540777

Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes

H. Jin,[†] M. Frumkin[†] and J. Yan
[†]*Computer Sciences Corporation*
NAS System Division, NASA Ames Research Center, Moffett Field, CA 94035-1000
{hjin,frumkin,yan}@nas.nasa.gov

Abstract

The shared-memory programming model is a very effective way to achieve parallelism on shared memory parallel computers. As great progress was made in hardware and software technologies, performance of parallel programs with compiler directives has demonstrated large improvement. The introduction of OpenMP directives, the industrial standard for shared-memory programming, has minimized the issue of portability. In this study, we have extended CAPTools, a computer-aided parallelization toolkit, to automatically generate OpenMP-based parallel programs with nominal user assistance. We outline techniques used in the implementation of the tool and discuss the application of this tool on the NAS Parallel Benchmarks and several computational fluid dynamics codes. This work demonstrates the great potential of using the tool to quickly port parallel programs and also achieve good performance that exceeds some of the commercial tools.

Keywords: OpenMP directives, computer-aided tools, automated parallel code generation, NAS Parallel Benchmarks.

1 Introduction

Porting applications to high performance parallel computers is always a challenging task. It is time consuming and costly. With rapid progressing in hardware architectures and increasing complexity of real applications in recent years, the problem becomes even more sever. Today, scalability and high performance are mostly involving hand-written parallel programs using message-passing libraries (e.g. MPI). However, this process is very difficult and often error-prone. The recent reemergence of shared-memory parallel (SMP) architectures, such as the cache coherent Non-Uniform Memory Access (ccNUMA) architecture used in the SGI Origin2000, show good prospects for scaling beyond hundreds of processors. Programming on an SMP is simplified by working in a globally accessible address space. The user can supply compiler directives to parallelize the code without explicit data partitioning. Computation is distributed inside a loop based on the index range regardless of data location and the scalability is achieved by taking advantage of hardware cache coherence. The recent emergence of OpenMP [13] as an industry standard offers a portable solution for implementing directive-based parallel programs for SMPs. OpenMP overcomes the portability issues encountered by machine-specific directives without sacrificing much of the performance and has gained popularity quickly.

Perhaps the main disadvantage of programming with directives is that inserted directives may not necessarily enhance performance. In the worst cases, it can create erroneous results when used incorrectly (writing message passing codes is even more error-prone). While vendors have provided tools to perform

error-checking and profiling [10], automation in directive insertion is very limited and often failed on large programs, primarily due to the lack of a thorough enough data dependence analysis. To overcome the deficiency, we have developed a toolkit, CAPO, to automatically insert OpenMP directives in Fortran programs and apply a degree of optimization. CAPO is aimed at taking advantage of detailed interprocedural data dependence analysis provided by Computer-Aided Parallelization Tools (CAPTools) [4], developed by the University of Greenwich, to reduce potential errors made by users and, with nominal help from user, achieve performance close to that obtained when directives are inserted by hand. Our approach is differed from other tools and compilers in two respects: 1) emphasizing the quality of dependence analysis and relaxing much of the time constraint on the analysis; 2) performing directive insertion and preserving the original code structure for maintainability. Translation of OpenMP codes to executables is left to proper OpenMP compilers.

In Section 2, we first outline the OpenMP programming model and give an overview of CAPTools and its extension, CAPO, for generating OpenMP programs. Then, in Section 3 we discuss the implementation of CAPO. Case studies of using CAPO to parallelize the NAS Parallel Benchmarks and two realistic computational fluid dynamics (CFD) applications are presented in Section 4 and conclusions are given in the last section.

2 Automatic Generation of OpenMP Directives

2.1 The OpenMP programming model

OpenMP [13] was designed to facilitate portable implementation of shared memory parallel programs. It includes a set of compiler directives and callable runtime library routines that extend Fortran, C and C++ to support shared memory parallelism. It promises an incremental path for parallelizing sequential software, as well as targeting at scalability and performance for any complete rewrites or new construction of applications.

OpenMP follows the *fork-and-join* execution model. A fork-and-join program initializes as a single lightweight process, called the *master thread*. The master thread executes sequentially until the first parallel construct (`OMP PARALLEL`) is encountered. At that point, the master thread creates a team of threads, including itself as a member of the team, to concurrently execute the statements in the parallel construct. When a work-sharing construct such as a parallel do (`OMP DO`) is encountered, the workload is distributed among the members of the team. An implied synchronization occurs at the end of the `DO` loop unless a "`NOWAIT`" is specified. Data sharing of variables is specified at the start of parallel or work-sharing constructs using the `SHARED` and `PRIVATE` clauses. In addition, reduction operations (such as summation) can be specified by the `REDUCTION` clause. Upon completion of the parallel construct, the threads in the team synchronize and only the master thread continues execution. The fork-and-join process can be repeated many times in the course of program execution.

Beyond the inclusion of parallel constructs to distribute work to multiple threads, OpenMP introduces a powerful concept of *orphan directives* that greatly simplifies the task of implementing coarse grain parallel algorithms. Orphan directives are directives outside the lexical extent of a parallel region. This allows the user to specify control or synchronization from anywhere inside the parallel region, not just from the lexically contained region.

