

A Comparison of Automatic Parallelization Tools/Compilers on the SGI Origin2000

Michael Frumkin, Michelle Hribar, Haoqiang Jin, Abdul Waheed and Jerry Yan
MRJ Technology Solutions, Inc., NASA Ames Research Center
MS T27A-2, Moffett Field, CA 94035-1000
{frumkin, hribar, hjin, waheed, yan}@nas.nasa.gov

Abstract. *Porting applications to new high performance parallel and distributed computing platforms is a challenging task. Since writing parallel code by hand is time consuming and costly, porting codes would ideally be automated by using some parallelization tools and compilers. In this paper, we compare the performance of three parallelization tools and compilers based on the NAS Parallel Benchmark and a CFD application, ARC3D, on the SGI Origin2000 multiprocessor. The tools and compilers compared include: 1) CAPTools: an interactive computer aided parallelization toolkit, 2) Portland Group's HPF compiler, and 3) the MIPSPro FORTRAN compiler available on the Origin2000, with support for shared memory multiprocessing directives and MP runtime library. The tools and compilers are evaluated in four areas: 1) required user interaction, 2) limitations, 3) portability and 4) performance. Based on these results, a discussion on the feasibility of computer-aided parallelization of aerospace applications is presented along with suggestions for future work.*

1 Introduction

High performance computers have evolved rapidly over the past decade. During the past few years, the majority of HPC systems in use were distributed memory computers, ranging from small clusters of workstations or PC's to large-scale machines from IBM, Cray, Intel, and other vendors. Recently, shared memory computers have experienced resurgence; in particular, multiprocessor workstations and PCs are becoming commonplace, and are being used as components of distributed memory systems. Thus, computing platforms are now combining elements of both shared and distributed memory. Even more recently, underlying infrastructure to support the creation of wide area networks of computers, including large-scale machines, has been developed. Projects such as NPACI [\[1\]](#), and PACI [\[2\]](#) have aimed to provide a user with transparent access to such a "computational grid", which is essentially a distributed, heterogeneous collection of parallel computers. Such grids pose many new requirements with respect to programming models, compilation strategies and execution environments.

During that same period, scientists at research laboratories across the country were required to expend a very large effort porting codes to new architectures in an attempt to fully utilize their

performance potential. Not only are they forced to struggle with immature hardware and system software, but the large performance gap between the “theoretically possible” and the “actually achieved” has resulted in extensive efforts to re-code applications by-hand using machine-specific optimizations. While these efforts did narrow the said performance gap, many major supercomputer users, including those in the aerospace industry, have remained on the sidelines. In anticipation of the increasing complexity and size of future applications, industrial users will not be able to afford such porting efforts every few years. Instead, in order to protect investments in code maintenance and development, technology must be developed to transform the parallelization process such that it 1) requires less time and effort, 2) generates codes with good performance, 3) is able to handle a wide range of existing legacy applications and 4) is portable to future machines. Currently, there are four major approaches besides hand-coding for porting applications to parallel architectures. These include:

- relying on vendor-supplied parallelizing compilers/tools [\[3-5\]](#);
- annotating/rewriting application using data- and task-parallel directives/languages [\[6-11\]](#);
- rewriting application codes using semi-custom building blocks or libraries [\[12, 13\]](#); and
- using interactive parallelization/tuning environments [\[4, 14-16\]](#).

These approaches differ from one another in terms of relevant target applications, pertinent architectures, requirements for user involvement, as well as maturity. Therefore, the effectiveness of these approaches is a worthwhile study, especially for users interested in modernizing large legacy applications onto evolving HPC architectures.

This paper reports an initial comparison of the effectiveness of some of these approaches using NAS Parallel Benchmarks [\[17\]](#) and a computational fluid dynamics (CFD) application, ARC3D [\[18\]](#), as an initial test suite. The SGI Origin2000, a distributed shared memory computer, was selected as the platform for evaluation because of its proven performance with NPB 2.3 [\[19\]](#), a hand-written version of the benchmarks using MPI. The evaluation of parallel libraries was omitted because of large re-coding cost. The candidates selected for this comparison in each category are:

- CAPTools [\[20\]](#), a computer aided parallelization toolkit that translates sequential FORTRAN programs into message-passing SPMD codes;
- Portland Group’s High Performance FORTRAN compiler [\[21\]](#); and
- the MIPSPro FORTRAN compiler available on the Origin2000, which supports shared memory multiprocessing directives and MP runtime library.

In this study, the three approaches were evaluated in four areas: 1) user interaction, 2) limitations, 3) portability and 4) performance. Earlier studies (e.g., [\[22, 23\]](#)) that compared automated parallelization tools, data parallel languages, and hand-written message-passing codes emphasized performance results and found that the message passing code performed the best. Our study, on the other hand, considered performance as one of many criteria for success, which include level of user input to the tool, limitations of their functionality, and portability of generated code for real

applications. Given the diversity of parallel programming paradigms and computing platforms, application-interoperability across platforms presents a daunting task to tools and compiler developers. While the performance of parallelized code is critical, evaluating tools and compilers in terms of user interaction, limitations, and portability of parallelized code can provide valuable feedback to tool developers and users.

The outline of the paper is as follows: Section 2 provides information about the software, applications and computing platform used in our tests; Section 3 provides the comparison of the approaches; and Section 4 provides conclusions and future research.

2 Experimental Set-up

The SGI Origin2000, a distributed shared memory machine with 64 processors and 16GB of globally addressable memory, was used for this evaluation. The processors are arranged in pairs with 512MB local memory on each node. Each processor has a MIPS R10000 64-bit CPU (195 MHz) with two 32KB primary caches and one 4MB secondary cache. The programming environment included *IRIX 6.4*, SGI's native FORTRAN77 compiler *f77.7.2*, and Portland Group's HPF compiler *pghpf2.4*. This section describes, in detail, the three parallelizing tools and compilers tested and the test suite.

2.1 Parallelizing Tools/Compilers

2.1.1 CAPTools

Among the plethora of interactive parallelization tools and environments, three accept serial FORTRAN code without requiring data- or task-parallel annotations. These are FORGE Explorer DMP [4], FORESYS [24] and CAPTools [20]. FORGE Explorer DMP generates message-passing parallel code, but does not perform extensive interprocedural dependence analysis. Therefore, the user must meticulously partition data at every routine to generate consistent efficient parallel code. While FORESYS performs a superb job of "cleaning-up" ancient FORTRAN syntax, its parallelization ability is limited to F90 transformations. In contrast, CAPTools performs an extensive interprocedural dependence analysis and combines its results with user directions to automatically partition data across relevant routines for the entire program. For this reason, we chose to use CAPTools for our study.

More specifically, CAPTools performs interactive parallelization of serial code using graphical interfaces. Version 1.3 beta was used for this study. CAPTools takes as input sequential FORTRAN 77 code and generates parallel SPMD message-passing code as output. Communication within the parallel code is performed via calls to a communication library. Versions of this library, which are based on libraries such as MPI and PVM, are available for many different HPC platforms. The user must guide the parallelization process by deciding on the level of dependence analysis, the data partitioning strategy, etc. This user interaction will be described in detail in Section 3.1.

