

OpenMP-Oriented Applications for Distributed Shared Memory Architectures *

Ami Marowka

Zhenying Liu

Barbara Chapman

Department of Computer Science

The University of Houston, Texas

E-mail: amimar@cs.uh.edu

ABSTRACT

The fast emergence of OpenMP as the preferable parallel programming paradigm for small-to-medium scale parallelism could decline unless OpenMP will show capabilities to be the model-of-choice for large scale high performance parallel computing of the next decade.

The main stumbling block from adapting OpenMP for distributed shared memory (DSM) machines, which are based on architecture like cc-NUMA, stems from the absence of capabilities for data placement among processors and threads for achieving data locality. The absence of such mechanism causes remote memory accesses and inefficient cache memory use, both of which lead to poor performance.

This paper presents a simple software programming approach called Copy-inside-Copy-back (CC) that exploits the privatization mechanism of OpenMP for data placement and re-placement. This technique enables one to distribute data without taking the control and the flexibility from the programmer, and thus, is an alternative to the traditional implicit and explicit approaches. Moreover, the CC approach enables SPMD style of programming that makes the development process of an OpenMP application more structured, and simply to modify and debug.

The CC technique was tested and analyzed using the NAS Parallel Benchmarks on SGI Origin 2000 multiprocessors machine. The lesson learnt from this study shows that OpenMP can deliver the desired large-scale parallelism although fast copy mechanism is essential.

KEY WORDS

OpenMP, Data Locality, NPB, Programming Model

1 Introduction

The fast emergence of OpenMP [9] as the preferable parallel programming paradigm for small-to-medium scale parallelism could decline unless OpenMP will show capabilities

to be the model-of-choice for large scale high performance parallel computing of the next decade.

OpenMP was designed to be "mean and lean", simple, portable parallel programming model for shared memory architecture. OpenMP is based on multi-threading model of computation and is destined for the scientific community. On the other hand, this community uses high-performance parallel machines which are multiprocessors machines with distributed-shared memory (DSM) based on NUMA architecture. The main stumbling block from adapting OpenMP on NUMA architecture stems from the absence of facilities for data placement among processors and threads to achieve data locality. The absence of such mechanism causes remote memory accesses and inefficient cache memory use, both of which lead to poor performance.

This issue is currently the main open question in the agenda of the OpenMP community. There are members who suggest that one should annotate the application source code with explicitly compiler directives to handle the data distribution task. Their model of imitation is HPF-like directives [1, 2, 4, 10]. On the other hand, some colleagues claim that run-time system can do it implicitly without the intervention or discretion of the programmer [3]. However, they have something in common. Both are concern with losing the achievements of OpenMP – its simplicity and portability. We are sharing the same concerns and think that there is a third way. We believe that the current version of OpenMP includes rich enough mechanisms for constructing portable and scalable applications. By applying software engineering techniques, it is possible to achieve the desired portable performance from NUMA machines.

The contribution of this paper is as follows. First, we show how to apply Single-Program-Multi-Data (SPMD) style of parallel programming to OpenMP applications. SPMD parallel programming is achieved by using PRIVATE clause for memory allocation to each thread of execution. By using SPMD programming, the development process of the application becomes more structured and simple to write, modified and debugged. Privatization of data enables one to save memory, increase data locality and

*This work was partially supported by NASA Ames Research Center under contract NCC2-5394, and by NSF under grant number NSF ACI 99-82160

achieve thread-safe programming. Moreover, using a simple software programming approach called Copy-inside-Copy-back (CC), we show how data distribution and redistribution can be implemented without taking the control and the flexibility from the programmer, and thus an alternative to the traditional implicit and explicit techniques. This will be demonstrated by studying the BT and FT NAS Parallel Benchmarks. The BT and FT kernels were ported to OpenMP by NASA researchers in a simple and straightforward manner [8]. It was challenging to pick BT kernel for our study. At first sight, BT has not been a good candidate to demonstrate our ideas since it cannot be privatized in a straightforward manner as was done in previous works [5, 6, 7]. After the first impression, we have found that privatization could be applied in BT kernel by using CC technique with negligible modification and enjoy from all the advantages of SPMD way of programming. On the other hand, in the FT kernel the CC approach was used by NAS scientists implicitly and from different points of view. In the following sections, we study step by step the different angles of using CC approach in FT and BT kernels.