2.2 CAPTools

The Computer-Aided Parallelization Tools (CAPTools) [4] is a software toolkit that was designed to automate the generation of message-passing parallel code. CAPTools accepts FORTRAN-77 serial code as input, performs extensive dependence analysis, and uses domain decomposition to exploit parallelism. The tool employs sophisticated algorithms to calculate execution control masks and minimize communication. The generated parallel codes contain portable interface to message passing standards, such as MPI and PVM, through a low-overhead library.

There are two important strengths that make CAPTools stands out. Firstly, an extensive set of extensions [5] to the conventional dependence analysis techniques has allowed CAPTools to obtain much more accurate dependence information and, thus, produce more efficient parallel code. Secondly, the tool contains a set of browsers that allow user to inspect and assist parallelization at different stages.

2.3 Generating OpenMP directives

The goal of developing computer-aided tools to help parallelize applications is to let the tools do as much as possible and minimize the amount of tedious and error-prone work performed by the user. The key to automatic detection of parallelism in a program and, thus parallelization is to obtain accurate data dependences in the program. Generating OpenMP directives is simplified somehow because we are now working in a globally addressed space without explicitly concerning data distribution. However, we still have to realize that there are always cases in which certain conditions could prevent tools from detecting possible parallelization, thus, an interactive user environment is also important.

The design of the CAPTools-based automatic parallelizer with OpenMP, CAPO, had kept the above tactics in mind. CAPO uses the data dependence analysis engine in CAPTools, exploits loop level parallelism in a program, and inserts OpenMP directives automatically. The schematic structure of CAPO is illustrated in Figure 1. The detailed implementation of the tool is given in Section 3. CAPO takes a serial code as input and first performs the data dependence analysis. User knowledge on certain input parameters in the source code may be entered to assist this analysis for more accurate results. The process of generating OpenMP directives is summarized in the following three stages.

1) *Identify parallel loops and parallel regions.* The loop-level analysis loops are classified as parallel (including reduction), serial or potential pipeline based on the data dependence information. Parallel loops to be distributed with work-sharing directives for parallel execution are identified by traversing the call graph of the program from top to down. Only outer-most parallel loops are considered, partly due to the very limited support of multi-level parallelization in available OpenMP compilers. Parallel regions are then formed around the distributed parallel loops. Attempt is also made to identify and create parallel pipelines. Details are given in Sections 3.1-3.3.

2) *Optimize loops and regions.* This stage is mainly for reducing overhead caused by fork-and-join and synchronization. A parallel region is first expanded as far as possible and may include calls to subroutines that contain additional (*orphaned*) parallel loops. Regions are then merged together if there is no violation of data usage in doing so. Region expansion is currently limited to within a subroutine. Synchronization optimization between loops in a parallel region is performed by checking if the loops can be executed asynchronously. Details are given in Sections 3.2 and 3.4.

3) *Transform codes and insert directives.* Variables in common blocks are analyzed for their usage in all parallel regions in order to identify threadprivate common blocks. If a private variable is used in a non-threadprivate common block, the variable is treated with a special code transformation. A routine needs to be duplicated if its usage conflicts at different calling points. Details are given in Sections 3.5-3.7.

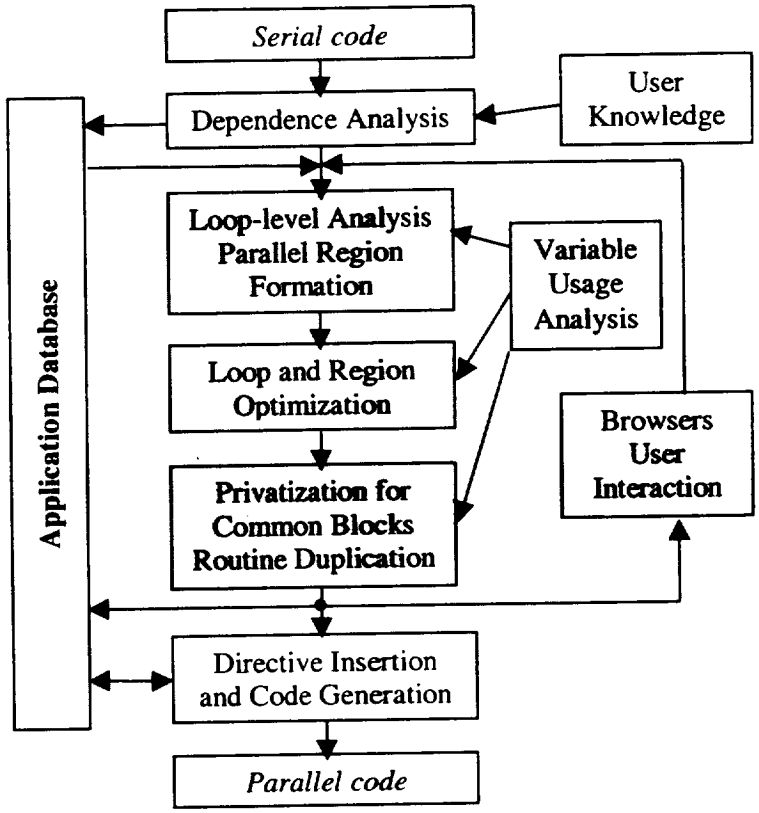

Figure 1: Schematic flow chart of the CAPO architecture.

