The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance

H. Jin[*], M. Frumkin[*] and J. Yan
NAS Technical Report NAS-99-011 October 1999

{hjin,frumkin,yan}@nas.nasa.gov
NAS System Division
NASA Ames Research Center
Mail Stop T27A-2
Moffett Field, CA 94035-1000

Abstract

As the new ccNUMA architecture became popular in recent years, parallel programming with compiler directives on these machines has evolved to accommodate new needs. In this study, we examine the effectiveness of OpenMP directives for parallelizing the NAS Parallel Benchmarks. Implementation details will be discussed and performance will be compared with the MPI implementation. We have demonstrated that OpenMP can achieve very good results for parallelization on a shared memory system, but effective use of memory and cache is very important.

Keywords: NAS Parallel Benchmarks, OpenMP program, shared memory system, parallel program performance.

[*] MRJ Technology Solutions, NASA Contract NAS2-14303, Moffett Field, CA 94035-1000

The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance

H. Jin*, M. Frumkin* and J. Yan

NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000

Abstract

As the new ccNUMA architecture became popular in recent years, parallel programming with compiler directives on these machines has evolved to accommodate new needs. In this study, we examine the effectiveness of OpenMP directives for parallelizing the NAS Parallel Benchmarks. Implementation details will be discussed and performance will be compared with the MPI implementation. We have demonstrated that OpenMP can achieve very good results for parallelization on a shared memory system, but effective use of memory and cache is very important.

1. Introduction

Over the past decade, high performance computers based on commodity microprocessors have been introduced in rapid succession. These machines could be classified into two major categories: distributed memory (DMP) and shared memory (SMP) multiprocessing systems. While both categories consist of physically distributed memories, the programming model on an SMP presents a globally shared address space. Many SMP's further include hardware supported cache coherence protocols across the processors. The introduction of ccNUMA (cache coherent Non-Uniform Memory Access) architecture, such as SGI Origin 2000, promises performance scaling up to thousands of processors. While all these architectures support some form of message passing (e.g. MPI, the de facto standard today), the SMP's further support program annotation standards, or directives, that allow the user to supply information to the compiler to assist in code parallelization. Currently, there are two widely accepted standards for annotating programs for parallel executions: HPF and OpenMP. High Performance Fortran (HPF) [1] provides a data parallel model of computation for DMP systems. OpenMP [2], on the other hand, is a set of compiler directives that enhances loop-level parallelism for parallel programming on SMP systems.

OpenMP is, in a sense, "orthogonal" to the HPF type of parallelization because computation is distributed inside a loop based on the index range regardless of data location. Parallel

* MRJ Technology Solutions, NASA Contract NAS2-14303, Moffett Field, CA 94035-1000

programming with directives offers many advantages over programming with the message passing paradigm:

- simple to program, with incremental path to full parallelization;
- shared memory model, no need for explicit data distribution;
- scalability achieved by taking advantage of hardware cache coherence; and
- portability via standardization activities.

Perhaps the main disadvantage of programming with compiler directives is the implicit handling of global data layout. This, in the worst case, may create performance bottleneck and not be easily overcome. While some vendors have provided extensions to improve data locality with explicit data distributions and/or placement, these solutions are not portable and effective use of cache to get an efficient parallel program is still going to be an important issue.

In this study we will examine the effectiveness of OpenMP directives with the NAS Parallel Benchmarks (NPB) [3]. These benchmarks, extracted from a set of important aerospace applications, mimic a class of computation in computation fluid dynamics (CFD), and have been used to characterize high performance computers. Further description of the benchmarks will be given in section 3. The concepts of programming with a shared memory model and OpenMP directives are outlined in section 2. Detailed implementations of directive-based NPBs and their performance are discussed in section 4. The conclusion is in section 5.

2. SMP Architecture and OpenMP Directives

The use of globally addressable memory in shared memory architectures allows users to ignore the interconnection details of parallel machines and exploit parallelism with minimum grief. Insertion of compiler directives into a serial program is the most common and cost-effective way to generate a parallel program for the shared memory parallel machines.

OpenMP [2] is designed to support portable implementation of parallel programs for shared memory multiprocessor architectures. OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran, C and C++ to express shared memory parallelism. It provides an incremental path for parallel conversion of any existing software, as well as targeting at scalability and performance for a complete rewrite or an entirely new software.

A *fork-join* execution model is employed in OpenMP. A program written with OpenMP begins execution as a single process, called the *master thread*. The master thread executes sequentially until the first parallel construct is encountered (such as a “`PARALLEL`” and “`END PARALLEL`” pair). The master thread, then, creates a team of threads, including itself as part of the team. The statements enclosed in the parallel construct are executed in parallel by each thread in the team until a worksharing construct is encountered. The “`PARALLEL DO`” or “`DO`” directive is such a worksharing construct which distributes the workload of a `DO` loop among the members of the current team. An implied synchronization occurs at the end of the `DO` loop unless an "`END DO NOWAIT`" is specified. Data sharing of variables is specified at the start of parallel or worksharing 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.

OpenMP introduces a powerful concept of orphan directives that greatly simplify the task of implementing coarse grain parallel algorithms. Orphan directives are directives encountered outside the lexical extent of the parallel region. The concept allows user to specify control or synchronization from anywhere inside the parallel region, not just from the lexically contained region.

There are also tools available that allow the user to examine the correctness of an OpenMP program and profile the execution for tuning purpose [4].

3. NAS Parallel Benchmarks

3.1. Benchmark Description

NAS Parallel Benchmarks (NPB's) [1] were derived from CFD codes. They were designed to compare the performance of parallel computers and are widely recognized as a standard indicator of computer performance. NPB 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. The benchmarks are specified only algorithmically ("pencil and paper" specification) and referred to as NPB 1. Details of the NPB-1 suite can be found in [1], but for completeness of discussion we outline the seven benchmarks (except for the integer sort kernel, **IS**) that were implemented and examined with OpenMP in this study.

- **BT** is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are **B**lock-**T**ridiagonal of 5×5 blocks and are solved sequentially along each dimension.

- **SP** is a simulated CFD application that has a similar structure to BT. 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 **S**calar **P**entadiagonal bands of linear equations that are solved sequentially along each dimension.

- **LU** is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block **L**ower and **U**pper triangular systems.
- **FT** contains the computational kernel of a 3-D fast **F**ourier **T**ransform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFT's, one for each dimension.
- **MG** uses a V-cycle **M**ulti**G**rid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine. It tests both short and long distance data movement.
- **CG** uses a **C**onjugate **G**radient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries.
- **EP** is an **E**mbarrassingly **P**arallel benchmark. It generates pairs of Gaussian random deviates according to a specific scheme. The goal is to establish the reference point for peak performance of a given platform.

3.2. Source Code Implementations

Sample implementations of NPB 1 are referred to as NPB 2 [4]. The NPB-2 implementations were based on Fortran 77 (except for **IS**, which was written in C) and the MPI message passing standard. They were intended to approximate the performance a typical user can expect for a portable parallel program on a distributed memory computer. The latest release of NPB2.3 [4] also contains a serial version of the benchmarks (NPB2.3-serial), which is a stripped-down version of the MPI implementation. The serial version was intended to be used as a good starting point for automated parallel tools and compilers and for other types of parallel implementations.