The remainder of this paper is organized as follows: Section 2 is a brief introduction to OpenMP. In Section 3 we study the BT and FT Benchmarks using the CC approach. In Section 4 we present and analyze the running results conducted on the Origin 2000 machine. Section 5 presents the related works suggested to make OpenMP appropriate for distributed shared memory NUMA machines. Section 6 summarizes our conclusions.

2 OpenMP Programming Model

OpenMP is a tool for writing multi-threaded applications in a shared memory environment. It consists of a set of compiler directives and library routines. The compiler generates a multi-threaded code based on the specified directives. OpenMP is essentially a standardization of the last 15 years or so of SMP (Symmetric Multi-Processor) development and practice. Using OpenMP, it is relatively easy to create parallel applications in Fortran, C and C++. Compiler and 3rd-party applications support is common and growing.

An OpenMP program begins with a single thread of execution, called the master thread. The master thread spawns teams of threads in response to OpenMP directives, which perform work in parallel. Parallelism is thus added incrementally; the serial program evolves into a parallel one. OpenMP directives are inserted at key locations in the source code. These directives take the form of comments in Fortran and `#pragmas` in C and C++. The compiler interprets the directives and creates the necessary code to parallelize the indicated tasks/regions. The parallel region is the basic construct that creates a team of threads and initiates parallel execution. Most OpenMP directives apply to structured blocks, which are blocks of code with one entry point at the top and one exit point at the bottom.

The number of threads created when entering parallel regions is controlled by the value of the environment variable `OMP_NUM_THREADS`. The number of threads can also be set by a function call from within the program, which takes precedence over the environment variable. It is possible to vary the number of threads created in subsequent parallel regions. Each thread executes the block of code enclosed by the parallel region.

In general, there is no synchronization between threads. Different threads may reach the end of the parallel region at different times. OpenMP does provide constructs for synchronization, but the code should not be written in such a way that its output depends upon different threads executing statements at particular times. OpenMP provides a number of constructs for thread synchronization and coordination among them: `critical`, `atomic`, `barrier`, `master`. These are sufficient for many needs, but OpenMP also provides a set of runtime thread locking functions which can be used for fine control. When all threads reach the end of the parallel region, all but the master thread go out of existence and the master continues alone. The OpenMP directive clauses - `Private`, `Shared` and `Default` - control whether the listed variables are shared among different threads, or are private (local) to each thread.

OpenMP provides several constructs for sharing work among threads in a team. These are: `Parallel for/DO`, `Parallel Sections`, `Workshare` and `Single` directive. These constructs are placed inside an existing parallel region. The result is to distribute execution of associated statements among the existing threads. A number of environment variables may be set to control aspects of OpenMP execution. For example, the number of threads, loop scheduling, and the enabling of nested parallelism and dynamic adjustment of the number of threads. OpenMP also provides a number of routines that may be called from within your code. These may be used to get and set the number of threads, enable or disable dynamic thread allocation, check whether the code is executing in parallel, etc. Changes in the runtime environment made by these routines have precedence over the corresponding environment variables.

3 OpenMP-SPMD Design

In this section we explain step-by-step how to design OpenMP applications for SPMD-style programming. The BT and FT NAS benchmarks [8, 11] are used as test cases. Our starting point is the OpenMP-BT version implemented by NASA researchers for the cc-NUMA architecture like Origin 2000 [1]. We study in detail the parallelism degree of the application, explore its weaknesses, struggle with the problems that prevent us from using SHADOW buffers and from privatizing in a straightforward manner as was proposed in [5, 6, 7]. Then, we show how with negligible modification, the application can be redesigned to take advantage of SPMD-style programming. In the next step, we present how the CC approach is used in FT kernel and

```

DO STEP = 1, NITER
  CALL COMPUTE-RHS
  CALL X-SOLVE
  CALL Y-OLVE
  CALL Z-OLVE
  CALL ADD
END DO

```

Figure 1. BT NAS Benchmark

```

!$OMP PARALLEL DO PRIVATE(I,J,K)
  DO K = 1, N
    DO J = 1, N
      ---
      ---
      DO I = 1, N
        RHS (I,J,K) = RHS(I,J,K) - RHS(I-1,J,K)
      ENDDO
    ENDDO
  ENDDO
!$OMP END PARALLEL

```

Figure 2. X-SOLVE Parallel Region.

explain the different perspectives taken in the FT and BT kernels for achieving scalable performance using the same technique.