2.1.2 PGHPF Compiler

The data parallel model of computations stipulates that calculations are performed concurrently with data distributed across processors. Each processor processes the segment of data it owns. Data parallel languages and compilers implement this model by directives and statements included in the program; the user does not need to explicitly partition the data as with message-passing. Several tools [25] and compilers support HPF [6]; these include: *Digital Fortran 5.1* from DEC, *xlhpf* from IBM, *VAST-HPF* from Pacific-Sierra Research Corp., and *pgHPF* from Portland Group Inc. *pgHPF* was selected for this comparison because of its portability across hardware platforms and its availability on the test platform. The compiler provides feedback to help the user to tune a code. The information provided includes: parallelizability of loops, reasons why a loop was not parallelized, and warnings on expensive data redistribution. The tuning process involves changing data-distribution to reduce amount of communication, privatizing variables and arrays in independent loops and interchanging loops for improving cache performance.

2.1.3 SGI Origin2000 Compiler Directives

The final approach we evaluated is the combination of user directives and parallelizing compiler tools available on the SGI Origin2000. Directives are inserted in the sequential code to parallelize loops that do not have data dependencies across iterations. Various iterations of parallelized loops are scheduled at different processors and executed in a fork-and-join fashion by MP runtime library on the Origin2000. A master thread initiates the program, creates multiple slave threads, schedules the iterations of parallelized loops on all the threads including itself, waits for the completion of a parallel loop by all the slave threads, and executes sequential portions of the program. Slave threads wait for work (i.e., for parts of subsequent parallel loops) when the master thread executes a sequential portion of the code. As in case of HPF, this model of parallelism is also implemented by compiler directives. While HPF uses directives as hints for the compiler to exploit data parallelism, shared memory multiprocessing directives simply distribute computation within loops without explicitly dealing with data locality. No explicit message-passing is needed because all memory references are in a global address space where consistency is guaranteed by the hardware.

Two native tools are available on Origin2000 to assist the development of a directives-based shared memory program:

- Power Fortran Accelerator (PFA), which can automatically (i.e., with no user input) insert parallelization directives in sequential code and transform the loops to enhance their performance; and
- Parallel Analyzer View (PAV), which can annotate the results of dependence analysis of PFA and present them graphically.

Based on the analysis presented by PAV, users can manually insert directives to parallelize additional loops not parallelized by PFA (due either to its conservative dependence analysis or lack of deep interprocedural analysis support). Users can assist the compiler to improve data locality by

using data distribution directives and/or setting environment variables for different types of data placement.

Shared memory systems are often supplied with compilers that support their customized directives. The emerging OpenMP standard [19] is an effort to allow the directives-based code to be portable across different shared memory systems. SGI compiler supports both its customized directives as well as those outlined in the OpenMP standard.

2.2 Test Applications

In this study, we characterize the performance of the parallelization tools using NAS Parallel Benchmarks (NPB) and a simple aerospace application, ARC3D.

2.2.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB's) [17, 26] were derived from CFD codes. They were designed to compare the performance of highly parallel computers and are widely recognized as a standard indicator of computer performance. NPB 2.3 [27] consists of five kernels and three simulated CFD applications derived from important classes of aerophysics applications. These 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. Six of the benchmarks — LU, SP, BT, FT, MG and CG — were used for this study:

- **LU** is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven block diagonal system. This system was obtained from finite-difference discretization of the Navier-Stokes equations in 3D, by splitting it into block lower- and upper triangular systems.
- **SP** is a simulated CFD application that uses an implicit algorithm to solve the 3D compressible Navier-Stokes equations. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has scalar pentadiagonal bands of linear equations that are solved using Gaussian elimination without pivoting. Within the algorithm, this solution is performed by the ADI solver (alternating direction implicit) which solves the three sets of systems of equations sequentially along each dimension.
- **BT** is a simulated CFD application that solves systems of equations resulting from approximately factored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions. The BT code solves block-tridiagonal systems of 5×5 blocks, but the solution algorithm has the same structure as SP.
- **FT** contains the computational kernel of a three-dimensional Fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional FFT's, one for each dimension.
- **MG** uses a multi-grid method to compute the solution of the three-dimensional scalar Poisson equation. It requires highly structured long distance communication and tests both short and long distance data movement. The algorithm solves for the solution over the grid

of data points that is recursively made coarse and then fine again. This changing grid size requires that the arrays in the code are indirectly accessed.

- **CG** uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel tests unstructured grid computations and communications by using a sparse, unstructured matrix with randomly generated locations of entries. CG requires indirect array elements accesses.

2.2.2 ARC3D

In addition to using the benchmarks, a moderate size CFD application, ARC3D [18] was also tested. It solves Euler and Navier-Stokes equations in three dimensions using a single rectilinear grid. A Beam-Warming algorithm is used to approximately factorize an implicit scheme of finite difference equations, which is then solved alternatively in three directions. The implemented Alternating Direction Implicit (ADI) solver sweeps through each of the directions one at a time, with partial updating of the fields after each sweep. The solver of ARC3D is similar in structure to SP, but is a more realistic application. It includes a curvilinear coordinate system, turbulent model and more realistic boundary conditions.

2.2.3 Test Suite Summary

As shown in Table 1, four major dimensions are critical to parallelizing the NAS benchmarks and ARC3D application: grid structure, array accessing, data partitioning and pipelining.

Table 1: Summary of Parallelization Requirements for NAS Benchmarks

Benchmark/ Application	Grid		Array Access		Partitioning		Pipelining
	Struct.	Unstruct.	Direct	Indirect	Static	Dynamic	
SP	√		√		√		√
LU	√		√		√		√
BT	√		√		√		√
MG	√			√	√		
CG		√		√	√		
FT	√		√			√	
ARC3D	√		√		√		√

With the exception of CG, all the benchmarks represent computations on a structured mesh. The unstructured mesh in CG also requires indirect array accessing. Although MG uses a structured mesh, it still requires the use of indexed arrays because the granularity of the mesh changes throughout the course of the algorithm. Furthermore, FT requires dynamic repartitioning (data redistribution) of the arrays between the solution of FFT in each dimension. Without the

repartitioning, the amount of communication is prohibitively large. Finally, BT, SP, LU and ARC3D require pipelined computation. This analysis indicates the requirements of a parallelization tool for aerospace applications. An ideal parallelization tool/compiler would be able to handle all cases effectively.

3 Comparison of Parallelization Approaches

In this section, we present a comparison of three selected approaches using experimental setup described in Section 2. These approaches were evaluated from four perspectives: 1) user interaction, 2) limitations, 3) portability and 4) performance.