3.3. Our Starting Point

The starting point for our study is the serial version of NPB2.3. Since the NPB2.3-serial is a stripped-down version of the MPI implementation, its structure was kept to be as close as possible to the MPI implementation and, thus, was not tuned in a *serial* sense. For example, a number of working arrays necessary for the MPI code can be reduced or eliminated in the serial code to improve the sequential performance. In an effort to reduce the noise in the results due to the presence of these "imperfections," we applied optimizations to the sequential codes, noticeably BT and SP, to improve implementation efficiencies and code organization that could also limit the parallelization based on the insertion of "directives." For example, BT and SP were reorganized so that memory requirements for Class A on one Origin 2000 node are reduced by 6 times and 2 times respectively. Execution speed has been improved significantly. For LU, we tested two types of implementations for parallelization: “pipeline”, which is used in the MPI version, and "hyperplane" (or "wavefront") that favors data-parallelism.

In the next section we first discuss the improvements that we applied to the serial version, noticeably BT and SP, then we describe the OpenMP implementations that are based on the improved serial versions. To avoid any confusion with the original NPB2.3-serial, we will refer to the modified and improved serial versions of benchmarks as the Programming Baseline for NPB (PBN). PBN is also the base for the HPF implementations of the NPB’s [6].

4. Implementation

4.1. Sequential Improvements

As mentioned above, during the process of parallelization with directives, we noticed that NPB2.3-serial contains redundant constructs left over from the MPI implementation. Several working arrays can also be used more efficiently in the serial version to improve the performance. In the following, we describe on the improvements applied to BT and SP since these changes are more significant and may have implication in real applications.

In the original version of BT in NPB2.3-serial, separate routines are used to form the left-hand side of the block tridiagonal systems before these systems are solved. Intermediate results are stored in a six-dimensional working array (`LHS`); two five-dimensional arrays are used as additional working space. (See next section for more details.) This is preferable for the multi-partition scheme [4] used in the parallel MPI implementation to achieve coarse-grained communication in the Gaussian elimination/substitution process. This has the drawback of using large amount of memory, thus, potentially increasing the memory traffic. For the directive version, the explicit communication is not a concern. So, before the parallelization of BT, we replaced these large working arrays with significantly smaller arrays to reduce the memory usage and improve the cache performance. This change also requires fusing several loops.

An example of the relevant change is given in Figure 1. On the left panel, the five-dimensional arrays `FJAC` and `NJAC` are assigned in the first loop nesting group and used in the second loop nesting group. After merging the two outside loop nests (J and K) of the two groups, the two working arrays (`FJAC` and `NJAC`) can be reduced to three-dimensional, as indicated on the right panel.

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 0, NX+1
      FJAC(*,*,I,J,K) = ...
      NJAC(*,*,I,J,K) = ...
    END DO
  END DO
END DO
DO K = 1, NZ
  DO J = 1, NY
    DO I = 1, NX
      LHS(*,*,1,I,J,K) <= FJAC(*,*,I-1,J,K)
                          NJAC(*,*,I-1,J,K)
      LHS(*,*,2,I,J,K) <= FJAC(*,*,I,J,K)
                          NJAC(*,*,I,J,K)
      LHS(*,*,3,I,J,K) <= FJAC(*,*,I+1,J,K)
                          NJAC(*,*,I+1,J,K)
    END DO
  END DO
END DO
```

```
DO K = 1, NZ
  DO J = 1, NY
    DO I = 0, NX+1
      FJAC(*,*,I) = ...
      NJAC(*,*,I) = ...
    END DO
    DO I = 1, NX
      LHS(*,*,1,I,J,K) <=
                 FJAC(*,*,I-1)
                 NJAC(*,*,I-1)
      LHS(*,*,2,I,J,K) <=
                 FJAC(*,*,I)
                 NJAC(*,*,I)
      LHS(*,*,3,I,J,K) <=
                 FJAC(*,*,I+1)
                 NJAC(*,*,I+1)
    END DO
  END DO
END DO
```

Figure 1: The left panel illustrates the use of working arrays in BT from NPB2.3-serial. The right panel shows the same code section but with the use of much smaller working arrays. The first two dimensions of `FJAC`, `NJAC` and `LHS` are used for 5×5 blocks.

The same optimization can be applied to the use of array `LHS`, which was reduced to four dimensions. The final memory-optimized version of BT uses only 1/6 of the memory needed by BT in NPB2.3-serial. The performance improvement of the new version is obvious: the serial execution time has been reduced by a factor of two on four different machines (see the summary in Table 1 and Figure 2). The timing profile of the key routines in BT is given in Table 2. As one can see, the memory optimization on BT has improved the performance of the three solvers (`X/Y/Z_SOLVE`) by about 50%.

Table 1: Performance improvements of the optimized BT and SP on a single node. The execution time is given in seconds and the MFLOP/sec numbers are included in parenthesis.

Processor Type	**Size**	**NPB2.3**	**Optimized**	**Change**
		BT		
Origin2000 (250MHz)	Class A	2162.4(77.82)	1075.2(156.51)	50.3%
T3E Alpha (300MHz)	Class W	218.1(35.39)	117.0(65.95)	46.4%
SGI R5000 (150MHz)	Class W	549.8(14.04)	265.0(29.13)	51.8%
PentiumPro (200MHz)	Class W	316.8(24.36)	121.2(63.69)	61.7%

	SP			
Origin2000 (250MHz)	Class A	1478.3(57.51)	971.4(87.52)	34.3%
T3E Alpha (300MHz)	Class A	3194.3(26.61)	1708.3(49.76)	46.5%
SGI R5000 (150MHz)	Class W	1324.2(10.70)	774.1(18.31)	41.5%
PentiumPro (200MHz)	Class W	758.9(18.68)	449.0(31.57)	40.8%

Table 2: Breakdown comparison of routines in the optimized and NPB2.3 BT. Timing was obtained on an Origin2000 node (195 MHz) for the Class A problem size. The relative percentage of timing is included in parenthesis.

Component	**Time (NPB2.3)**	**Time (Optimized)**	**Change**
RHS	232.87 (8.84%)	248.59 (16.98%)	(6.75%)
XSOLVE	722.91 (27.44%)	365.78 (24.98%)	49.40%
YSOLVE	793.75 (30.13%)	376.55 (25.72%)	52.56%
ZSOLVE	862.92 (32.76%)	463.74 (31.67%)	46.26%
ADD	21.72 (0.82%)	21.40 (1.46%)	1.47%
Total	2634.20 (100.0%)	1464.25 (100.0%)	44.41%

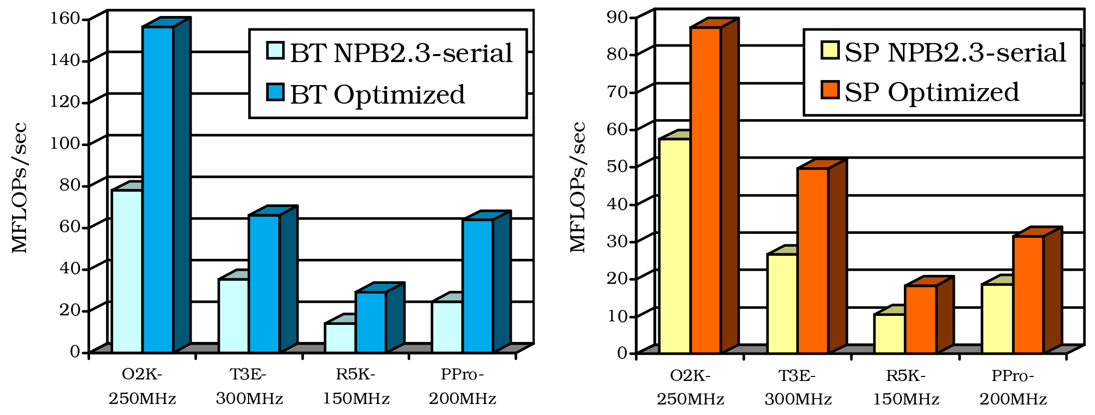

Figure 2: Comparison of serial performance of the optimized BT and SP with NPB2.3-serial.

A similar optimization, namely on the more effective use of working array `LHS`, has been applied to the serial version of SP. The execution times of the optimized SP and the version from NPB2.3 are compared in Table 1 and Figure 2. The improvement of execution time for SP is not as significant as that for BT because in SP, the temporary working arrays are smaller. The serial performance is still improved by about 40% on average.

4.2. Application Benchmark BT

The main iteration loop in BT contains the following steps:

```
DO STEP = 1, NITER
  CALL COMPUTE_RHS
  CALL X_SOLVE
  CALL Y_SOLVE
  CALL Z_SOLVE
  CALL ADD
END DO
```

The `RHS` array is first computed from the current solution (`COMPUTE_RHS`). Block tridiagonal systems are formed and solved for each direction of X, Y, and Z successively (`X_SOLVE`, `Y_SOLVE`, `Z_SOLVE`). The final solution is then updated (`ADD`).

The optimized serial version of BT is our starting point for the OpenMP implementation. There are several steps in the parallelization process:

1) Identify loops where different iterations can be executed independently (parallel loops).
2) Insert "`!$OMP PARALLEL DO`" directives for the outer-most parallel loops to ensure large granularity and small parallelization overhead. If several parallel loops can be grouped into a single parallel region, the "`!$OMP PARALLEL`" directive is used. This can potentially reduce the fork-and-join overhead.
3) List all the privatizable variables in the "`PRIVATE()`" constructs.
4) Touch the data pages by inserting initialization loops in the beginning of the program. On a cache coherent non-uniform memory architecture (ccNUMA) like the Origin 2000, a data page is owned by a processor that touches the data page first unless page migration is turned on. Such a scheme needs to be tested on other SMP systems.

The identification of parallel loops and privatizable variables can be assisted by computer-aided tools such as CAPO [8] (based on a parallelization tool kit, CAPTools [9], developed at the University of Greenwich). Additional parallelization involves reduction sums outside the iteration loop in computing solution errors and residuals, which are easily handled with the "`REDUCTION`" directive.

Timing profiles of the key code sections in the OpenMP BT were measured on the Origin2000. The results are shown in Figure 3. A major portion of the execution time is spent in the three solvers (`xsolve`, `ysolve`, `zsolve`), which scale fairly well (close to linear). The `zsolve` routine spends more time than `xsolve` and `ysolve` because data were touched initially in favor of the x and y solver. The worse scalability of `rhsz` can also be attributed to the same cause. The second-order stencil operation in `rhsz` uses the K±2, K±1 and K elements of the solution array to compute `RHS` for the z direction:

```
RHS(I,J,K) = A*U(I,J,K-2) + B*U(I,J,K-1) + C*U(I,J,K)
             + D*U(I,J,K+1) + E*U(I,J,K+2).
```

Such an operation accesses memory with long strides and is not cache friendly. One way to improve the performance is by first copying a slice of data in the K direction to a small 1-D working array, performing the stencil operation, then copying the result back to `RHS`. This technique, in fact, is used in the FT benchmark as described in a later section. The slightly worse performance of the "`ADD`" routine can be attributed to the small amount of calculation done inside the parallel loop and the parallization overhead of the "`PARALLEL DO`" directive seems to play a role. Overall, the directive-based BT performs very close to the MPI version (optimized for memory usage) although the latter still scales better (see section 0 for more discussion).

4.3. Application Benchmark SP

The iteration procedure of SP is very similar to BT although the approximate factorization is different. In one time iteration, the following steps are taken in SP:

```
CALL COMPUTE_RHS
CALL TXINVR
CALL X_SOLVE
CALL TXINVR
CALL Y_SOLVE
CALL TXINVR
CALL Z_SOLVE
CALL TZETAR
CALL ADD
```

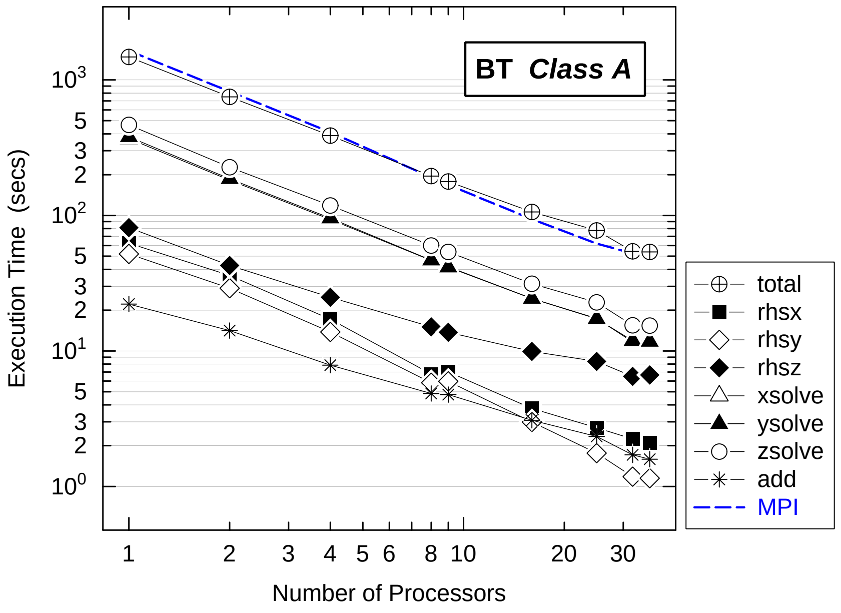

Figure 3: Execution profile of different components in OpenMP BT, measured on the Origin 2000. The total time from the MPI version is shown as a reference (same for other cases).

The optimized serial version of SP as described in section 4.1 was used for parallelization. The steps taken to parallelize SP were very similar to those for BT. The outer-most loops in each of these routines were parallelized with the "`PARALLEL DO`" directive. To further reduce the parallelization overhead, several end-of-loop synchronizations have been removed (mostly in `COMPUTE_RHS`) by the use of "`!$OMP END DO` **`NOWAIT`**", as was done in BT.

The profile of the directive-based parallel version of SP is shown in Figure 4. For the class A size, the overall performance of the OpenMP SP is better than the MPI version from NPB2.3 on less than 9 processors, but the MPI version scales better (see more discussion in section 4.8). Further analysis has indicated that the performance bottleneck of the directive version is from `rhsz` in `COMPUTE_RHS` although routine `ADD` did not scale as well. The situation is very similar to what was in BT (see section 4.3) where the second-order stencil operation on the K direction in `rhsz` has caused more caches misses and remote memory access than on the other two directions. This is also the reason for poor performance in `zsolve`, which runs about a factor of 2-3 times slower than `xsolve` and `ysolve` on 1 processor and 4 times slower on 16 processors.

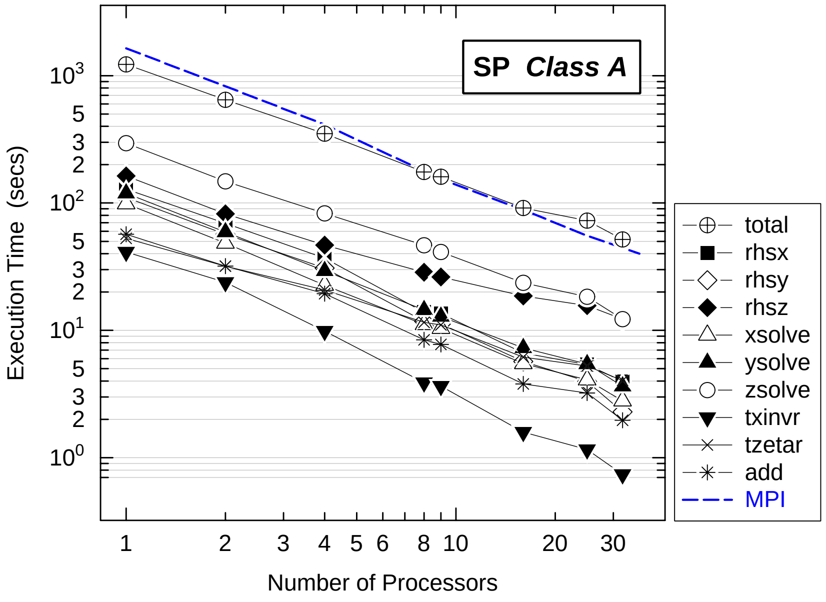

Figure 4: Execution profile of different components in the directive-based SP, measured on the Origin 2000.

4.4. Application Benchmark LU

LU factorizes the equation into lower and upper triangular systems. The systems are solved by the SSOR algorithm in the following iteration loop.

