 2. The overall performance presented in Table 3 indicates that this simulation can be more efficiently parallelized than the one in case study 1. Compared with the 2 processor system, the parallel efficiency of a 3 processor system decreased because its communication volume of the

Table 4: **Scaling performance of case study 2 with a partitioning of 2×1.**

flow field scale	execution time on 1 processor (s)	execution time on 2 processor system (s)	speedup of 2 processor system	parallel efficiency of 2 processor system
40×15	45	45	1	50%
60×23	108	66	1.64	82%
80×30	199	140	1.42	71%
100×38	331	218	1.52	76%
120×45	472	276	1.71	86%
140×53	712	403	1.77	88%
160×60	908	519	1.75	87%

Table 5: **Superlinear performance of case study 2.**

flow field scale	800×300		
number of processors	partition	execution time (s)	parallel efficiency over 2 processor system
2	2×1	2095	100%
3	3×1	1249	112%
4	2×2	1012	104%

processor holding the non-boundary task is doubled. Compared with the 3 processor system, the parallel efficiency of the 4 processor system increased because the amount of the computation per processor is further decreased so that caches are better utilized.

The performance results presented in Table 4 show the impact of the grid density to the parallel efficiency on a 2 processor system. The parallel efficiency increases as the grid density increases, except for the first two cases where their flow fields are small. This is because the ratio of computation to communication becomes larger as the grid density increases so that computation and communication can be further overlapped to gain better performance.

As the grid density increases to a certain degree, a workstation runs out of memory so that the execution slows down significantly. Further increasing the number of workstations could increase the accumulated memory size for solving a large density grid problem. If the problem size is sufficiently large but does not make workstations run out of their memories, the parallelized CFD program in case study 2 can efficiently utilize the cache in each workstation to achieve superlinear speedup performance. Table 5 presents the superlinear speedup performance results. As the grid density increases to 800×300, we obtain superlinear speedups on 2, 3, and 4 workstations.

7. Related work

We briefly overview some representative projects and comparisons with Auto-CFD. The related work mainly comes from the areas of pre-compiler transformation systems, parallel compilers, and algorithm design.

Pre-complier transformation systems. Both ADAPTOR (Automatic DAta Parallelism TranslaTOR) [4] and VAST_HPF [15] are source to source transformation systems that transform the data-parallel HPF programs into Fortran 77 or Fortran 90 programs. The input of our pre-compiler is a standard Fortran sequential program. Gnews [19] transforms a sequential Fortran program to parallel Fortran code with MPI calls. The user must insert directives into these sequential programs properly in order to obtain good performance. Gnews mainly performs the translation operations. The kernel of Gnews does not include any techniques to exploit program parallelism. In contrast, our pre-compiler includes parallelization optimizations for the source to source transformations. An early related work is the ParaScope Editor [8], which is a tool designed to help skilled users interactively transform a sequential Fortran 77 program into a parallel program.

Parallel compilers. Polaris compiler [16] [3] transforms a sequential Fortran77 program to parallel programs on both distributed shared memory systems and shared memory systems. Our target system is a message-passing oriented clusters. SUIF Explorer [12] is an interactive and interprocedural parallelizing compiler that has been implemented on a Digital TurboLaser machine. It attempts to provide a general-purpose parallelizer, leaving a few unresolved dependencies to users. Our pre-compiler orients a specific class of applications and requires little parallel programming skill from users.

POOMA(Parallel Object-Oriented Methods and Applications) [17] is an object-oriented framework for applications in computational science requiring high-performance parallel computers. It is a library of C++ classes designed to represent common abstractions in these applications. POOMA requires parallel programming skills from users.

Algorithms design. A large number of parallelizing algorithms have been proposed. Authors in [2] have evaluated some well-known systems and have concluded that no system is effective in covering different types of applications. The approach of incorporating all these algorithms into an optimizing compiler has its limitations for achieving optimal performance [11].

Aiming at exploiting data parallelism, both research groups at Rice and Stanford Universities propose algorithms to maximize parallelism and minimize communications [1, 11, 20]. Papers [1] and [11] use five-point stencil as one of their case studies. Paper [20] also uses some simple stencil examples, and mainly focuses on presenting optimization of CSHIFT operations in HPF format [7]. However, these examples represent a small part of a CFD program. In addition, their studies do not include $branch$, $while$ and $multi-module$ structures, but only consider DO loops. Our pre-compiler treats its in-

put as a whole package by considering different types of programming structures in CFD programs.

Authors in [6] use DAG matrix to minimize redundant dependencies and synchronizations. Eliminating redundant synchronizations in single loops on shared-memory multiprocessors was studied in early work [14]. Authors in [10] further studied removing redundant dependences in multiple nested loops. The author in [13] studied parallelizing iterative loops with conditional branching. The work in [21] focuses on optimizations for eliminating barrier synchronization of SPMD programs. Because our pre-compiler first partitions the grid as balanced as possible, the barrier synchronization used to determine the end of the CFD program execution will not be a sensitive factor that degrades performance. So we only focus on minimizing synchronization and communication among the grid neighbors. Paper [9] presents a global communication optimization technique based on data-flow analysis and linear algebra. It aims at eliminating redundancy while our work targets at combining non-redundant synchronizations in addition to eliminating redundant synchronizations. Another related work is [5] that presents a new compiler algorithm for global analysis and communication optimization in data-parallel programs. Again, this work does not consider optimization for complicated data structures to which we apply our optimization techniques.