3.1 User Interaction

The type of user interaction required by each of the three approaches is different. We parallelize a simple implementation of the Jacobi method, given in Figure 1, to illustrate these differences. Loops labeled [10](#), [20](#), and [30](#) can be computed in parallel, with the recognition of a reduction operation in the third loop. Loop [40](#) is a sequential “while” loop. If parallelized by hand using MPI message-passing, the parallelized code will appear similar to that listed in Figure 2. In addition to computing these loops in parallel, two sets of SEND/RECEIVE are needed to update boundary values of T for each iteration.

```
PROGRAM JACOB
REAL T(1000),TNEW(1000),TOL
INTEGER N
PRINT*,'Enter the values of N and TOL ...'
READ*,N,TOL
DO 10 I=1,N
    T(I)=0.0
10 CONTINUE
40 CONTINUE
DO 20 I=2,N-1
    TNEW(I)=(T(I-1)+T(I+1))/2.0
20 CONTINUE
TNEW(1)=T(2)/2.0
TNEW(N)=(T(N-1)+100.0)/2.0
DIFMAX=0.0
DO 30 I=1,N
    DIFF=ABS(TNEW(I)-T(I))
    IF (DIFF.GT.DIFMAX)DIFMAX=DIFF
    T(I)=TNEW(I)
30 CONTINUE
IF (DIFMAX.GT.TOL) GOTO 40
WRITE(*,*)DIFMAX
END
```

Figure 1: Jacobi Serial Code

```

PROGRAM JACOB
  INCLUDE 'mpi.h'
  REAL T(1001),TNEW(1000),TOL
  REAL DIFF, DIFMAX, GLOB_DIFMAX
  INTEGER N, NUM_PROC, NODE, ERROR, I_LOW, I_HIGH

  CALL MPI_INIT(ERROR)

  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NUM_PROC, ERROR)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, NODE, ERROR)

  IF (NODE.EQ.0) THEN
    PRINT*,'Enter the values of N and TOL ...'
    READ*,N,TOL
  ENDIF

  CALL MPI_BCAST(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ERROR)
  CALL MPI_BCAST(TOL, 1, MPI_REAL, 0, MPI_COMM_WORLD, ERROR)
  I_LOW = (N/NUM_PROC)*(NODE) + 1
  I_HIGH = I_LOW + (N/NUM_PROC) - 1
  DO 10 I=I_LOW, I_HIGH
    T(I)=0.0
10  CONTINUE

  40  CONTINUE
    CALL MPI_SEND(T(I_LOW), 1, MPI_REAL, NODE -1, 99,
+ MPI_COMM_WORLD, ERROR)
    CALL MPI_SEND(T(I_HIGH), 1, MPI_REAL, NODE +1, 99,
+ MPI_COMM_WORLD, ERROR)
    CALL MPI_RECV(T(I_LOW-1), 1, MPI_REAL, NODE-1, 99,
+ MPI_COMM_WORLD, ERROR)
    CALL MPI_RECV(T(I_HIGH+1), 1, MPI_REAL, NODE+1, 99,
+ MPI_COMM_WORLD, ERROR)
    DO 20 I=MAX(2, I_LOW), MIN(N-1, I_HIGH)
      TNEW(I)=(T(I-1)+T(I+1))/2.0
20  CONTINUE
    IF ((1.GE.I_LOW).AND.(I.LE.I_HIGH))
+ TNEW(1) = T(2)/2.0
    IF ((N.GE.I_LOW).AND.(N.LE.I_HIGH))
+ TNEW(N)=(T(N-1)+100.0)/2.0
    DIFMAX=0.0
    DO 30 I=I_LOW, I_HIGH
      DIFF=ABS(TNEW(I)-T(I))
      IF (DIFF.GT.DIFMAX)DIFMAX=DIFF
      T(I)=TNEW(I)
30  CONTINUE
    CALL MPI_REDUCE(DIFMAX, GLOB_DIFMAX, 1, MPI_REAL, MPI_MAX, 0,
+ MPI_COMM_WORLD, ERROR)
    IF (GLOB_DIFMAX.GT.TOL) GOTO 40
    WRITE(*,*) GLOB_DIFMAX
    CALL MPI_FINALIZE(ERROR)
  END

```

Figure 2: MPI Version of the Jacobi Code

3.1.1 CAPTools

CAPTools first parses the serial code and performs an interprocedural dependence analysis before proceeding to the interactive domain decomposition stage. The user has the option to control the thoroughness of the analysis and supply knowledge about input parameter values to assist in the analysis. For example, the user may state that N , an input parameter shown in Figure 1, is positive and “non-trivial” (value is always bigger than a low threshold). This information can potentially reduce analysis time and allows more parallelism in the code to be detected.

After the analysis, an array is then selected to be partitioned through the partition browser as shown in Figure 3. CAPTools will propagate this action across subroutine boundaries, making use of the result obtained in the dependence analysis. If necessary, additional arrays can be partitioned. For the example shown in Figure 1, it is sufficient to partition array T and CAPTools propagates the partitioning to array $TNEW$. In the next few steps, execution control masks are then inserted to ensure that loop bounds are set appropriately to follow the “owner computes” rule. Communication required to exchange data along partition boundaries is then generated and optimized to reduce latencies and overhead. At each stage of this interactive parallelization process, a set of “browsers” provides feedback to the user. In many cases, efficient parallel code cannot be generated without user supplied knowledge about the code, specific data partitioning and parallelization strategies.