```
DO ISTEP=1,ITMAX
  CALL COMPUTE_RHS
  CALL JACLD
  CALL BLTS
  CALL JACU
  CALL BUTS
  CALL ADD
END DO
```

As in BT and SP, the RHS is first calculated. Then the lower-triangular and diagonal systems are formed (`JACLD`) and solved (`BLTS`), followed by the upper-triangular system (`JACU` and `BUTS`). The solution is lastly updated. In solving the triangular systems, the solution at (i,j,k) depends on those at ($i+e$,j,k), (i,$j+e$,k) and (i,j,$k+e$) where e=-1 for `BLTS` and e=1 for `BUTS`.

There are at least two methods to implement LU in parallel: hyperplane and pipelining. The *hyperplane* (or wavefront) algorithm exploits the fact that, for all the points on a given hyper-plane defined by $l = i+j+k$, calculations can be performed independently, provided the solution for $l+e$ is available. This is illustrated in the left panel of Figure 5 for a 2-D case. Bullets indicate points where solutions are already calculated and circles are points to be calculated. The calculations are performed along the diagonal. As we will see, the hyperplane algorithm does not utilize cache lines well for either column-major or row-major storage of data (see later discussion in this section).

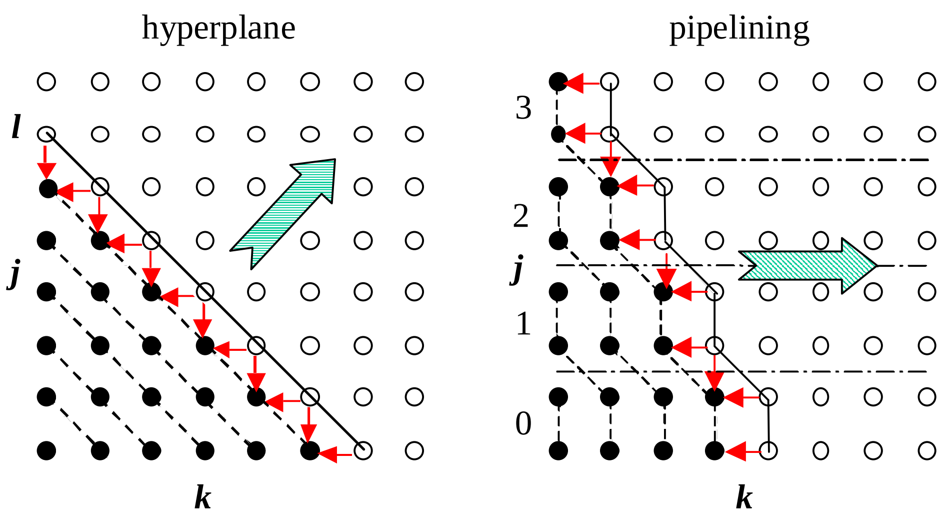

Figure 5: Schematic illustration of the hyperplane and pipeling algorithms.

The implementation of the parallelized hyperplane algorithm with directives is straightforward. Index arrays for all points on a given hyperplane *l* are first calculated. Loops for solving all points on a given hyperplane can then be performed in parallel. A sketch of the implementation for the lower triangular system is summarized below.

```
     DO L=LST, LEND
       CALL CALCNP(L, NP, INDI, INDJ)
!$OMP PARALLEL DO PRIVATE(I,J,K,N)
       DO N=1,NP
         I = INDI(N)
         J = INDJ(N)
         K = L – I - J
         CALL JACLD(I,J,K)
         CALL BLTS(I,J,K)
       END DO
     END DO
```

The upper triangular system is similar except that the L loop starts from `LEND` and decrements to `LST`. The index arrays can be pre-calculated before the time iteration loop to eliminate the repetition and improve the performance.

The second method for the parallel implementation of the SSOR algorithm in LU is *pipelining*. The method is illustrated in the right panel of Figure 5 for a case where four processors are working on the pipeline with work distributed along the J direction. Processor 0 starts from the low-left corner and works on one slice of data for the first K value. Other processors are waiting for data to be available. Once processor 0 finishes its job, processor 1 can start working on its slice for the same K and, in the meantime, processor 0 moves onto the next K. This process continues until all the processors become active. Then they all work concurrently to the opposite end. The cost of pipelining results mainly from the wait in startup and finishing. A 2-D pipelining can reduce the wait cost and was adopted in the MPI version of LU [10].

The implementation of pipeline in LU with OpenMP directives is done by the point-to-point synchronization through the "`!$OMP FLUSH()`" directive. This directive ensures a consistent view of memory from all threads for the variables enclosed in the argument and is used at the precise point at which the synchronization is required. As an example of using this directive to set up a pipeline, the structure of the lower-triangular solver in `SSOR` is illustrated in the following.

```
 !$OMP PARALLEL PRIVATE(K,iam,numt)
      iam = omp_get_thread_num()
      numt = omp_get_num_threads()
      isync(iam) = 0
 !$OMP BARRIER
      DO K=KST,KEND
        CALL JACLD(K)
        CALL BLTS(K,iam,isync,numt)
      END DO
 !$OMP END PARALLEL
```

The K loop is placed inside a parallel region, which defines the length of the pipeline (refer to Figure 5). Two OpenMP library functions are used to obtain the current thread ID (`iam`) and the total number of threads (`numt`). The globally shared array "`isync`" is used to indicate the availability of data from neighboring threads: 1 for ready, 0 for not ready. Together with the `FLUSH` directive it sets up the point-to-point synchronization between threads, as was done in routine `BLTS`.