3.1 BT NAS Benchmark

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 Block-Tridiagonal of 5x5 blocks and are solved sequentially along each dimension. The main iteration loop in BT contains the steps shown in Figure 1.

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). Each one of the (X-SOLVE, Y-SOLVE, Z-SOLVE) subroutines is enclosed in the parallel region as shown in Figure 2.

In Figure 2 the loop control variables (I, J, K) are related to the (X, Y, Z) dimensions respectively. In this case RHS is processed along the X dimension. The analysis of the parallel region must be done at two levels: the application level and underlying architecture level. For the upper application level OpenMP does the parallelization on the outer-most parallel loop. Hence, the K iterations are distributed among the threads evenly, while the stencil operation in the parallel region uses I-1 and I elements of the solution array to compute RHS along the X-dimension. Moreover, the stencil operation shows data dependence in the X-direction and therefore prevents us from parallelization in the X-dimension. Therefore, the parallelization is done along the Z dimension.

In the underlying architecture level, RHS is distributed among the processors. BT uses the First Touch policy mechanism supported by most operating systems [1, 2, 4]. Based on this policy, a data page is owned by a processor that touches the data page first unless page migration is turned on. Initialization loops at the beginning of the program touch the data pages first in (*, *, BLOCK) manner so the distribution of RSH is done along the Z dimension (Figure 4). Since the physical distribution of RHS is done along the Z-dimension while the parallelization of the DO LOOP iterations is done along the X-dimension, each thread works only on the local part of RHS attached to the host processor.

Y-SOLVE is very similar. In this case, RHS is processed along the Y-dimension while the parallelization of the DO LOOP iterations is done along the Z-dimension. There is no change in the physical distribution of RHS. Therefore, Y-SOLVE keeps the locality of the processing as it does in X-SOLVE.

Z-SOLVE calls for slight change. Since RHS is processed along the Z-dimension and the stencil operation shows data dependence in this direction, the parallelization along the Z-dimension cannot be done. To overcome this problem, the K loop is pushed inside the nested loops while the J loop is moved to be the outer-most loop as shown in Figure 3.

Now, since the DO LOOP iterations are distributed along the Y-dimension and the physical distribution of RHS remains along the Z-dimension, crossing the boundary of the local memory domain is inevitable. Therefore, the processing of RHS is suffered from remote memory accesses that cause significant communication overhead.

This is the right moment to check if privatization and SHADOWING technique [5, 6, 7] can be adopted here. As we show, the processing of RHS in Z-SOLVE creates remote memory accesses along the Z dimension. These remote memory accesses prevent us from declaring the distributed parts of RHS as private. Therefore, privatization cannot be applied to RHS in a straightforward manner and we have to turn to a different solution. SHADOWING is a technique that enables us to exchange private data between each pair of processors via shared memory buffers called

```

!$OMP PARALLEL DO PRIVATE(I,J,K)
  DO J = 1, N
    DO I = 1, N
      ---
      ---
      DO K = 1, N
        RHS (I,J,K) = RHS(I,J,K) - RHS(I,J,K-1)
      ENDDO
    ENDDO
  ENDDO
!$OMP END PARALLEL

```

Figure 3. Z-SOLVE Parallel Region.

SHADOWS. Such a technique is very useful in cases like Domain Decomposition methods where the boundary data of adjacent processors are exchanged. In Z-SOLVE the entire data of RHS are required for processing and therefore SHADOWING technique is not appropriate in this case.

3.2 Copy-inside-Copy-back Technique

The motivation for using privatization stems from the following advantages. Privatization enables structured programming, which makes applications simple to understand and debug. Memory is saved since private memory is allocated only at the entry to a parallel region, and is deallocated at the exit. Locality is enhanced since accessing private data does not require remote memory accesses. And safe thread programming is enabled since private data free the programmer from difficulties with race conditions, conflicts, and deadlocks. Armed with this motivation we start to redesign BT.