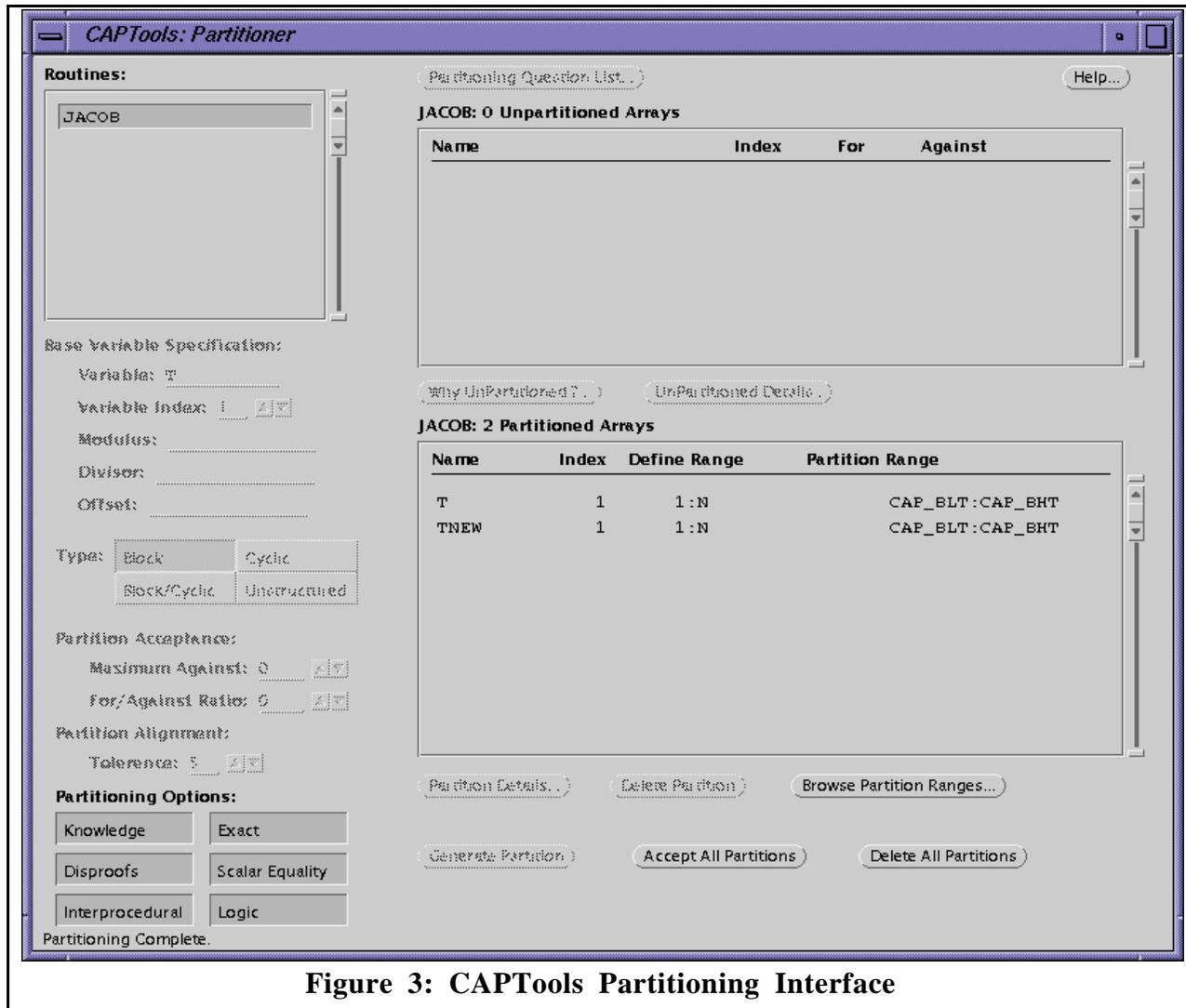


Figure 3: CAPTools Partitioning Interface

The generated parallel code, shown in Figure 4, employs calls to CAPTools' library to perform initialization (e.g., CAP_SETUPPART) and communication (e.g., CAP_SEND, CAP_RECEIVE). One can readily recognize the similarity of Figure 4 with Figure 2. For example, the communications of boundary values of T was implemented via two CAP_EXCHANGE calls, and the reduction operation for loop 30 was performed by CAP_COMMUNICATE.

```

PROGRAM PARALLELJACOB
INTEGER CAP_LEFT,CAP_RIGHT
PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
REAL T(1000),TNEW(1000),TOL
INTEGER N
INTEGER CAP_BLT,CAP_BHT
COMMON /CAP_RANGE/CAP_BLT,CAP_BHT
INTEGER CAP_ICOUNT
EXTERNAL CAP_RMAX
REAL CAP_RMAX
INTEGER CAP_PROCNUM,CAP_NPROC
COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
CALL CAP_INIT
IF (CAP_PROCNUM.EQ.1)PRINT*,'Enter the values of N and TOL ...'
IF (CAP_PROCNUM.EQ.1)READ*,N,TOL
CALL CAP_RECEIVE(N,1,1,CAP_LEFT)
CALL CAP_SEND(N,1,1,CAP_RIGHT)
CALL CAP_RECEIVE(TOL,1,2,CAP_LEFT)
CALL CAP_SEND(TOL,1,2,CAP_RIGHT)
CALL CAP_SETUPPART(1,N,CAP_BLT,CAP_BHT)
DO 10 I=MAX(1,CAP_BLT),MIN(N,CAP_BHT),1
  T(I)=0.0
10 CONTINUE
40 CONTINUE

CALL CAP_EXCHANGE(T(CAP_BHT+1),T(CAP_BLT),1,2,CAP_RIGHT)
CALL CAP_EXCHANGE(T(CAP_BLT-1),T(CAP_BHT),1,2,CAP_LEFT)
DO 20 I=MAX(2,CAP_BLT),MIN(N-1,CAP_BHT),1
  TNEW(I)=(T(I-1)+T(I+1))/2.0
20 CONTINUE
IF ((1.LE.CAP_BHT).AND.(1.GE.CAP_BLT)) THEN
  TNEW(1)=T(2)/2.0
ENDIF
IF ((N.LE.CAP_BHT).AND.(N.GE.CAP_BLT)) THEN
  TNEW(N)=(T(N-1)+100.0)/2.0
ENDIF
DIFMAX=0.0
DO 30 I=MAX(1,CAP_BLT),MIN(N,CAP_BHT),1
  DIFF=ABS(TNEW(I)-T(I))
  IF (DIFF.GT.DIFMAX) THEN
    DIFMAX=DIFF
  ENDIF
  T(I)=TNEW(I)
30 CONTINUE
CALL CAP_COMMUTATIVE(DIFMAX,2,CAP_RMAX)
IF (DIFMAX.GT.TOL) THEN
  GOTO 40
ENDIF
IF (CAP_PROCNUM.EQ.1)WRITE(*,*) DIFMAX
CALL CAP_FINISH()
END

```

Figure 4: CAPTools Generated Parallel Code

With the test suite (NPB's and ARC3D), more work was required to generate codes with good performance. For instance, in the case with LU, BT and SP, some distributed arrays were "local" — none of their boundary elements needed updates from other processors. Therefore, it is more efficient to calculate these values than to communicate them. To correct this, the user can remove the local arrays from the "partitioned array list" in the CAPTools partitioning interface before code generation. This type of tuning greatly improves performance; for example, the execution time of LU on four processors was reduced by 38%. Other possible tuning optimizations involve choosing different available options for parallelizing the code. The use of buffered communication, for example, improved the performance of LU on four processors by 8%.

In ARC3D, however, we had to rewrite the serial code to obtain good performance from CAPTools. The solvers used for the x-, y- and z-directions were originally written to solve for grid points on one two-dimensional plane at a time. When solving for the direction with partitioned data, the computation is pipelined across the processors for each two-dimensional plane being solved. This resulted in many small pipelines, each incurring idle time. In order to reduce this idle time, the computation was regrouped so that the three-dimensional grid points were solved at once. This resulted in one large pipeline and significantly reduced idle time.

In short, it took a few hours to parallelize the benchmarks and ARC3D. Some required a few more days of tuning. In comparison, months were required to write the parallel code by hand.

3.1.2 HPF

HPF relies on data distribution to achieve parallelism. After identifying the main arrays in the code, the user must align the arrays and distribute the main array. This step is similar to CAPTools except that the user must insert the data distribution directives by hand. The user can fine-tune the code by changing the distributions and privatizing arrays. When data distribution directives are introduced into the Jacobi example (Figure 1), the compiler can parallelize loops 10 and 20 automatically. Because the compiler reported that loop 30 used a global variable `DIFMAX`, the user introduced `DIFFarr` to keep maximum locally, and used the intrinsic function `MAXLOC` to derive `DIFMAX` from `DIFFarr` (see Figure 5). This requires a significant level of understanding of the program. Inserting these directives and loop modifications into the serial code by hand can be quite time consuming. Therefore, generating an initial parallel code using HPF takes longer than all other approaches considered in this study. The tuning of HPF code involves redistribution, privatizing and loop interchanges based on compiler feedback and profiling (e.g. *pgprof* that came with *pghp*). The HPF program development cycle required a few weeks per benchmark.

```

PROGRAM JACOB
REAL T(1000),TNEW(1000),TOL
REAL diffarr(1000)
integer maxposition(1)
!hpf$ align (:) with T :: TNEW, diffarr,maxposition
!hpf$ distribute (BLOCK) :: T
INTEGER N
PRINT*,'Enter the values of N and TOL ...'
READ*,N,TOL
DO 10 I=1,N
    T(I)=0.0
10 CONTINUE
40 CONTINUE
DO 20 I=2,N-1
    TNEW(I)=(T(I-1)+T(I+1))/2.0
20 CONTINUE
TNEW(1)=T(2)/2.0
TNEW(N)=(T(N-1)+100.0)/2.0
DIFMAX=0.0
DO 30 I=1,N
    DIFFarr(I)=ABS(TNEW(I)-T(I))
    T(I)=TNEW(I)
30 CONTINUE
maxposition = MAXLOC(DIFFarr(:))
i = maxposition(1)
DIFMAX = DIFFarr(i)
IF (DIFMAX.GT.TOL) GOTO 40
WRITE(*,*)DIFMAX
END

```

Figure 5: HPF Parallel Code

3.1.3 SGI Origin2000 Compiler Directives

Loop parallelization directives (e.g., DOACROSS) are added to the beginning of parallelizable loops. After the Power FORTRAN Accelerator (PFA) analyzed and automatically inserted directives into the serial Jacobi code (Figure 1), the Parallel Analyzer View (PAV) was used to present the results of dependence analysis and parallelization. As shown in Figure 6, an icon indicates the parallelizability of each loop. Clicking on a specific loop provides results from the dependence analysis. If a loop is found to be unparallelizable, a brief explanation is also provided.