By traversing the call graph one more time OpenMP directives are lastly added for parallel regions and parallel loops with variables properly listed. The variable usage analysis is performed at several points to identify how variables are used (e.g. private, shared, reduction, etc.) in a loop or region. Such analysis is required for the identification of loop types, the construction of parallel regions, the treatment of private variables in common blocks, and the insertion of directives.

Intermediate results can be stored into or retrieved from a database. User assistance to the parallelization process is possible through browsers implemented in CAPO (Directives Browser) and in CAPTools. The Directives Browser is designed to provide more interactive information from the parallelization process, such as reasons why loops are parallel or serial, distributed or not distributed. User can concentrate on areas where potential improvements could be made, for example, by removing false data dependences. It is part of the iterative process of parallelization.

3 Implementation

In the following subsections, we will give some implementation details of CAPO organized according to the components outlined in Section 2.3.

3.1 Loop-level analysis

In the loop-level analysis, the data dependence information is used to classify loops in each routine. Loop types include *parallel* (including *reduction*), *serial*, and *pipeline*. A parallel loop is a loop with no loop-carried data dependences and no exiting statements that jump out of the loop (e.g. `RETURN`). Loops with I/O (e.g. `READ`, `WRITE`) statements are excluded from consideration at this point. Parallel loop includes

the case where a variable is rewritten during the iteration of the loop but the variable can be privatized (i.e. having a local copy on each thread) to remove the loop-carried output dependence.

A reduction loop is considered as a special parallel loop since the loop can first update partial results in parallel on each thread and then update the final result atomically (or reduction). The reduction operations, such as "+", "-", "min", "max", etc. can be executed in a CRITICAL section.

A special class of loops, called *pipeline loop*, has loop-carried true dependencies and the lengths of these dependence vectors are determinable and with the same sign. Such a loop can potentially be used to form parallel pipelining with an outside loop nesting. Compiler techniques for finding pipeline parallelism through affine transforms are discussed in [11]. The pipeline parallelism can be implemented in OpenMP directives with point-to-point synchronization. This is discussed in Section 3.3.

A serial loop is a loop that can not be run in parallel due to loop-carried data dependences, I/O or exiting statements. However, a serial loop may be used for the formation of a parallel pipeline.

3.2 Setup of parallel region

In order to achieve good performance, it is not enough to simply stay with parallel loops at a finer grained level. In the context of OpenMP, it is possible to express coarser-grained parallelism with parallel regions. Our next task is to use the loop-level information to define these parallel regions.

There are several steps to construct parallel regions:

a) Identify parallel loops to be distributed by traversing the call graph in a top-down approach. Only outer-most parallel loops with enough granularity are considered. Parallel regions are then formed around the distributed parallel loops, including pipeline loops if no parallel loops can be found at the same level and parallel pipelines can be formed.

b) Expand each parallel region as much as possible in a routine to the top-most loop nest that contains no I/O and exiting statements and is not part of another parallel region. If a variable will be rewritten by multiple threads in the potential parallel region and the variable cannot be privatized (the memory access conflict test), back down one loop nest level. A reduction loop is in a parallel region by itself.

c) Include in the region any preceded code blocks that satisfy the memory access conflict test and are not yet included in other parallel regions. Orphaned directives will be used in routines that are called inside a parallel region but outside a distributed parallel loop.

d) Join two neighboring regions to form a larger parallel region if possible.

e) Treat parallel pipelines across subroutine boundaries if needed (see next subsection).

3.3 Pipeline setup

A potential pipeline loop (as introduced in Section 3.1) can be identified by analyzing the dependence vectors in symbolic form. In order to set up a parallel pipeline, an outer loop nest is required. If the top loop in a potentially parallel region is a pipeline loop and the loop is also in the top-level loop nesting of the routine, then the loop is further checked for loop nest in the immediate parent routine. The loop in the parent routine can be used to form an "upper" level parallel pipeline only if all the following tests are

true: a) such a parent loop nest exists, b) each routine called inside the parent loop contains only a single parallel region, and c) except for pipeline loops all distributed parallel loops can run asynchronously (see Section 3.4). If any of the tests failed, the pipeline loop will be treated as a serial loop.

OpenMP provides directives (e.g. "OMP FLUSH") and library functions to perform the point-to-point synchronization, which makes the implementation of pipeline parallelism possible with directives. These directives and functions are used to ensure the pipelined code section will not be executed in a thread before the work in the neighboring thread is done. Such an execution requires the scheduling scheme for the pipeline loop to be STATIC and ORDERED. Our implementation of parallel pipelines with directives is started from an example given in the OpenMP Program Application Interface [13]. The pipeline algorithm is used for parallelizing the NAS benchmark LU in Section 4.1 and also described in [12].

3.4 End-of-loop synchronization

Synchronization is used to ensure the correctness of program execution after a parallel construct (such as END PARALLEL or END DO). By default, synchronization is added at the end of a parallel loop. Sometime the synchronization at the end of a loop can be eliminated to reduce the overhead. We used a technique similar to the one in [15] to remove synchronization between two loops.