We keep RHS in the shared memory and add RHS-SLICE, a private working area inside the parallel region (Figure 4). At the beginning of the parallel region, each thread copies a corresponding slice from RHS into its private RHS-SLICE. At the end of the parallel region, the opposite happens. Each thread copies back the content of its private RHS-SLICE to the corresponding place in the shared memory RHS. The modified parallel region is shown in Figure 5.

The purpose for using the CC approach in the example of BT is to reduce the communication overhead occurring in the X-SOLVE subroutine. The long strides which cause low cache reuse could be handled also during the copy-inside phase. However, we chose not to do so since code modifications are required that we do not encourage. In the next subsection we describe how the CC approach is integrated implicitly in the FT kernel. In this case the motivation for using The CC technique is to increase cache reuse.

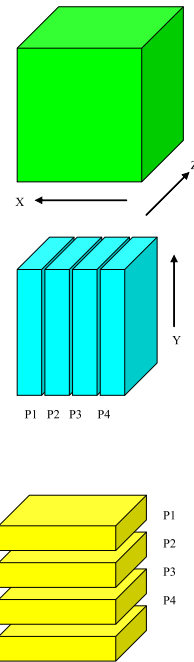


Figure 4. Distribution of PRIVATE Slices.

3.3 FT NAS Benchmark

FT NAS benchmark contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFT's, one for each dimension. The basic loop structure of the 1-D FFT (for the first dimension) is shown in Figure 6. 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.

3.4 Discussion

The BT and FT kernels, which are described above, use the same CC approach but from different perspectives and goals as summarized in Table 1. In BT the private slice RHS-SLICE, which is copied into the parallel region, is a large slice comparable to the private slice W that is used in FT. The granularity of the computation that is wrapped by the pair of copies is a fine grain in the case of BT. In the case of FT it is a coarse grain. The copying of the slices in BT is done from remote memory to the private memory, while in FT the copying is done from local memory to the private memory. Thus, the purpose of using CC technique

```

!$OMP PARALLEL DO PRIVATE(I,J,K,RHS- SLICE)
COPY-INSIDE(RHS,RHS-SLICE)
  DO J = 1, N
    DO I = 1, N
      ---
      ---
    DO K = 1, N
      RHS-SICE(I,J,K) = RHS-SLICE(I,J,K)
                        - RHS-SLICE(I,J,K-1)

    ENDDO
  ENDDO
ENDDO
COPY-BACK(RHS,RHS-SLICE)
!$OMP END PARALLEL

```

Figure 5. Modified Z-SOLVE Parallel Region.

```

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

```

Figure 6. FT NAS Benchmark.

	BT vs. FT			
	Data Size	Data Grain	Copy From/To	Goal
BT	Large	Fine	Remote/Private	Reduc Comm.
FT	Small	Coarse	Local/Private	Inc. Cache Reuse

Table 1. Different Perspectives Using CC

in BT is to reduce the communication overhead where as in FT the goal is to increase the cache reuse.

In general, each parallel region can declare its working-area as private and import/export the input/output data in the same manner. In this way, privatization is achieved with all the value-added and the advantages that make high performance applications in well written software engineering techniques. Moreover, the methodology shown here does not call for modifications or rewriting of existence code. Only adding a pair of copy calls and their corresponding private working-areas. Thus, the advantage of OpenMP namely, the simplicity and the portability of the model are not harmed.

Another issue which is solved implicitly using the Copy-inside- Copy-back technique is the data placement. The data distribution is actually executed in the Copy-inside phase where each thread copies a slice from the shared memory working-area for processing privately. In this way the data distribution can be done in any shape and dimension. Thus, the control and the flexibility of the distribution process are not taken from the programmer. Actually, data redistribution occurs when a successive parallel region is encountered and is based on the programmer's decision.

The remaining point of examination is the scalability issue. The communication overhead that might harm the scalability lies in the pair copy calls that are added to each parallel region. Fast copy can be implemented by using a Fortran 90 array section assignment statement where the compiler can take advantage from doing it in a burst mode. Moreover, these copy calls should be more light-weight compared to the coarse grain computation enclosed in the parallel region and therefore a great savings in remote memory accesses and cache-coherence memory management. However, fast copying of large data chunks between far nodes of multiprocessor machines is essential for achieving high scalability and is an implementation depended feature. Therefore, only practical benchmarks will show its performance. The next section depicts our benchmarks results.