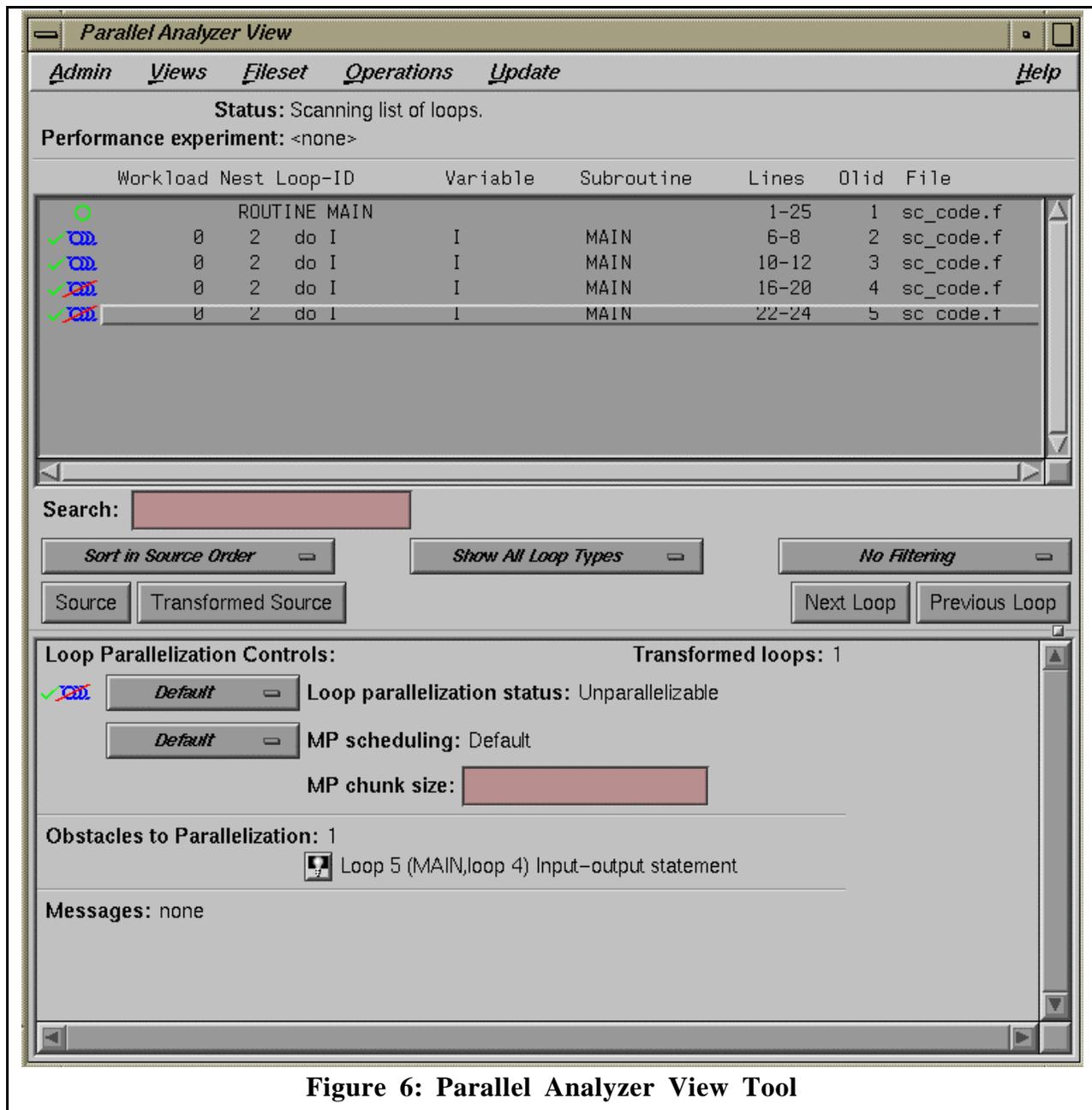


Figure 6: Parallel Analyzer View Tool

The automatically (PFA-based) parallelized code is given in Figure 7. PFA identified two parallelizable loops (commented as loops [2](#) and [3](#)) and inserted DOACROSS directives to parallelize them. Data structures within a parallelized loop are either marked as “local” to each processor or “shared” among all processors executing a subset of iterations. Because PFA could not identify the reduction in loop [4](#), it was not parallelized. This loop can be parallelized manually via the insertion of a DOACROSS directive and the declaration of DIFMAX as a reduction variable.

While generating an initial parallel code using shared memory multiprocessing directives takes only a few minutes, tuning usually takes several weeks. The results presented by PAV help focus code

modification efforts to specific unparallelized loops. Additionally, data distribution directives can be manually inserted in an effort to co-locate the data and computation [28].

```

C$SGI start 1
PROGRAM MAIN
IMPLICIT NONE
INTEGER*4 I, N
REAL*4 T(1002_8), TNEW(1000_8), TOL, DIFMAX, DIFF
LOGICAL*4 tmp0
PRINT *, 'Enter the values of N and TOL ...'
READ *, N, TOL
C$SGI start 2
C$DOACROSS local(I), shared(N, T)
DO I = 1, N, 1
    T(I + 1) = 0.0
END DO
C$SGI end 2
tmp0 = .TRUE.
DO WHILE(tmp0)
C$SGI start 3
C$DOACROSS local(I), shared(T, N, TNEW)
DO I = 2, (N + -1), 1
    TNEW(I) = ((T(I) + T(I + 2))) * 5.0E-01)
END DO
C$SGI end 3
TNEW(1) = (T(3) * 5.0E-01)
TNEW(N) = ((T(N) + 1.0E+02)) * 5.0E-01)
DIFMAX = 0.0
C$SGI start 4
DO I = 1, N, 1
    DIFF = ABS((TNEW(I) - T(I + 1)))
    IF(DIFMAX .LT. DIFF) THEN
        DIFMAX = DIFF
    ENDIF
    T(I + 1) = TNEW(I)
END DO
C$SGI end 4
tmp0 = TOL .LT. DIFMAX
END DO
WRITE(*, *) DIFMAX
STOP
END ! MAIN
C$SGI end 1

```

Figure 7: Directives Parallel Code (Using PFA)

Parallelization with compiler directives is, in a sense, “orthogonal” to the HPF style of parallelization because computation inside a loop is distributed based on the index range regardless of data location. Data locality is handled in the hardware that implements a cache coherent, non-uniform memory access architecture. The resulting global address space allows the user to avoid any explicit message-passing to access non-local memory. While developing a message-passing program requires the user to perform a domain-decomposition for the application, a directives-

based program is instead just an extension of its sequential counterpart. Since developing sequential code is a relatively simple process compared to developing parallel code, directives-based parallelism reduces the level of effort. Although it is convenient to leave the details of locality and consistency of data in the memory to hardware, additional effort is needed on the part of a user to eliminate excessive memory access overhead. Without this additional memory performance tuning effort, a directives-based program may not perform as well as a message-passing program.