To be able to execute two loops asynchronously and to avoid a thread synchronization directive between them we have to perform a number of tests aside from the dependence information provided by CAPTools. The tests verify whether a thread executing a portion of the instructions of one loop will not read/write data read/written by a different thread executing a portion of another loop. Hence, for each non-private array we check that the set of written locations of the array by the first thread and the set of read/written locations of the array by the second thread do not intersect. The condition is known as the Bernstein condition (see [9]). If Bernstein condition (BC) is true the loops can be executed asynchronously. The BC test is performed in two steps. The final decision is made conservatively: if there is no proof that BC is true it set to be false. We assume that the same number of threads execute both loops, the number of threads is larger than one and there is an array read/written in both loops.

Check the number of loop iterations. Since the number of the threads can be arbitrary, the number of iterations performed in each loop must be the same. If it cannot be proved that the number of iterations is the same for both loops the Bernstein condition set to be false.

Compare array indices. For each reference to a non-privatizable array in the left hand side (LHS) of one loop and for each reference to the same array in another loop we compare the array indices. If we can not prove for at least one dimension that indices in both references are different then we set BC to be false. The condition can be relaxed if we assume the same thread schedule is used for both loops.

3.5 Variable usage analysis

Properly identifying variable usage is very important for the parallel performance and the correctness of program execution. Variables that would cause memory access conflict among threads need to be privatized so that each thread will work on a local copy. For cases where the privatization is not possible, for instance, a variable would partially be updated by each thread, the variable should be treated as shared and the work in the loop or region can only be executed in sequential (except for the reduction operation). Private variables are identified by examining the data dependence information, in particular, output

dependence for memory access conflict and true dependence for value assignment. Partial updating of variables is checked by examining array index expressions.

With OpenMP, if a private variable needs its initial value from outside a parallel region, the FIRSTPRIVATE clause can be used to obtain an initial copy of the original value; if a private variable is used after a parallel region, the LASTPRIVATE clause can be used to update the shared variable.

The reduction operation is commonly encountered in calculation. A typical implementation of parallel reduction has a private copy of each reduction that is first created for each thread, the local value is calculated on each thread, and the global copy is updated according to the reduction operator. OpenMP only supports reductions for scalar values. For array, we first transform the code section to create a local array and, then, update the global copy in a CRITICAL section.

3.6 Private variables in common blocks

For a private variable, each thread keeps a local copy and the original shared variable is untouched during the course of updating the local copy. If a variable declared in a common block is private in a loop, changes made to the variable through a subroutine call may not be updated properly for the local copy of this variable. If all the variables in the common block are privatizable in the whole program, the common block can be declared as *threadprivate*. However, if the common block can not be thread-privatized, additional care is needed to treat the private variable.

The following algorithm is used to treat private variables in a common block. The algorithm identifies and performs the necessary code transformation to ensure the correctness of variable privatization. Let us use the following convention: R_INSIDE for routine called inside a parallel loop, R_OUTSIDE for routine called outside a parallel loop, R_CALL for routine in a call statement, R_CALLBY for routine that calls the current routine, and V (or VC, VD, VN) for a variable named in a routine.

```
TreatPrivate(V, R_ORIG, callstatement) {
    check V usage in callstatement
    if V is not used in the call (via dependences) || is on the command parse tree
            || is not defined in a regular common block in a subroutine along the call path
        return
    TreatVinCall (VC, R_CALL) (V is referred as VC in R_CALL) {
        if VC is in the argument list of R_CALL
            return VC
        if R_CALL is R_OUTSIDE {
            if VC is not declared in R_CALL {
                replicate the common block in which V is named as VN
                set VC to VN from the common block
                set V to VN in the private variable list if R_CALL==R_ORIG
            }
        }
        else {
            add VC to the argument list of R_CALL
            if VC is defined in a common block of R_CALL
                add R_CALL & VC to RenList (for variable renaming later on)
            else declare VC in R_CALL
```

```
            for each calledby statement of R_CALL {
                VD is the name of VC used in R_CALLBY
                TreatVinCall(VD, R_CALLBY) and set VD to the returned value
                add VD to the argument of call statement to R_CALL in R_CALLBY
            }
            for each call statement in R_CALL
                TreatPrivate(VC, R_CALL, callstatement_in_R_CALL)
        }
        return VC
    }
}
```

Rename common block variables listed in RenList.

The algorithm starts with private variables listed for a parallel region in routine R_ORIG, one variable at a time. It is used recursively for each call statement in the parallel region along the call graph. A list of routine-variable pairs (R_CALL,VC) is stored in RenList during the process to track where private variables appear in common blocks. These variables in common blocks are renamed at the end.

As an example in Figure 2 the private array B is assigned inside subroutine SUB via the common block /CSUB/ in loop S2. Applying the above algorithm, the private variable B is added as C to the argument list of SUB and the original variable C in the common block in SUB is renamed to C_CAP to avoid usage conflict. In this way the local copy of B inside loop S2 will be updated properly in subroutine SUB.

```
S1   common /csub/ b(100),
     &           a(100,100)
 S2   do j=1, ny
 S3     call sub(j, nx)
        do i=1, nx
          a(i,j) = b(i)
        end do
      end do

 S4   subroutine sub(j, nx)
 S5   common /csub/ c(100),
     &           a(100,100)
      c(1) = a(1,j)
      c(nx) = a(nx,j)
      do i=2, nx-1
        c(i) = (a(i+1,j) +
     &          a(i-1,j))*0.5
      end do
      return
      end
```

```
  S1   COMMON/CSUB/B(100),A(100,100)
!$OMP PARALLEL DO PRIVATE(I,J,B)
  S2   DO J=1, NY
  S3     CALL SUB(J, NX, B)
         DO I=1, NX
           A(I,J) = B(I)
         END DO
       END DO
!$OMP END PARALLEL DO

  S4   SUBROUTINE SUB(J, NX, C)
  S5   COMMON/CSUB/C_CAP(100),A(100,100)
  S6   DIMENSION C(100)
       C(1) = A(1,J)
       C(NX) = A(NX,J)
       DO I=2, NX-1
         C(I) = (A(I-1,J)+A(I-1,J))*0.5
       END DO
       RETURN
       END
```