```
      SUBROUTINE BLTS(K,iam,isync,numt)
      integer iam, isync(0:iam)
      ilimit = MIN(numt,JEND-JST)
      if (iam.gt.0 .and. iam.le.ilimit) then
        do while (isync(iam-1).eq.0)
!$OMP FLUSH(isync)
        end do
        isync(iam-1) = 0
!$OMP FLUSH(isync)
      endif
!$OMP DO
      DO J=JST,JEND
```

```
          Do the work for (J,K)
        ENDDO
!$OMP END DO nowait
        if (iam .lt. ilimit) then
          do while (isync(iam).eq.1)
!$OMP FLUSH(isync)
          end do
          isync(iam) = 1
!$OMP FLUSH(isync)
        endif
        RETURN
        END
```

The two `WHILE` loops set up waits through the variable `isync`. The `FLUSH` directive ensures the value of `isync` is up-to-date for all threads at the point where the directive is present. The first synchronization before the J loop behaves similar to the *receive* function in a message passing program, which waits for availability of data from the previous thread (`iam-1`). The second synchronization is similar to the *send* function, which sets a flag for the next thread. It is necessary to remove the end-of-loop synchronization for the J loops in both `JACLD` and `BLTS` with the "`NOWAIT`" construct since the pipeline implies asynchronous for the loop iterations.

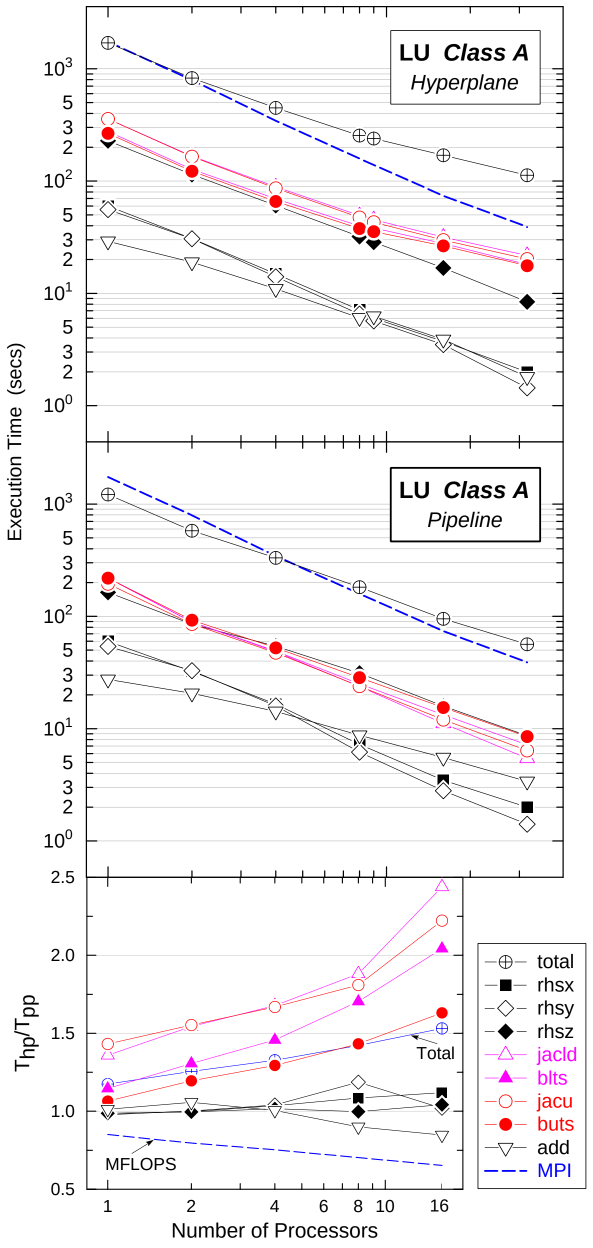

Figure 6: Execution profile of different components in LU for both the hyperplane and pipelining algorithms. Two methods are compared at the bottom.

Both parallel versions were tested on the Origin 2000 and compared with the MPI implementation. The time profiles for key routines are plotted in the upper panel of Figure 6 for the hyperplane version and in the middle for the pipeline version. Timing ratios of the two versions are given in the bottom panel. The pipeline implementation clearly has better performance than the hyperplane version, about 50% better on 16 processors. The RHS behaves similar, but the main differences come from the four main routines in the SSOR solver and become larger as the number of processors increases. It can be attributed to better cache utilization in the pipeline implementation. To support this argument, cache misses for the two versions were measured with the hardware counter available on the Origin 2000 R10K processor and the results are listed in Table 3 measurements done on 1 CPU and 4 CPUs. With one CPU, the pipeline version has slightly less L1 and L2 cache misses, but the TLB miss is significantly less. With four CPUs, both versions have less L2 cache misses and, still, the pipeline version has much better cache performance.

When comparing with the message passing implementation of pipelining, the directive-based version does not scale as well. We believe this performance degradation in the directive implementation due to the sizable synchronization overhead in the 1-D pipeline as against the 2-D pipeline used in the message passing version.

Table 3: Cache performance of the hyperplane and pipeline versions of LU, measured with the `perfex` tool on the Origin2000. Cycles and cache misses are given in seconds. Numbers were obtained for 1/10th of the full iteration for Class A.

	1 CPU		**4 CPUs**	
Counter	**hyperplane**	**pipeline**	**hyperplane**	**pipeline**
Cycles (secs)	152.987	133.395	158.671	137.769
L1 cache miss (secs)	47.508	44.693	47.296	44.760
L2 cache miss (secs)	39.164	32.234	20.744	14.417
TLB miss (secs)	30.419	13.317	31.205	12.857
L1 cache line reuse	5.646	7.062	6.226	9.315
L2 cache line reuse	9.165	10.618	18.106	25.016
L1 hit rate	0.8495	0.8760	0.8616	0.9031
L2 hit rate	0.9016	0.9139	0.9477	0.9616

4.5. Kernel Benchmark FT

Benchmark FT performs the spectral method with first a 3-D fast Fourier transform (FFT) and then the inverse in an iterative loop.

```
CALL SETUP
CALL FFT(1)
DO ITER=1, NITER
  CALL EVOLVE
  CALL FFT(-1)
  CALL CHECKSUM
END DO
```

The 3-D data elements are filled with pseudo random numbers in `SETUP`. Since the random number seed for each K value can be pre-calculated, the K loop that initializes each data plane can be done in parallel, as illustrated in the following:

```
  DO K=1, D3
```

```
          seeds(K) = Calc_seeds(K, D1, D2)
        END DO
!$OMP PARALLEL DO PRIVATE(K...)
        DO K=1, D3
          U0(K_plane) = Generate_prandom(seeds(K), D1*D2)
        END DO
```

The 3-D FFT in the kernel is performed with three consecutive 1-D FFTs in each of the three dimensions. The basic loop structure of the 1-D FFT is as follows (for the first dimension).