In summary, while three approaches differ in the amount of time required to generate an initial code, they all required a few weeks to tune each code. In comparison, it took several months to generate parallel code by hand.

3.2 Limitations

As discussed in Section 2.2.3, we have a defined set of parallelization requirements for our test suite of representative CFD benchmarks and applications. Ideally, the tools should be able to handle all such requirements in order to be useful to us. Unfortunately, each parallelization approach has some limitations that may impact the level of effort for parallelizing code or achieving high performance. In this subsection, we focus on the limitations of our selected parallelization techniques as well as tools and compilers used to apply these techniques.

The version of CAPTools that we tested does not fully handle serial code with unstructured meshes and indirect array accesses. Therefore, CAPTools was unable to parallelize CG and MG. We do note that the limitations for CG stem from the NPB implementation, not from the conjugate gradient method itself. Nevertheless, since unstructured meshes are often used in large CFD applications, this constraint may restrict the use of CAPTools to applications with structured grids only. Furthermore, CAPTools requires that the data distribution remain static during the course of application execution. For this reason, CAPTools cannot effectively parallelize FT. The most efficient way to parallelize the code is to transpose the data so that the FFT calculation in FT is never performed over a partitioned dimension.

HPF requires that loops that are executed in parallel exhibit no dependencies between processors' sections of data. This means that the pipelined computation in SP, BT, LU and ARC3D cannot be expressed as parallel loops in HPF. For this reason, different algorithms must be used to get around this constraint for the benchmarks. To parallelize LU, a hyper-plane algorithm [29] was used for the SSOR solver. This algorithm does not scale as well as the pipelining algorithm used in the CAPTools and hand-coded versions. For ARC3D, SP and BT, the data must be redistributed during the course of the algorithm so that the pipelined computation's data dependencies are contained within the processor. The data is first distributed in one dimension: the z dimension. Then, the computation (Gaussian Elimination) is performed in the x and y dimensions on each processor without any need for communication. Before the computation is performed in the z dimension, the data has to be redistributed; otherwise, HPF could not perform the pipelined computation in parallel for the partitioned z dimension. This redistribution is

accomplished by transposing the data so that it is partitioned in the x dimension. The transposition requires an expensive all-to-all communication, but no subsequent communication is required for the computation in the z dimension. The data transposition results in code that does not scale as well as one with pipelined communication, which is used in CAPTools and the hand-coded version, and requires allocation of arrays with an alternative distribution.

The Origin2000 compiler directives can be used to parallelize all the test codes. For example, directives successfully parallelized applications that use unstructured meshes or require indirect array accesses. However, compiler directives are often not effective in cases where loop-carried dependencies are encountered. For a given loop nest, where the outer-most loop is dependent across iterations but an inner loop is not, it may be possible to transform the loop nest to make the independent loop index as the outer-most index. Parallelization of this modified loop nest will exhibit better performance because of increased granularity. This technique was used for several loop nests in BT and SP. In cases where a loop nest cannot be transformed, the inner loop has to be parallelized with lower granularity. Such fine-grained parallelization results in higher overhead to synchronize threads at the end of each parallel loop. In other cases, where data dependencies may be present in multiple dimensions and no loop index that is independent across iterations can be found, the algorithm may have to be changed as a final resort. One such example involved LU, where main parts of the code (upper and lower triangulation procedures) exhibited dependencies in all dimensions. Similar to HPF case, a modified LU algorithm that employs a hyperplane and restricts the loop-carried dependencies to one dimension has to be implemented. Inner loops were parallelized in this modified code but the scalability was not as high as that of message-passing code that used a pipelined communication. In addition to limitations due to of loop structure, automatic tools like PFA have their own limitations as well. For instance, PFA's dependence analysis could not recognize the reduction in one of the loops for our example Jacobi code. It was also the case with PFA-parallelized implementation of ARC3D. Additionally, due to the lack of interprocedural analysis support in PFA, it was unable to parallelize any loop that contains a procedure call. This created problems with BT, FT, and ARC3D. In cases where a "called procedure" in an otherwise parallelizable loop was independent of other loop iterations, appropriate parallelization directives were inserted by hand to improve the performance.

3.3 Portability

Since an important goal of this effort involves identifying alternatives to rewrite code as machines evolve. CAPTools' communication is performed by calls to a library. This library provides a machine-independent layer by relying on MPI, PVM or machine-specific communication calls or directives. Porting this library to different machines is relatively easy. One of the purported strengths of HPF is that it is based on a standard and consequently portable. Currently, there is an HPF compiler for most major high performance systems, but whether this will still be applicable to computational grids [\[1, 2\]](#) involving multiple levels of heterogeneous multiprocessors, remains an open question. Finally, the compiler directives on the Origin2000 are not portable. However,

directives based on the OpenMP [19] standard, provide a promising development path for future share-memory multiprocessors.

3.4 Performance

Figure 8 compares the execution time of the three automated approaches (CAPTools, PFA-dir and HPF) to the performance of the hand-coded benchmarks (NPB-2.3-MPI).