Figure 2: An example of treating a private variable in a common block.

3.7 Routine duplication

Routine duplication is performed after all the analyses are done but before directives are inserted. A routine needs to be duplicated if it causes usage conflicts at different calling points. For example, if a routine contains parallel regions and is called both inside and outside other parallel regions, the routine is

duplicated so that the original routine is used outside parallel regions and the second copy contains only orphaned directives without "OMP PARALLEL" and is used inside parallel regions. Routine duplication is often used in a message-passing program to handle different data distributions in the same routine.

4 Case Studies

We have applied CAPO to parallelize the NAS parallel benchmarks and two computational fluid dynamics (CFD) codes well known in the aerospace field: ARC3D and OVERFLOW. The parallelization (see Section 2.3) started with the interprocedural data dependence analysis on sequential codes. This step was the most computationally intensive part. The result was saved to an application database for later use. The loop and region level analysis was then carried out. At this point, the user inspects the result and decides if any changes are needed. The user assists the analysis by providing additional information on input parameters and removing any false dependences that could not be resolved by the tool. This is an iterative process, with user interaction involved. As we will see in the examples, the user interaction is nominal. OpenMP directives were lastly inserted.

In the case studies, we used an SGI workstation (R5K, 150MHz) and a Sun E10000 node to run CAPO. The resulting OpenMP codes were tested on an SGI Origin2000 system, which consisted of 64 CPUs and 16 GB globally addressable memory. Each CPU in the system is a R10K 195 MHz processor with 32KB primary data cache and 4MB secondary data cache. The SGI's MIPSpro Fortran 77 compiler (7.2.1) was used for compilation with the "-O3 -mp" flag.

4.1 The NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) were designed to compare the performance of parallel computers and are widely recognized as a standard indicator of computer performance. The NPB suite consists of five kernels and three simulated CFD applications derived from important classes of aerophysics applications. The five kernels mimic the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes. Details of the benchmark specifications can be found in [2] and the MPI implementations of NPB are described in [3].

In this study we used six benchmarks (LU, SP, BT, FT, MG and CG) from the sequential version of NPB2.3 [3] with additional optimization described in [7]. Parallelization of the benchmarks with CAPO is straightforward except for FT where additional user interaction was needed. User knowledge on the grid size (≥ 6) was entered for the data dependence analysis of BT, SP and LU. In all cases, the parallelization process for each benchmark took from tens of minutes up to one hour, most of the time being spent in the data dependence analysis. The performance of CAPO generated codes is summarized in Figure 3 together with comparison to other parallel versions of NPB: MPI from NPB2.3, hand-coded OpenMP [7], and versions generated with the commercial tool SGI-PFA [17].

CAPO was able to locate effective parallelization at the outer-most loop level for the three application benchmarks and automatically pipelined the SSOR algorithm in LU. As shown in Figure 3, the performance of CAPO-BT, SP and LU is within 10% to the hand-coded OpenMP version and much better than the results from SGI-PFA. The SGI-PFA curves represent results from the parallel version generated by SGI-PFA without any change for SP and with user optimization for BT (see [17] for details). The

worse performance of SGI-PFA simply indicates the importance of accurate interprocedural dependence analysis that usually cannot be emphasized in a compiler. It should be pointed out that the sequential version used in the SGI-PFA study was not optimized, thus, the sequential performance needs to be counted for the comparison. The hand-coded MPI versions scaled better, especially for LU. We attribute the performance degradation in the directive implementation of LU to less data locality and larger synchronization overhead in the 1-D pipeline used in the OpenMP version as compared to the 2-D pipeline used in the MPI version. This is consistent with the result of a study from [12].

The basic loop structure for the Fast Fourier Transform (FFT) in one dimension in FT is as follows.

```
DO K=1,D3
 DO J=1,D2
  DO I=1,D1
   Y(I) = X(I,J,K)
  END DO
  CALL CFFTZ(...,Y)
  DO I=1,D1
   X(I,J,K) = Y(I)
  END DO
 END DO
END DO
```