```
!$OMP PARALLEL DO PRIVATE(I,J,K,W)
        DO K = 1, D3
          DO J = 1, D2
            DO I = 1, D1
              W(I) = U(I, J, K)
            END DO
            CALL CFFTZ(..., W)
            DO I = 1, D1
              U(I, J, K) = W(I)
            END DO
          END DO
        END DO
```

A slice of the 3-D data (U) is first copied to a 1-D work array (W). The 1-D fast Fourier transform routine `CFFTZ` is called for W. The result is returned to W and, then, copied back to the 3-D array (U). Iterations of the outside K and J loops can be executed independently and the "**`PARALLEL DO`**" directive is added to the K loop with working array W as private. Better parallelism could be achieved if the nested J loop is also considered. However, this would require the use of non-standard extensions to OpenMP directives provided in the SGI MIPSpro compiler used in this study.

Inside the iteration loop, routine `EVOLVE` computes the exponent factors for the inverse FFT. It contains three nested `DO` loops. The outer loop is selected for parallelization with directives. Lastly, a parallel reduction is implemented in routine `CHECKSUM`.

The execution profile of several main components of the parallel code for Class A is shown in Figure 7. The three 1-D FFT routines and `SETUP` scale up very well although `EVOLVE` performs slightly worse for more than 16 processors. `CHECKSUM` used very little time, thus, was not shown in the figure. It is worthwhile to point out that the overall performance of the OpenMP version is about 10% better than the hand-coded MPI implementation from NPB2.3. The better performance of the directive version is due to the elimination of a 3-D data array which was needed in the MPI version. The change has improved the memory utilization of the code.

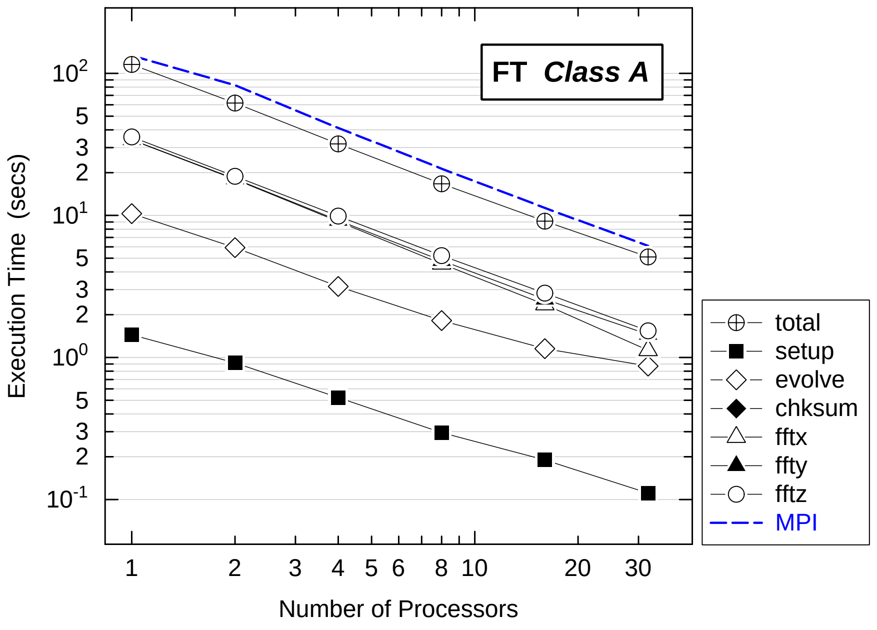

Figure 7: Execution profile of different components in the OpenMP-based FT. The total time is also compared with the MPI version from NPB2.3.

4.6. Kernel Benchmark MG

The iteration loop of MG consists of the multigrid V-cycle operation and the residual calculation. The V-cycle algorithm is performed in the following sequence:

```
CALL rprj3
CALL psinv
CALL interp
CALL resid
CALL psinv
```

The residual is first restricted from the fine grid to the coarse with the projection operator (`rprj3`). An approximate solution is then calculated on the coarsest grid (`psinv`), followed by the prolongation of the solution from the coarse grid to the fine (`interp`). At each step of the prolongation, the residual is calculated (`resid`) and a smoother is applied (`psinv`).

Parallelization of MG with directives is straightforward. "`PARALLEL DO`" directives are added to the outer-most loops in the above mentioned routines. Reductions (`+` and `MAX`) are used in the calculation of norm. Since the loops are well organized in the benchmark, it is not surprising that the performance of the OpenMP version is very close to the MPI version (as compared in Figure 8). The execution profile of each component in MG has shown consistent results and good speedup. It indicates that the loop-level parallelization has worked well for MG.

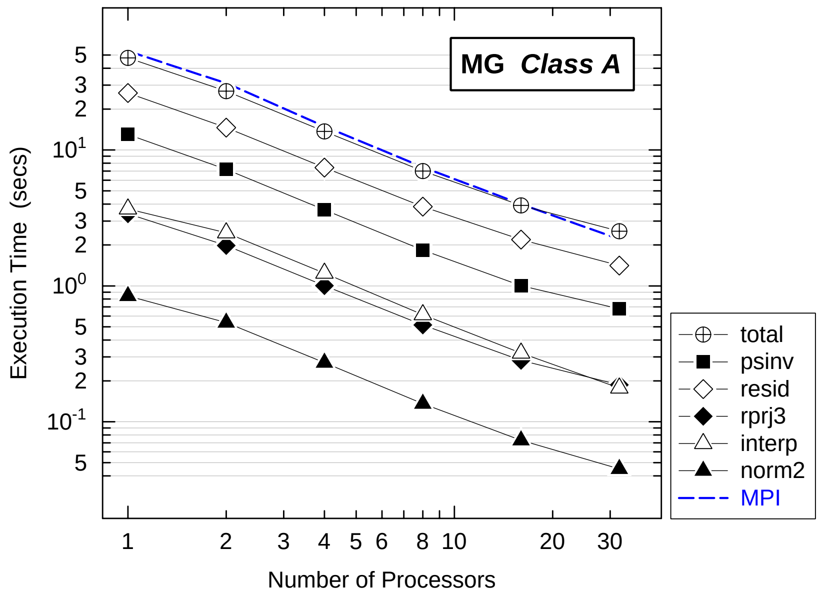

Figure 8: Execution profile of different components in the OpenMP-based MG. The total timing of the MPI version from NPB2.3 is plotted as a reference (dash line).

4.7. Kernel Benchmarks CG and EP

CG and EP are probably the easiest ones for parallelization in the seven studied benchmarks. So we summary them here in the same subsection.

The parallelization of CG is mostly performed on the loops inside the conjugate gradient iteration loop, which consists of one sparse-matrix vector multiplication, two reduction sums and several *paxpy* operations. Norms are calculated (via reduction) after the iteration loop. Adding "PARALLEL DO" directives with proper reductions on these loops seems working reasonably well, as illustrated by the time profile in Figure 9. The change of execution time from 1 to 2 CPUs is sub-linear, but from 4 to 8 CPUs is super-linear. This seems occurring in the MPI version as well. But for more than 16 processors, the performance of the directive version degrades quickly, most likely due to that the overhead associated with directives wins over the small workloads for the loops considered. Setup of the

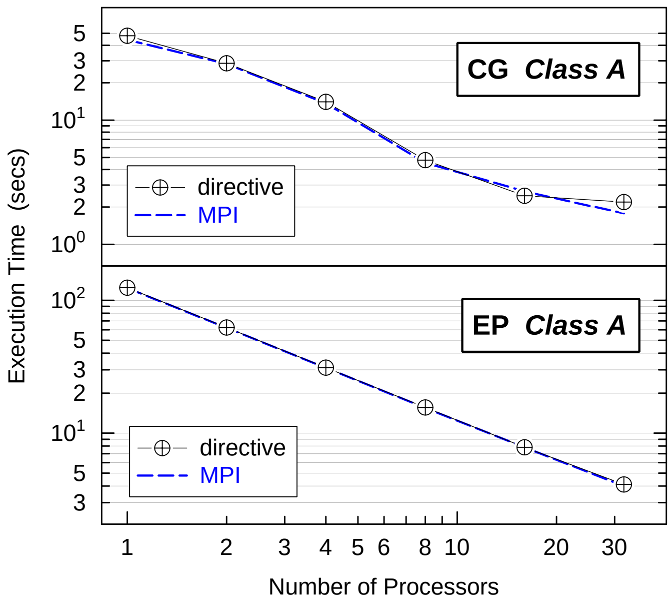

Figure 9: Execution time of the OpenMP directive-based CG and EP versus the MPI implementations.

sparse matrix (`makea`) with random numbers was done in serial. Profiling with the `perfex` tool on the Origin 2000 has indicated that irregularly accessing the matrix causes large amount of L1/L2 cache misses, which certainly affect the performance on large number of processors.

In the EP benchmark, the generation of pseudo random numbers can be done in parallel (as in the case of FT). The main loop for generating Gaussian pairs and tallying counts is totally parallel, with several reduction sums at the end. Performance data (Figure 9, lower panel) have indicated that the directive parallel version has a linear speedup, essentially overlapping with the MPI implementation.

4.8. Performance Summary

To summarize the results, the reported MFLOPs per second per processor for all seven benchmarks are plotted in Figure 10 and the execution times are in Figure 11. These numbers were obtained on the Origin 2000, 195 MHz, for the Class A problem. The measured execution time and MFLOPs are included in the Appendix, together with the corresponding values from the MPI versions for a comparison.

As one can see, overall OpenMP versions performed quite well, close or similar to the MPI correspondence in many cases, even better for FT. However, the OpenMP version generally does not scale as well as the MPI version. We believe this is mainly due to the lack of nested parallelization (on multiple dimensions) in the implementation as was done in the MPI version. Although the OpenMP standard has a notion of nested parallel `DO`'s, but the compiler tested does not fully support this function. The second

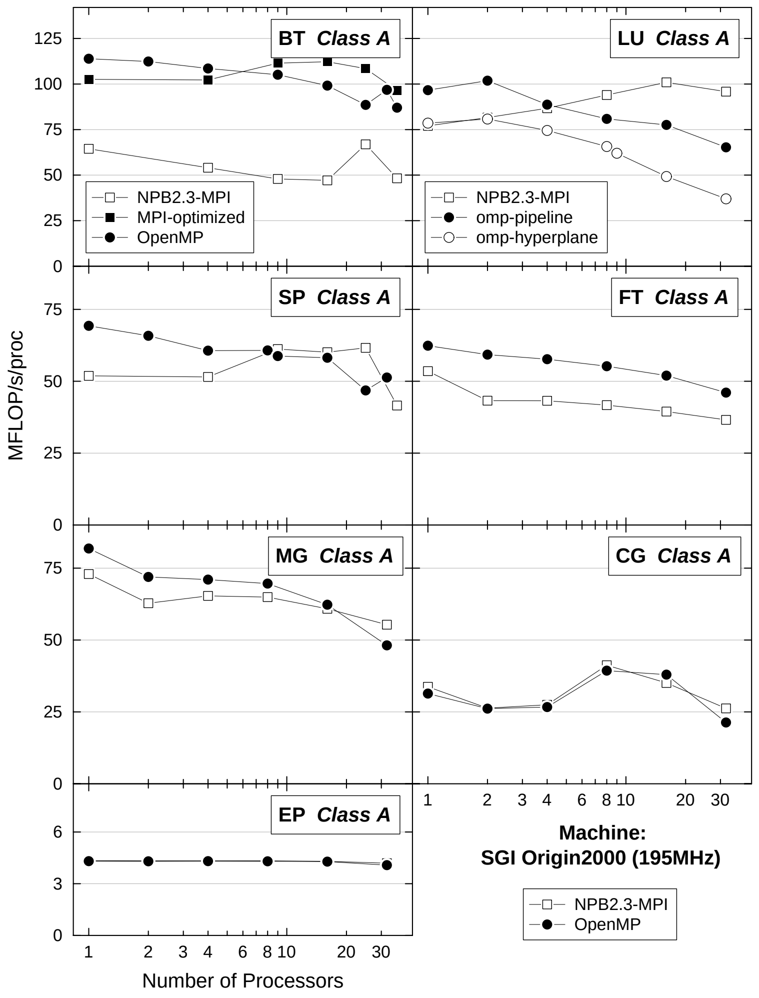

Figure 10: Measured MFLOP/sec/processor for the seven benchmarks parallelized with OpenMP directives (filled symbols) in comparison with the NPB2.3-MPI version (open symbols). Two additional curves are included for the optimized MPI version of BT and a hyperplane OpenMP implementation of LU.

factor is due to the long-stride access of memory in several codes, which is not cache friendly and causes remote-memory congestion. In the MPI version, the memory access is pretty much local (to a compute node). The utilization of overlapping computation and com-munication in the MPI versions, such as BT and SP, improves scalability.

At least in the case of CG, the performance degradation on large number of processors is due to the poor cache handling in the code, as indirectly indicated by the relative small MFLOP values compared to the other benchmarks (except for EP). The parallelization for CG with directives was done at much finer-grained loop levels and the overhead associated with it will likely dominate on the large number of processors.

In LU, the pipeline implementation performed better than the hyperplane version, in both time and scalability. The pipeline version has better cache performance. In order to get even closer to the performance of the MPI version, a 2-D pipelining seems necessary.

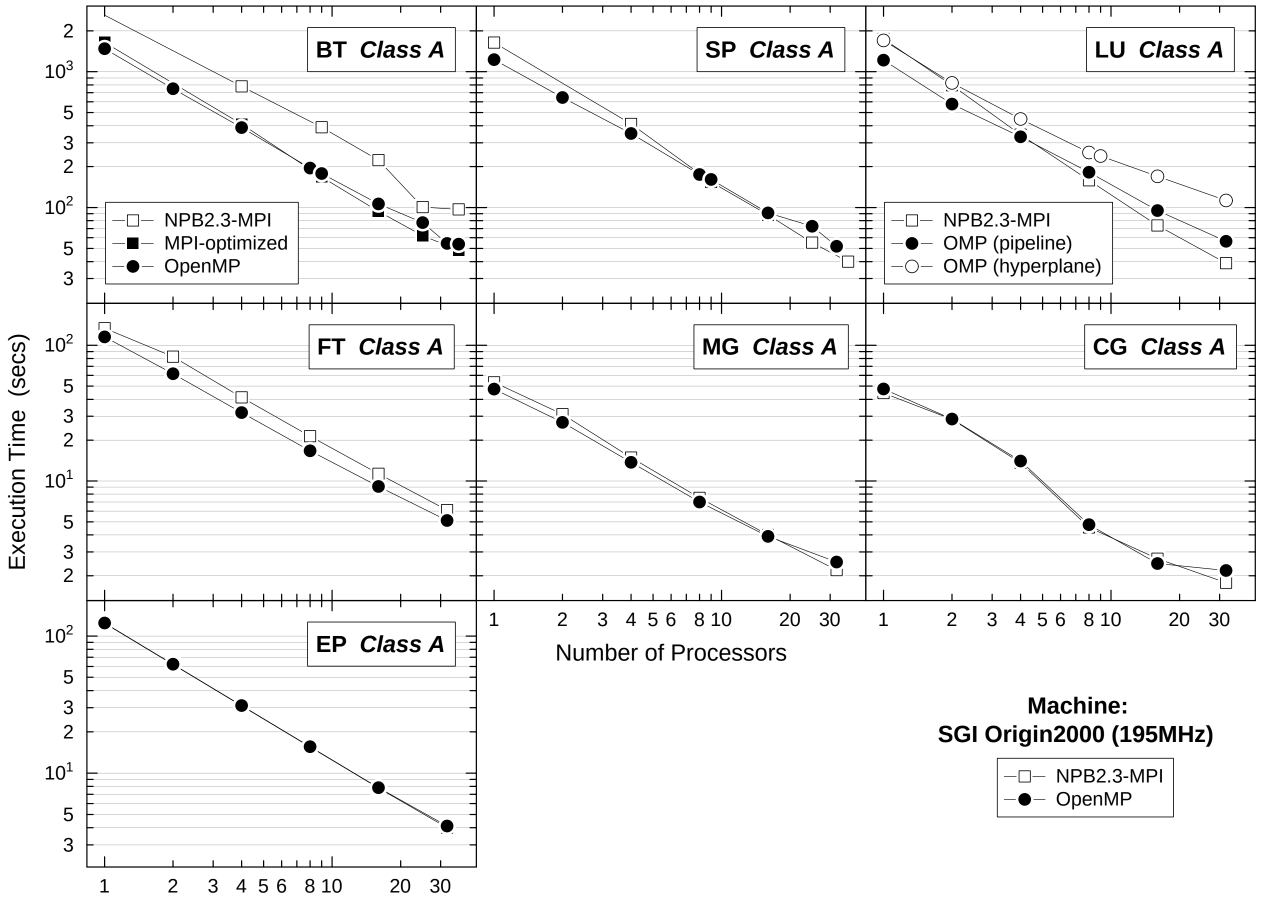

Figure 11: Measured execution time for the seven benchmarks parallelized with OpenMP directives (filled symbols) in comparison with the NPB2.3-MPI version (open symbols). Two additional curves are included for the optimized MPI version of BT and a hyperplane OpenMP implementation of LU.