4 Benchmarks Analysis

This section depicts running benchmarks followed by analysis of the results. The test-cases, the BT and FT NAS kernels, were carried out on the SGI Origin 2000 multi-processor machine. A brief description of this machine is provided below. The running results which are shown in this section are for CLASS A. The execution times were measured a few times for each run, and the average times were calculated. The runs were executed in dedicated user mode with 32 processors, with MIPSpro7 FORTRAN 90 compiler from SGI with the options: `-64 -c -O3 -mp -IPA`, OpenMP API version 1.1. First touch policy was activated, and the environment variable `_DSM_MIGRATE` was set to OFF. Parameters that are not mentioned were set to their default values.

4.1 SGI Origin 2000

The SGI Origin 2000 (O2k) is the most widely used NUMA system in the HPC community. A standard O2k configuration could have up to 128 processors in a single system, although custom installations of up to 512 processors in a single system exist. The O2k is composed of "nodes" of two MIPS R12k processors connected to a local memory bank via a crossbar switch. The topology of the system is a hypercube of routers, where each router can have two nodes directly connected to it. In configurations larger than 64 processors, the hypercube router topology is replaced by hypercubes of routers connected through metarouters. The latency cost of a remote memory access is at most three times that of a local memory access.

The benchmarks presented in this paper were executed on NCSA's machine with 32 (195MHz, R10000) processors, instruction cache size of 32 KB and cache line of 64 bytes, data cache size of 32 KB and cache line of 32 bytes, secondary unified cache of 4 MB and cache line of 128 bytes and Operating system IRIX 6.5

4.2 Results Analysis

Figures 7 and 8 show the running results for BT and FT applications respectively. The times shown for BT are only of the first order stencil operation along the Z-dimension in the Z-SOLVE subroutine, and the pair of copies that wraps it, as explained in sections 3.1 and 3.2. The times shown for FT are only of the CFFTZ subroutine and the pair of copies that wraps it, as explained in section 3.3.

At the bottom of each figure there is an execution times table of the running results. The times are in seconds. Two run types are shown, *private* and *shared*. The *shared* denotes that the main working area is allocated in shared memory. The *private* denotes that the working area is defined as private and that the CC approach is applied. Thus, the *private* columns contain the computation times, copy-inside and copy-back times, while the *shared* columns con-

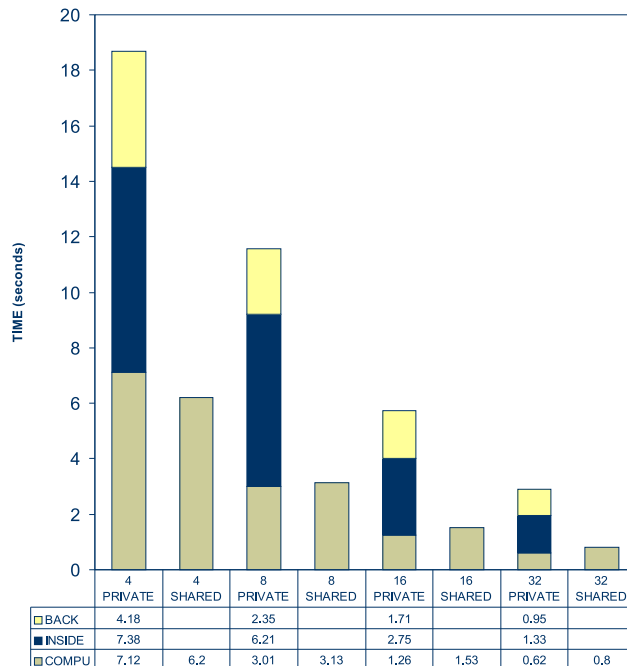


Figure 7. BT Running Results.

tain only the computation times. The results shown are for 4, 8, 16, 32 processors.