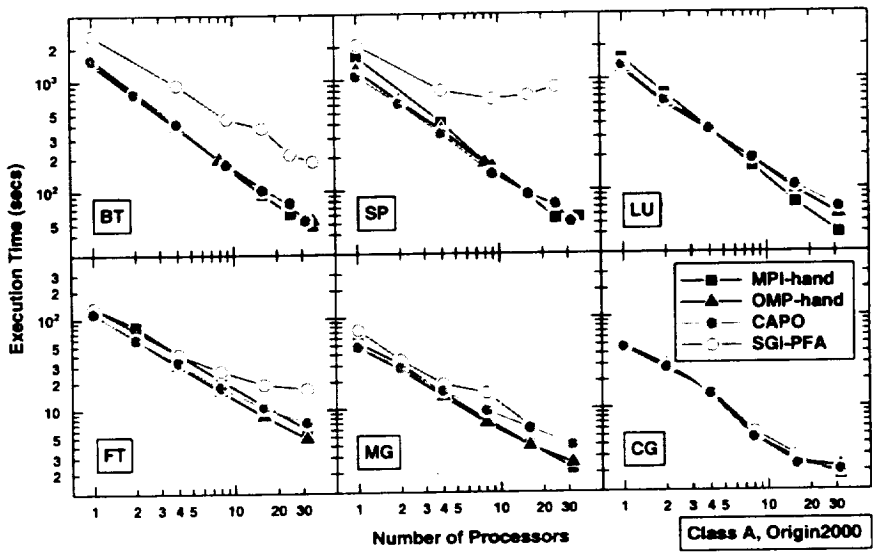

Figure 3: Comparison of the OpenMP NPB generated by CAPO with other parallel versions: MPI from NPB2.3, OpenMP by hand, and SGI-PFA.

A slice of the 3-D data (X) is first copied to a 1-D work array (Y). The 1-D FFT routine CFFTZ is called to work on Y. The returned result in Y is then copied back to the 3-D array (X). Due to the complicated pattern of loop limits inside CFFTZ, CAPTools could not disprove the loop-carried true dependences by the working array Y for loop K. These dependences were deleted by hand in CAPO to identify the K loop as a parallel loop.

The resulted parallel FT code gave a reasonable performance as indicated by the curve with filled circles in Figure 3. It does not scale as well as the hand-coded versions (both in MPI and OpenMP), mainly due to the unparallelized code section for the matrix creation which was artificially done with random number generators. Restructuring the code section was done in the hand-coded version to parallelize the matrix creation. Again, the SGI-PFA generated code performed worse.

The directive code generated by CAPO for MG performs 36% worse on 32 processors than the hand-coded version, primarily due to an unparallelized loop in routine `norm2u3`. The loop contains two reduction operations of different types. One of the reductions was expressed in an IF statement, which was not detected by CAPO, thus, the routine was ran in serial. Although this routine takes only about 2% of the total execution time on a single node, it translates into a large portion of the parallel execution on large number of processors, for example, 40% on 32 processors. All the parallel versions achieved similar results for CG.

4.2 ARC3D

ARC3D is a moderate-size CFD application. It solves Euler and Navier-Stokes equations in three dimensions using a single rectilinear grid. ARC3D has a structure similar to NPB-SP but contains curve linear coordinates, turbulent models and more realistic boundary conditions. The Beam-Warming algorithm is used to approximately factorize an implicit scheme of finite difference equations, which is then solved in three directions alternatively.

For generating the OpenMP parallel version of ARC3D, we used a serial code that was already optimized for cache performance by hand [16]. The parallelization process with CAPO was straightforward and OpenMP directives were inserted without further user interaction. The parallel version was tested on the Origin2000.and the result for a 194x194x194-size problem is shown in the left panel of Figure 4. The results from a hand-parallelized version with SGI multi-tasking directives (*MT by hand*) [16] and a message-passing version generated by CAPTools (*CAP MPI*) [8] from the same serial version are also included in the figure for comparison.

As one can see from the figure, the OpenMP version generated by CAPO is essentially the same as the hand-coded version in performance. This is indicative of the accurate data dependence analysis and sufficient parallelism that was exploited in the outer-most loop level. The MPI version is about 10% worse than the directive-based versions. The MPI version uses extra buffers for communication and this could contribute to the increase of execution time.

4.3 OVERFLOW

OVERFLOW is widely used for airflow simulation in the aerospace community. It solves compressible Navier-Stokes equations with first-order implicit time scheme, complicated turbulence model and Chimera boundary condition in multiple zones. The code has been parallelized by hand [6] with several approaches: PVM for zone-level parallelization only, MPI for both inter- and intra-zone parallelization, multi-tasking directives, and multi-level parallelization. This code offers a good test case for our tool not only because of its complexity but also its size (about 100K lines of FORTRAN 77).

In this study, we used the sequential version (1.8f) of OVERFLOW. CAPO took 25 hours on a Sun E10K node to complete the data dependence analysis. A fair amount of effort was spent on pruning data dependences that were placed due to lack of necessary knowledge during the analysis. An example of false dependence is illustrated in the following code segment:

```
      NTMP2 = JD*KD*31
      DO 100 L=LS,LE
        CALL GETARX(NTMP2,TMP2,ITMP2)
        CALL WORK(L,TMP2(ITMP2,1),TMP2(ITMP2,7),...)
        CALL FREARX(NTMP2,TMP2,ITMP2)
  100 CONTINUE
```

Inside the loop nest, the memory space for an array TMP2 is first allocated by GETARX. The working array is then used in WORK and freed afterwards. However, the data analysis has reviewed that the loop contains loop-carried true dependences caused by variable TMP2, thus, the loop can only be executed in serial. The memory allocation and de-allocation are performed dynamically and cannot be handled by CAPO. This kind of false dependence can safely be removed with Dependence Browser included in the tool. Even so, CAPO provides an easy way for user to interact with the parallelization process. The

OpenMP version was generated within a day after the analysis was completed and an additional few days were used to test the code.