Another observation is that the memory optimization for BT has impact not only on the serial performance, but also on the parallel performance. The OpenMP version of BT is about a factor of two better than the MPI version in NPB2.3 (filled circles vs. open squares in Figure 10). After applying the similar optimization technique to the MPI version, the performance has been improved by a factor of two (filled squares in Figure 10). Because of the limited cache size in a machine, it is still important to effectively utilize memory to reduce the cache misses (cache-memory traffic) and improve the performance. The improvement for SP is not as profound as that for BT, but the effect is visible on small number of processors.

5. Conclusion

The current work presented a study of the effectiveness of the OpenMP directives to parallelize the NAS Parallel Benchmarks. The seven benchmarks implemented show very good performance, even though the scalability is worse than the MPI counterpart. However, this situation could be improved with the use of nested parallelism (or even multi-level parallelism) at different loop nesting levels. Still the most plausible strength of OpenMP is its simplicity and the incremental approach towards parallelization.

The OpenMP implementations were based on the optimized sequential versions of the original NPB2.3-serial [4]. Together with an HPF implementation [6], they form the Programming Baseline for NPB (PBN). Techniques explored in the study can certainly be applied to more realistic applications and can be used in the development of parallization tools and compilers. In fact, this study resulted from the comparison of parallization tools and compilers [2] and was done in conjunction with the development of a parallelizing tool, CAPO [8], for the automated generation of OpenMP parallel programs. Another earlier work [11] has presented the study of tools and compiler for parallelization of NAS Benchmarks with directives.

Further work will be done on the improvement of the scalability of PBN-OpenMP and the development of an OpenMP C version of IS benchmark for the completion of the suite. New effort will emphasize on the development of a benchmark suite to incorporate the existing benchmarks to run concurrently on a computational grid environment.

We would like to acknowledge the valuable discussion with and suggestion from the original NPB developers, in particular, Drs. Maurice Yarrow and Rob Van der Wijngaart at NASA Ames Research Center. We also want to thank Dr. Gabriele Jost for testing the implementation.

References

[1] High Performance Fortran Forum, "High Performance Fortran Language Specification," CRPC-TR92225, January 1997, http://www.crpc.rice.edu/CRPC/softlib/TRs_online.html.

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

[3] 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.

[4] KAP/Pro Toolset, "Assure/Guide Reference Manual," Kuck & Associates, Inc. 1997.

[5] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, “The NAS Parallel Benchmarks 2.0,” *NAS Technical Report NAS-95-020*, NASA Ames Research Center, Moffett Field, CA, 1995. http://science.nas.nasa.gov/Software/NPB.

[6] M. Frumkin, H. Jin, and J. Yan, “Implementation of NAS Parallel Benchmarks in High Performance Fortran,” *NAS Technical Report NAS-98-009*, NASA Ames Research Center, Moffett Field, CA, 1998.

[7] M. Frumkin, M. Hribar, H. Jin, A. Waheed, and J. Yan, “A Comparison of Automatic Parallelization Tools/Compilers on the SGI Origin2000 using the NAS Benchmarks,” in Proceedings of SC98, Orlando, FL, 1998.

[8] H. Jin, J. Yan and M. Frumkin, “Automatic Generation of Directive-based Parallel Programs with Computer-Aided Tools,” in preparation.

[9] 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/.

[10] M. Yarrow and R. Van der Wijngaart, “Communication Improvement for the LU NAS Parallel Benchmark,” *NAS Technical Report NAS-97-032*, NASA Ames Research Center, Moffett Field, CA, 1997.

[11] 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.

Appendix

Table 4: Execution time and MFLOP/sec/proc (in parenthesis) of the OpenMP directive-based NPBs (7 benchmarks) measured on the SGI Origin 2000, 195MHz. For comparison, timings of the MPI implementation from NPB2.3 are also included in the table.

BT Class A			
#Procs	**NPB2.3-MPI**	**MPI-optimized**	**OpenMP**
1	2611.0 (64.45)	1641.2 (102.54)	1477.8 (113.87)
2	-	-	749.0 (112.34)
4	778.6 (54.04)	411.6 (102.22)	387.6 (108.54)
9	390.5 (47.88)	167.8 (111.44)	177.9 (105.09)
16	223.2 (47.12)	93.7 (112.22)	106.1 (99.09)
25	100.6 (66.90)	62.1 (108.48)	76.0 (88.55)
32	-	-	54.3 (96.79)
36	96.8 (48.27)	48.5 (96.42)	53.7 (87.00)

SP Class A		
#Procs	**NPB2.3-MPI**	**OpenMP**
1	1638.4 (51.89)	1227.1 (69.28)
2	-	646.0 (65.80)
4	412.8 (51.49)	350.4 (60.64)
8	-	175.0 (60.74)
9	154.4 (61.18)	160.8 (58.74)
16	88.4 (60.11)	91.4 (58.15)
25	55.2 (61.61)	72.7 (46.79)
32	-	51.8 (51.28)
36	56.8 (41.58)	-

LU Class A			
#Procs	**NPB2.3-MPI**	**OpenMP-hp**	**OpenMP-pipe**
1	1548.4 (77.05)	1518.3 (78.57)	1234.4 (96.64)
2	731.1 (81.58)	738.2 (80.81)	585.6 (101.86)
4	344.0 (86.71)	400.3 (74.49)	336.2 (88.69)
8	158.7 (93.99)	227.1 (65.68)	184.4 (80.88)
9	-	213.8 (61.99)	-
16	73.9 (100.90)	151.5 (49.22)	96.1 (77.54)
32	38.9 (95.83)	100.7 (37.02)	57.1 (65.23)

FT Class A		
#Procs	**NPB2.3-MPI**	**OpenMP**
1	133.4 (53.50)	114.46 (62.35)
2	82.5 (43.25)	60.22 (59.25)
4	41.3 (43.20)	30.94 (57.66)
8	21.4 (41.68)	16.15 (55.23)
16	11.3 (39.47)	8.59 (51.95)
32	6.1 (36.56)	4.84 (46.05)

MG Class A		
#Procs	**NPB2.3-MPI**	**OpenMP**
1	53.4 (72.92)	47.58 (81.81)
2	31.0 (62.80)	27.06 (71.92)
4	14.9 (65.33)	13.71 (71.00)
8	7.5 (64.90)	6.99 (69.60)
16	4.0 (60.84)	3.91 (62.27)
32	2.2 (55.31)	2.53 (48.15)

CG Class A		
#Procs	**NPB2.3-MPI**	**OpenMP**
1	44.40 (33.71)	47.74 (31.35)
2	28.46 (26.30)	28.65 (26.12)
4	13.62 (27.46)	14.03 (26.66)
8	4.54 (41.24)	4.76 (39.32)
16	2.67 (35.05)	2.46 (37.95)
32	1.78 (26.20)	2.19 (21.32)

EP Class A		
#Procs	**NPB2.3-MPI**	**OpenMP**
1	123.6 (4.34)	124.46 (4.31)
2	62.0 (4.33)	62.34 (4.30)
4	31.0 (4.33)	31.13 (4.31)
8	15.5 (4.33)	15.59 (4.30)
16	7.8 (4.30)	7.83 (4.28)
32	4.0 (4.19)	4.11 (4.08)