The results show that the performance is harmed drastically, when CC approach is applied to BT application, because of the overhead incurred by the copies pair that wraps the computation. In the case of FT, the performance is improved when the CC technique is implemented. The average gain is 5%. A careful examination of the FT results shows that although there is an improvement in the total time, the overhead incurred by the copies pair is unacceptable.

The CC approach is based on copy-to and copy-from private arrays. In the case of BT, the data slice is larger than in the case of FT and thus the overhead in BT is higher. One can notice that for 4 processors in BT the computation time of the shared-case is less than the private-case, due to low cache reuse when the working-slice is larger. The cache effect is also the reason why the copy-inside(write to private array)takes about two times more than copy-back(read from private array).

The private arrays are allocated on the stack. The default size of the SGI Origin 2000 is 16MB per thread, more than enough for our experiments. A recent survey of the OpenMP community shows various default stack sizes among the HPC vendors, as shown in Table 2. The values shown here are enough for high performance OpenMP applications.

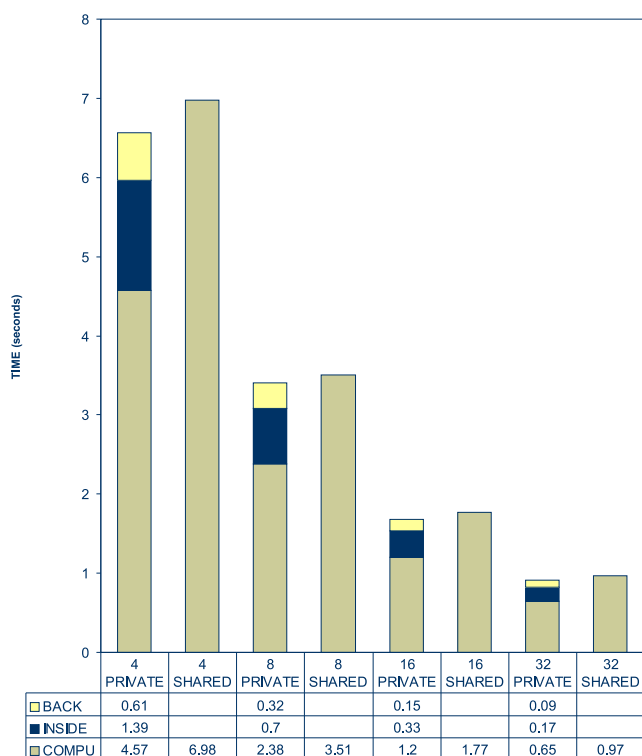


Figure 8. FT Running Results.

Privatization is a fundamental feature of OpenMP. Clauses like PRIVATE, THREADPRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, and COPYPRIVATE are using copy operations directly or indirectly. Thus, fast copy mechanism of large data chunks is essential for large-scale OpenMP applications.

5 Related Work

The efforts to cope with the data locality problem in OpenMP are focused in two directions. The explicit approach recommends to use compiler directives, while the

Vendor	32 Bit (MBytes)	64 Bit (MBytes)
SGI	16	16
IBM	4	4
Compaq	5	5
Sun	4	8
HP	8	8
Intel	1	1
KAI	1	1

Table 2. Default Stack Size from Various Vendors

implicit approach offers run time systems that take care of data distribution in a transparent fashion.

The best solution to the problem of allocating data and threads in an OpenMP program would be to implement a transparent, and highly optimized, dynamic migration of data. However, it is very hard for the operating system to determine when to migrate data. Dimitrios et al. [3] claim that due to the low remote-local memory access latency ratio of contemporary NUMA architectures, reasonably balanced page placement schemes, such as round-robin or random distribution, incur modest performance losses. They present a transparent, user-level page migration engine with an ability to gain back any performance loss that stems from sub-optimal placement of pages in iterative OpenMP programs.