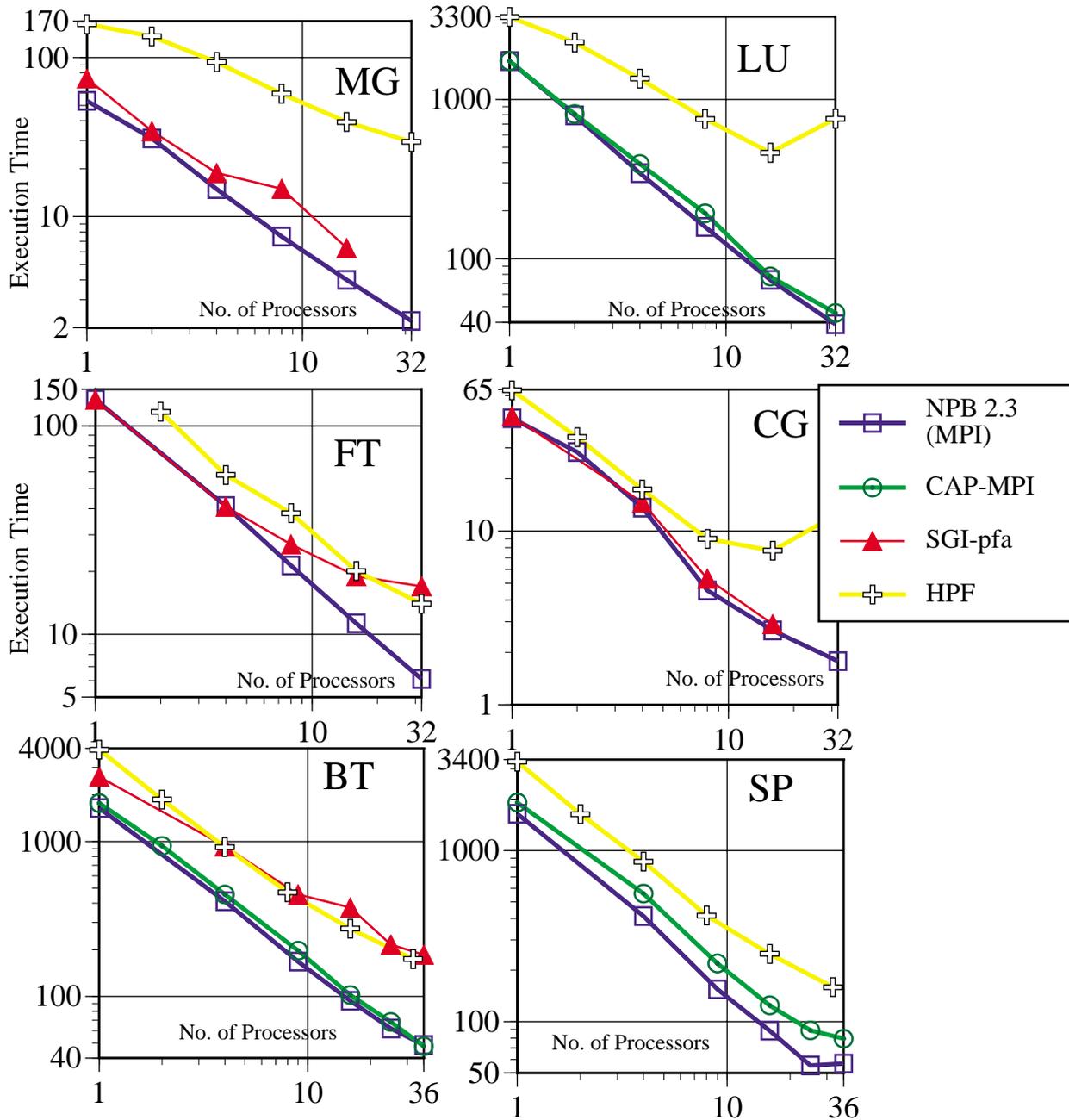


Figure 8: Performance Comparison for 6 Benchmarks

Figure 9a compares the execution time of automated approaches for ARC3D. These comparisons are presented using log-log plots to emphasize relatively small differences that cannot be appreciated on regular plots.

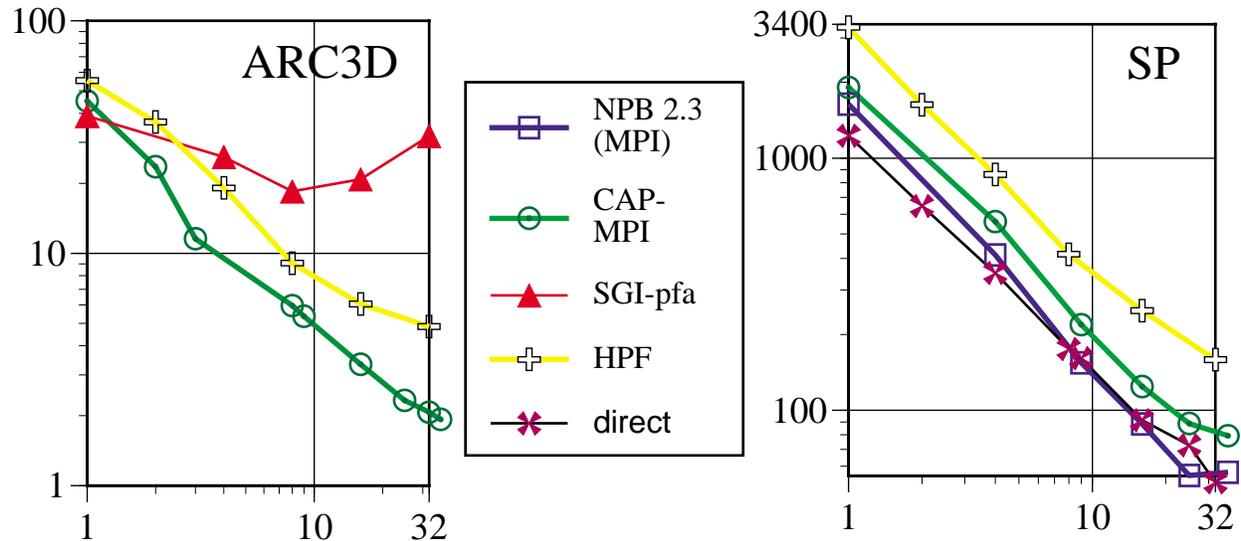


Figure 9: Performance of (a) ARC3D and (b) Hand-optimizing Directives-based SP

In general, we note the following trends:

- CAPTools consistently produces code that out-performs other two automated approaches for LU, BT, SP, and ARC3D. Nevertheless, it is unable to parallelize FT, CG and MG.
- The HPF code has the worst performance for almost all cases. Its performance on a single processor is over twice as slow as the other approaches, signifying that the problem is the compiler and overhead of processing HPF directives-related code. Furthermore, because it must use data redistribution for BT, SP and ARC3D instead of parallel pipelined computation and communication, the performance is worse than the other versions.
- The Origin2000 compiled code has good performance in some cases. For BT, we used the automated tool and did some manual tuning to force parallelization of the loops and employed data distribution directives to co-locate computation and data as much as possible. The performance of the resulting code is reasonable. However for SP, using PFA alone produced poor performing code because: (1) the dependence analysis failed to parallelize some large loops with complex code structures; and (2) cache utilization was low due to lack of single processor tuning of code. Several loops were modified to privatize temporary arrays and cache utilization was improved by properly padding arrays. Because of this tuning, the performance was better than the hand-coded NPB2.3 as shown in Figure 9b. For LU, the automated tool was unable to parallelize the code.
- The good performance of the cache-optimized code in Figure 9b emphasizes the importance of serial optimizations. While the results shown are for SP, they are indicative of the types of performance gains that were observed for all the benchmarks. The original serial test

codes were not written with cache performance in mind; therefore, their performance, both serial and parallel on the Origin2000 is not as good as it could be. For BT, the original serial code has a number of global arrays with power-of-two dimensions. Since cache-line sizes are also power-of-two, such arrays result in several unnecessary cache conflict-misses. A common technique to avoid such misses involves padding the array dimensions, so that accesses to different array elements are less likely to compete for the same cache line. Array padding alone reduced the sequential execution time by half for BT. Additional cache optimization techniques, such as array privatization, removing unnecessary temporaries, loop transformations, and code modifications to improve cache line reuse, are also relevant for shared memory systems developed with cache-based processors. Work is in progress to incorporate these cache optimizations into the generated parallel codes.

3.5 Comparison Summary

Table 2 presents a concise comparison of the parallelization approaches we evaluated in our study. Message-passing codes produced by hand exhibit the highest performance because the user can utilize their knowledge about the code to formulate a suitable parallelization strategy and tailor the implementation to the architecture. The use of the HPF language reduces the user's effort by shifting the machine-dependent implementation details to compiler writers and library builders. Unfortunately, applications which do not fit into pre-defined computation models and templates offered in the language either cannot be implemented or must execute at a reduced level of performance. If portability were not an issue, the Origin 2000 compiler directives, combined with detailed profiling and user tuning, would be a good approach. However, the need to limit compile time reduces the thoroughness in which inter-procedural dependence analysis could be applied, thus affecting the quality and granularity of the parallel code produced for complex applications. Therefore, good performance is achieved only with extensive tuning, which is what we wished to avoid by using automated parallelization. Finally, CAPTools provides excellent performance for some of the benchmarks without using many extensive hand optimizations. Unfortunately, there are currently many limitations with this software that prevent it from being applicable for many CFD applications.

4 Conclusions and Future Research

Comparing CAPTools, HPF and the Origin compiler directives has provided some insight for automating parallelization of aerospace applications. While all of the approaches are considered “automated” they still require user input/tuning to generate reasonable code. In summary, HPF requires the most work for the least payoff in terms of performance. CAPTools requires reasonable user input and generates code with good performance, but it cannot currently handle some features that are integral to most CFD applications. Finally, the compiler directives can achieve great performance, but with a lot of effort and without a guarantee of portability.