8. Conclusion

We have presented a pre-compiler for automatically transforming a Fortran CFD sequential program into a parallel SPMD program. The pre-compiler needs some directives for specifying the CFD applications and the cluster system but does not require parallelizing skills from users. Appendix 1 presents the required directives, and Appendix 2 gives an example of the automatic transformation result from a sequential program to a parallel program. We have also discussed the mirror-image decomposition technique with dependence test algorithm briefly and the technique of combining all non-redundant synchronizations in detail, which are two technical contributions for exploiting CFD parallelism and further reducing synchronization and communication overhead. Experiments have shown their effectiveness for parallelizing structured CFD sequential programs for execution on clusters. This pre-compiler is also applicable to general message-passing parallel systems.

References

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam, "Data and Computation Transformation for Multiprocessors", *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP'95), 1995, pp.166-178.

[2] U. Banerjee, R. Eigenmann, A. Nicolau, D. A. Padua, "Automatic program parallelization", *Proceedings of the IEEE*, Vol. 81, No. 2, 1993, pp.211-243.

[3] W. Blume, R. Eigenmann, J. Hoeflinger, and D. Padua, "Automatic Detection of Parallelism: A grand Challenge for High-Performance Computing", *IEEE Parallel and Distributed Technology*, Vol. 2, No. 3, 1994, pp.37-47.

[4] T. Brandes and F. Zimmermann, "ADAPTOR - A Transformation Tool for HPF Programs", *Programming Environments for Massively Parallel Distributed Systems*, Springer Verlag, April 1994, pp. 91-96

[5] S. Chakrabarti, M. Gupta, and J.-D. Choi, "Global Communication Analysis and Optimization", *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996, pp.68-78.

[6] H. Y. Chao and M. P. Harper, "Minimizing redundant dependencies and interprocessor synchronizations", *International Journal of Parallel Programming*, Vol. 23, No. 3, 1995, pp.245-262.

[7] High Performance Fortran. http://www.crpc.rice.edu/HPFF/.

[8] M. W. Hall, T. J. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. H. Paleczny and G. Roth, "Experiences using the ParaScope Editor, an interactive parallel programming tool", *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP'93), April 1993, pp. 33-43.

[9] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam and N. Shenoy, "A global communication optimization technique based on data-flow analysis and linear algebra", *ACM Transactions on Programming Languages and Systems*, Vol. 21 , No. 6, 1999, pp.1251-1297.

[10] V. P. Krothapalli and P. Sadayappan, "Removal of Redundant Dependences in DOACROSS Loops with Constant Dependences", *Proceedings of the Third SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP'91), 1991, pp.51-60.

[11] A. W. Lim, G. I. Cheong, and M. S. Lam, "An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication", *Proceedings of ACM International Conference on Supercomputing* (ICS'99), 1999, pp.228-237.

[12] S. W. Liao, A. Diwan, R. P. Bosch Jr., A. Ghuloum, M. S. Lam, "SUIF Explorer: An Interactive and Interprocedural Parallelizer", *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP'99), 1999, pp.37-48.

[13] G. Lee, "Parallelizing Iterative Loops with Conditional Branching", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 2, 1995.

[14] S. Midkiff and D. Padua, "Compiler Algorithm for Synchronization", *IEEE Transactions on Computer*, Vol. C-36, No. 12, 1987, pp.1485-1495.

[15] Pacific-Sierra Research. http://www.psrv.com/vasthpf.html.

[16] Polaris. http://polaris.cs.uiuc.edu/polaris/polaris.html.

[17] POOMA. http://www.acl.lanl.gov/PoomaFramwork/index.html

[18] D. Roose and R. Van Driessche, *Parallel Computers and Parallel Algorithms for CFD: An Introduction*, Special Course on Parallel Computing in CFD, AGARD R-807, NATO, 1995, ISBN 92-836-1025-3, pp. 1.1-1.23.

[19] M. M. Rosing and S. Yabusaki, "A programmable preprocessor for parallelizing Fortran-90", *Proceedings of Supercomputing'99*, November 1999.

[20] G. Roth, J. Mellor-Crummey, and K. Kennedy, "Compiling Stencils in High Performance Fortran", *Proceedings of Supercomputing'97*, 1997.

[21] C. W. Tseng, "Compiler Optimizations for Eliminating Barrier Synchronization", *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (PPoPP'95), 1995, pp.144-155.

[22] W. Wolfe, "Loop Skewing: The Wavefront Method Revisited", *International Journal on Parallel Programming*, Vol. 15, No. 4, 1986, pp. 279-293.

[23] M. Wolfe, "Parallelizing Compilers", *ACM Computing Surveys*, Vol. 28, No. 1, March 1996.

[24] L. Xiao, X. Zhang, Z. Kuang, B. Feng, and J. Kang, "Auto-CFD: A pre-compiler for effectively parallelizing CFD applications on clusters", Technique Report, College of William and Mary.