SGI, Compaq and PGI provide high level directives to specify data distribution and thread scheduling in OpenMP programs [1, 2, 4]. A major component in SGI and Compaq directives is the DISTRIBUTION directive. This specifies the manner in which a data object is mapped onto the system memories. Three distribution kinds, namely BLOCK, CYCLIC and *, are available to specify the distribution required for each dimension of an array. They also offer the DISTRIBUTION_RESHAP directive to perform data distribution at element granularity. Both vendors supply directives to associate computations with the location of data in storage. Compaq provides the NUMA directive to indicate that the iterations of the immediately following PARALLEL DO loop are to be assigned to threads in a NUMA-aware manner. The ON HOME directive informs the compiler exactly how to distribute iterations over memories, and ALIGN is used for specifying alignment of data. Similarly, SGI provides AFFINITY, a directive that can be used to specify the distribution of loop iterations based on either DATA or THREAD affinity. In addition to these, both vendors have added lower-level directives to directly influence the location of pages in memory such as MIGRATE_NEXT_TOUCH and PLACE_NEXT_TOUCH. PGI has different execution model of its HPF-like data distribution directives. PGI relies on one-side communication or MPI libraries to communicate among the nodes that may contain more than one processor since PGI mainly targets distributed memory systems.

The diversity of the approaches used by the vendors and the different syntax harm the portability and the simplicity of the model.

Applying SPMD style of programming to OpenMP applications was proposed in [5, 6, 7]. Under the SPMD strategy, the arrays were distributed evenly among the processors and converted the local part into array that is private to each thread. One or more shared buffers are created to exchange data as needed between the threads. This is most efficiently performed by extending the size of each private array to include the "shadow" regions, as is realized via SHADOW directive in HPF. The programmer must explicitly synchronize reading and writing of buffer data. Thus

each thread works on its private data, and sharing is enabled through small-shared buffers. The running results on the SGI Origin 2000 show considerable improvement of the performance.

6 Conclusions

OpenMP programming model is an important step towards standardization of programming in shared memory and distributed-shared memory environments. The main challenge of OpenMP is still open: how to overcome the data locality problem without hurting the portability and simplicity of the model.

This paper presents a simple approach called Copy-Inside-Copy-Back(CC) which enables SPMD style of programming and a technique for data placement and replacement without taking the programming control and flexibility from the developer. The technique was explained, implemented and tested using two test cases, the BT and FT NAS benchmarks.

Unfortunately, the experiments show that currently CC approach is appropriate only for coarse grain parallelism where the computation time to copies-pair time ratio justifies in using CC approach.

Privatization is a fundamental feature of OpenMP. Clauses like PRIVATE, THREADPRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, and COPYPRIVATE are using copy operations directly or indirectly. Thus, fast copy mechanism of large data chunks is essential for large-scale OpenMP applications.

References

- [1] Silicon Graphics Inc *Parallel Processing on Origin Series SystemsMIPSpro 7 Fortran 90 Commands and Directives Reference Manual*. techpubs.sgi.com
- [2] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson and C. D. Offner. *Extending OpenMP for NUMA machines*. Scientific Programming. Vol. 8, No. 3, 2000.
- [3] D. S. Nicolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta and E. Ayguad. *A Transparent Runtime Data Distribution Engine*. Scientific Programming. Vol. 8, No. 3, pp. 143-162, 2000.
- [4] J. Merlin, D. Miles, V. Schuster. *Distributed OMP: Extensions to OpenMP for SMP clusters*. In EWOMP 2000, Second European Workshop on OpenMP, Murrayfield Conference Centre, Edinburgh, Scotland, U.K., September 14- 15, 2000.
- [5] B. Chapman, A. Patil, A. Prabhakar. *Performance Oriented Programming for NUMA Architectures*. WOM-PACT 2001, Purdue University, West Lafayette, Indiana. July30-31, 2001

- [6] B. Chapman, F. Bregier, A. Patil and A. Prabhakar. *Achieving Performance under OpenMP on ccNUMA and Software Distributed Shared Memory Systems*. Concurrency and Computation: Practice and Experience, 14:1- 17, 2002
- [7] B. Chapman, O. Hernandez, A. Patil, and A. Prabhakar. *Program Development Environment for OpenMP Programs on ccNUMA Architectures*. In the Third International Conference, LSSC 2001, Sozopol, Bulgaria, June 2001. pp. 210-217.
- [8] H. Jin, M. Frumkin and J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. NAS Technical Report NAS-99-011, October 1999.
- [9] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface*. Version 2.0, November 2000.
- [10] Barbara Chapman, Piyush Mehrotra and Hans Zima, *Enhancing OpenMP with Features for Locality Control*. To appear.
- [11] 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