Table 2. A Comparison of Parallelization Approaches based on NPB and ARC3D

Process	Time/Effort	Performance	Portability	Applicability/ Limitations
Rewritten by hand	Extensive code revision required; error-prone	Excellent when implementation is tailored to machine	Dependent on portability of standards (e.g. MPI, PVM); tuning required	Applicable to any code
SGI Origin2000 Compilers	Minimal code modification; directives inserted as needed; tuning could be time consuming	Completely dependent on compiler and user's expertise; can be excellent if optimized for caches	Dependent on portability of standards (e.g. OpenMP);	Performs well for codes with simple structure and loop-level parallelism
HPF	Annotation required; need code restructuring to match language's programming paradigm	Compiler performance still in question to-date	Dependent on portability of HPF standards	Does not handle computational pipelines efficiently and require additional redistributions
CAPTtools	User must choose arrays to partition; more tuning is needed for good performance	Excellent when user tunes the code.	CAPTtools library is based on standard MPI and PVM libraries.	Cannot handle unstructured grids, indirect array accesses and data redistribution.

This study applied parallelization tools and compilers to real applications on high performance parallel and distributed systems. While these tools and compilers have shown promise to assist the users in parallelizing and optimizing their applications for specific platforms, their feasibility for aerospace applications depends on three factors:

- Ability to minimize a user's time and effort to parallelize or port legacy code to a new platform or programming environment;
- extensibility to appropriately deal with the limitations due to domain-specific characteristics of application codes; and
- development and adoption of standards that ensure portability.

Our study of three such tools/compilers indicates a potential value-added for applying these tools to aerospace applications: they can reduce the time and effort by a significant amount. In addition, these tools and compilers have adopted existing standards, such as MPI, HPF, and OpenMP, to ensure portability. We believe that developers need to focus on ways to deal with the limitations of their tools and compilers for domain-specific applications. For instance, we were able to parallelize sequential code to message-passing and shared memory multiprocessing programs with

significantly less effort compared to hand-parallelization. However, in some cases, the parallelization tools either failed to deal with a particular type of code or introduced additional performance bottlenecks due to the architecture of target platform. Feasibility and value of these computer-aided parallelization tools for aerospace (and other) applications will be greatly enhanced if we develop extensible mechanisms to deal with domain-specific limitations.

While this study provides a comparison of different approaches, the preliminary results show that automated parallelization approaches do exhibit promise for our needs at NASA. In a relatively short amount of time, we were able to generate parallel codes with reasonably good performance. We are extending this study to incorporate the evaluation of other parallelization tools/compiler (e.g. *SUIF* [15] and the D System [7]; other architectures (CRAY T3E and network of workstations); and other more complex applications (e.g. OVERFLOW).

5 References

- [1] "The National Partnership for Advanced Computational Infrastructure", NPACI, <http://www.npaci.edu/Research>
- [2] NCSA, "Partnerships for Advanced Computational Infrastructure", PACI, <http://www.cise.nsf.gov/acir/paci.html>
- [3] "CF90 Commands and Directives Reference Manual," Cray Research, Inc. SR-3901.10, 1993.
- [4] "Forge Explorer User's Guide", Advanced Parallel Research, Inc., <http://www.qpsf.edu.au/workshop/forge/forge.html>
- [5] "KAP/Pro Toolset for OpenMP", KAI, Inc., <http://www.kai.com:80>
- [6] H. P. F. Forum, "High Performance FORTRAN Language Specification Version 1.0," Scientific Programming, vol. 2, 1993.
- [7] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. Reed, "An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs," presented at Supercomputing '95, San Diego, CA, 1995.
- [8] S. Benkner, "Vienna FORTRAN 90- An Advanced Data Parallel Language," presented at International Conference on Parallel Computing Technologies (PACT-95), St. Petersburg, Russia, 1995.
- [9] L. Snyder, "A ZPL Programming Guide," University of Washington January, 1998 1998.
- [10] D. Gannon, S. X. Yang, and P. Beckman, "User Guide for a Portable Parallel C++ Programming System pC++," Department of Computer Science and CICA Nov. 21, 1994 1994.
- [11] D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine, "HPC++ and HPC++ Lib Toolkit," Indiana University, Bloomington, IN 1997.

- [12] S. Balay, B. Gropp, L. C. McInnes, and B. Smith, "PETSc Library," . Argonne National Laboratory: <http://www.mcs.anl.gov/petsc/petsc.html>.
- [13] L. S. Blackford and e. al., "ScaLAPACK: a Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance," presented at Supercomputing '96, Pittsburg, PA, 1996.
- [14] 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.
- [15] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. Lam, and J. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," Computer Systems Laboratory, Stanford University, Stanford, CA.
- [16] M. Cross, C. S. Ierotheou, S. P. Johnson, P. Leggett, and E. Evans, "Software Tools for Automating the Parallelisation of FORTRAN Computational Mechanics Codes," Parallel and Distributed Processing for Computational Mechanics, 1997.
- [17] D. Bailey, T. Harris, W. Saphir, R. V. d. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," NASA Ames Research Center, Moffett Field, CA RNR-95-020, 1995.
- [18] T. H. Pulliam, "Solution Methods In Computational Fluid Dynamics," in Notes for the von Kármán Institute For Fluid Dynamics Lecture Series. Rhode-St-Genese, Belgium, 1986.
- [19] "OpenMP: A Proposed Standard API for Shared Memory Programming", <http://www.openmp.org/>
- [20] C. S. Ierotheou, S. P. Johnson, M. Cross, and P. F. Leggett, "Computer Aided Parallelisation Tools (CAPTools)-Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," Parallel Computing, vol. 22, pp. 163-195, 1996.
- [21] "PGHPF", Portland Group, <http://www.pgroup.com>
- [22] J.-I. Berthou and L. Colombet, "Which Approach to Parallelizing Scientific Codes-That is the Question," Parallel Computing, vol. 23, pp. 165-179, 1997.
- [23] C. Cléménçon, K. M. Decker, V. R. Deshpande, A. Endo, J. Fritscher, P. A. R. Lorenzo, N. Masuda, A. Müller, R. Rühl, W. Sawyer, B. J. N. Wylie, and F. Zimmerman, "Tools-Supported HPF and MPI Parallelization of the NAS Parallel Benchmarks," Swiss Center for Scientific Computing, Manno, Switzerland TR-96-02, March 1996.
- [24] "FORESYS, FORtran Engineering SYStem - Manuel de référence," Simulog, 1, rue James Joule 78286 Guyancourt.

- [25] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi, "High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes," presented at Supercomputing '98, Orlando, FL, 1998.
- [26] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," NASA Ames Research Center, Moffett Field, CA RNR-91-002, January 1991.
- [27] "NAS-Parallel Benchmarks 2.3", NAS Division, NASA Ames Research Center, <http://science.nas.nasa.gov/Software/NPB>
- [28] A. Waheed and J. Yan, "Parallelization of NAS Benchmarks for Shared Memory Multiprocessors," presented at HPCN '98, Amsterdam, 1998.
- [29] M. Frumkin, H. Jin, and J. Yan, "HPF Implementation of NPB2.3," presented at ISCA 11th International Conference on Parallel and Distributed Computing Systems, Chicago, IL, 1998.