The right panel of Figure 4 shows the execution time per time-iteration of the CAPO-OMP version compared with the hand-coded MPI version and hand-coded directive (MT) version. All three versions were running with a test case of size 69×61×50, 210K grid points in single zone. Although the scaling is not quite linear (when comparing to ARC3D), especially for more than 16 processors, the CAPO version out-performed both hand-coded versions. The MPI version contains sizable extra codes [6] to handle intra-zone data distributions and communications. It is not surprising that the overhead is unavoidably large. However, the MPI version is catching up with the CAPO-OMP version on large number of processors. On the other hand, further review has indicated that the multi-tasking version used a fairly similar parallelization strategy as CAPO did, but in quite a few small routines the MT version did not place any directives for the hope that the compiler (SGI-PFA in this case) would automatically parallelize loops inside these routines. The performance number seemed to have indicated otherwise.

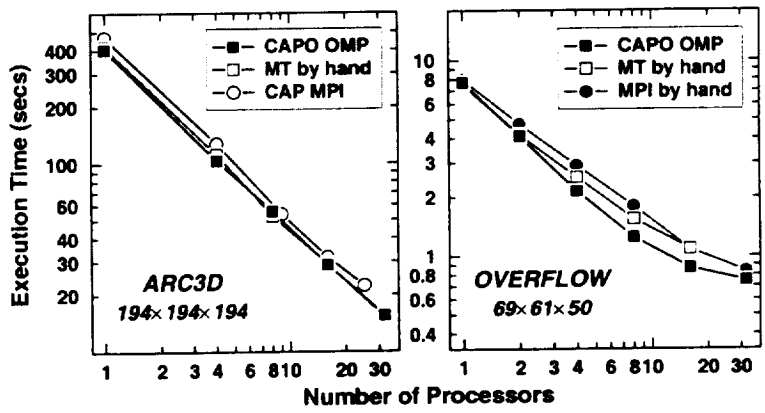

Figure 4: Comparison of execution times of CAPO generated parallel codes with hand-coded parallel versions for two CFD applications: ARC3D on the left and OVERFLOW on the right.

We also tested with a large problem of 1.5M grid points. The result was not included in the figure but CAPO's version has achieved 18-fold speedup on 32 processors of the Origin2000 (10 out of 32 for the small test case). It is not surprising that the problem with large grid size has achieved better parallel performance.

5 Related Work

There are a number of tools developed for code parallelization on both distributed and shared memory systems. The KAPro-toolkit [10] from Kuck and Associates, Inc. performs data dependence analysis and automatically inserts OpenMP directives in a certain degree. KAI has also developed several useful tools to ensure the correctness of directives insertion and help user to profile parallel codes. The SUIF compilation system [18] from Standard is a research product that is targeted at parallel code optimization for shared-memory system at the compiler level.

The SGI's MIPSpro compiler includes a tool, PFA, that tries to automatically detect loop-level parallelism, insert compiler directives and transform loops to enhance their performance. SGI-PFA is available on the Origin2000. Due to the constraints on compilation time, the tool usually cannot perform a comprehensive dependence analysis, thus, the performance of generated parallel programs is very limited. User intervention with directives is usually necessary for better performance. For this purpose, Parallel Analyzer View (PAV), which annotate the results of dependence analysis of PFA and present them

graphically, can be used to help user insert directives manually. More details of a study with SGI-PFA can be found in [17].

VAST/Parallel [14] from Pacific-Sierra Research is an automatic parallelizing preprocessor. The tool performs data dependence analysis for loop nests and supports the generation of OpenMP directives.

Parallelization tools like FORGExplorer [1] and CAPTools [4] emphasize the generation of message passing parallel codes for distributed memory systems. These tools can easily be extended to handle parallel codes in the shared-memory arena. Our work is such an example. As discussed in previous sections, the key to the success of our tool is the ability to obtain accurate data dependences combined with user guidance. An ability to handle large applications is also important.

6 Conclusion and Future Work

In summary, we have developed the tool CAPO that automatically generates directive-based parallel programs for shared memory machines. The tool has been successfully used to parallelize the NAS parallel benchmarks and several CFD applications with CAPO, as summarized in Table 1 which included also information for another CFD code, INS3D, the tool was applied to.

Table 1: Summary of CAPO applied on the NAS Parallel Benchmarks and three CFD applications.

Application	***NPB***		***ARC3D***	***OVERFLOW***	***INS3D***
	BT,SP,LU	***FT,CG,MG***			
Code Size	~3000 lines/ benchmark	~2000 lines/ benchmark	~4000 lines	851 routines, 100K lines	256 routines, 41K lines
Code Analysis a)	30 mins to 1 hour	10 mins to 30 mins	40 mins	25 hours	42 hours
Code Generation b)	1 min	1 min	1 min	1 day	2 days
Testing c)	1 day	1 day	1 day	3 days	3 days
Performance Compared to Hand-coded Version	within 5-10%	within 10% for CG 30-36% for FT,MG	within 6%	slightly better (see text in Section 4.3)	no hand-coded parallel version

a) *"Code Analysis" refers to time spent on the data dependence analysis, for NPB and ARC3D on an SGI Indy workstation and for OVERFLOW and INS3D on a Sun E10000 node.*

b) *"Code Generation" includes time user spent on interacting with the tool and code restructuring by hand (only for INS3D in four routines). The restructure involves mostly loop interchange and loop fuse that cannot be done by the tool.*

c) *"Testing" includes debugging and running a code and collecting results.*

By taking advantage of the intensive data dependence analysis from CAPTools. CAPO has been able to produce parallel programs with performance close to hand-coded versions in a relatively short period of

time. It should be pointed out, however, that the results did not show the effort in cache optimization of the serial code, such as for ARC3D. Our approach is different from parallel compilers in that it spends much of its time on whole program analysis to discover accurate dependence information. The generated parallel code is produced using a source-to-source transformation with very little modification to the original code and, therefore, is easily maintainable.

For larger and more complex applications such as OVERFLOW, it is our experience that the tool will not be able to generate efficient parallel codes without any user interactions. The importance of a tool, however, is its ability to quickly pinpoint the problematic codes in this case. CAPO (via Directives Browser) was able to point out a small percentage of code sections where user interactions were required for the test cases.

Future work will be focused in the following areas:

- Include a performance model for optimal placement of directives.
- Apply data distribution directives (such as those defined by SGI) rather than relying on the automatic data placement policy, *First-Touch*, by the operating system to improve data layout and minimize number of costly remote memory reference.
- Develop a methodology to work in a hybrid approach to handle parallel applications in a heterogeneous environment or a cluster of SMP's. Exploiting multi-level parallelism is important.
- Develop an integrated working environment for sequential optimization, code transformation, code parallelization, and performance analysis.

CAPO is available for testing. A copy of the tool can be obtained from the authors.

Acknowledgement: The authors wish to thank members of the CAPTools team at the University of Greenwich for their support on CAPTools and Dr. James Taft and Dr. Dennis Jespersen at NASA Ames for their support on the CFD applications used in the study. This work is supported by NASA Contract No. NAS2-14303 with MRJ Technology Solutions and No. NASA2-37056 with Computer Sciences Corporation.

References

[1] Applied Parallel Research Inc., "FORGE Explorer," http://www.apri.com/.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The NAS Parallel Benchmarks," *NAS Technical Report RNR-91-002*, NASA Ames Research Center, Moffett Field, CA, 1991.

[3] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *RNR-95-020*, NASA Ames Research Center, 1995. NPB2.3, http://www.nas.nasa.gov/Software/NPB/.

[4] C.S. Ierotheou, S.P. Johnson, M. Cross, and P. Legget, "Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, 22 (1996) 163-195. (http://captools.gre.ac.uk/)

[5] S.P. Johnson, M. Cross, and M.G. Everett, "Exploitation of symbolic information in interprocedural dependence analysis," *Parallel Computing*, 22 (1996) 197-226.

[6] D.C. Jespersen, "Parallelism and OVERFLOW," *NAS Technical Report NAS-98-013*, NASA Ames Research Center, Moffett Field, CA, 1998.

[7] H. Jin, M. Frumkin and J. Yan., "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," *NAS Technical Report*, NAS-99-011, NASA Ames Research Center, 1999.

[8] H. Jin, M. Hribar and J. Yan, "Parallelization of ARC3D with Computer-Aided Tools," *NAS Technical Report*, NAS-98-005, NASA Ames Research Center, 1998.

[9] C.H. Koelbel, D.B. Loverman, R. Shreiber, GL. Steele Jr., M.E. Zosel. "The High Performance Fortran Handbook," MIT Press, 1994, page 193.

[10] Kuck and Associates, Inc., "Parallel Performance of Standard Codes on the Compaq Professional Workstation 8000: Experiences with Visual KAP and the KAP/Pro Toolset under Windows NT," Champaign, IL; "Assure/Guide Reference Manual," 1997.

[11] Amy W. Lim and Monica S. Lam. "Maximizing Parallelism and Minimizing Synchronization with Affine Transforms," The 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, Jan,. 1997.

[12] X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gonzalez and J. Labarta, "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors," 1999 ACM International Conference on Supercomputing, Rhodes, Greece, 1999.

[13] OpenMP Fortran/C Application Program Interface, http://www.openmp.org/.

[14] Pacific-Sierra Research, "VAST/Parallel Automatic Parallelizer," http://www.psrv.com/.

[15] E.A. Stohr and M.F.P. O'Boyle, "A Graph Based Approach to Barrier Synchronisation Minimisation," in the Proceeding of 1997 ACM International Conference on Supercomputing, page 156.

[16] J. Taft, "Initial SGI Origin2000 Tests Show Promise for CFD Codes," *NAS News*, July-August, page 1, 1997. (http://www.nas.nasa.gov/Pubs/NASnews/97/07/article01.html)

[17] A. Waheed and J. Yan, "Parallelization of NAS Benchmarks for Shared Memory Multiprocessors," in Proceedings of High Performance Computing and Networking (HPCN Europe '98), Amsterdam, The Netherlands, April 21-23, 1998.

[18] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W.K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica Lam, and John Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," Computer Systems Laboratory, Stanford University, Stanford, CA.