# SC10 HPC Challenge Submission for XcalableMP

Jinpil Lee
*Graduate School of Systems and Information Engineering*
*University of Tsukuba*
*jinpil@hpcs.cs.tsukuba.ac.jp*

Masahiro Nakao
*Center for Computational Sciences*
*University of Tsukuba*
*mnakao@ccs.tsukuba.ac.jp*

Mitsuhisa Sato
*Center for Computational Sciences*
*University of Tsukuba*
*msato@cs.tsukuba.ac.jp*

## I. Introduction

XcalableMP[1], XMP for short, is a directive-based language extension for distributed memory systems, which is proposed by the XMP Specification Working Group which consists of members from academia, Research Lab and Industries mainly from Japan. It allows users to easily develop parallel programs for distributed memory systems and to tune performance with minimal and simple notation. A part of the design is based on the experiences of HPF(High Performance Fortran)[5], Fujitsu XPF (VPP FORTRAN)[6], and OpenMPD[7]. The specification is available at the web site of XcalableMP[1].

In this submission of this year, the highlights are as follows:

- We demonstrate the programmability and the expressiveness of XMP, that is, how to write efficient parallel programs easily in XMP. In this submission, we take three benchmarks, RandomAccess, HPL and FFT in HPCC Benchmarks, and CG in NAS Parallel Benchmarks[12], Laplace solver using simple Jacobi iteration. The Laplace solver is an simple example to use shadow and reflect directives which communicates and synchronize the overlapped regions. In the CG benchmark, we show the technique of two-dimensional parallelization using replicated data data distribution. It should be noted that the parallelized codes are almost compatible to the sequential version by ignoring the XMP directives.
- We report several experimental results with different sizes and different machines, T2K-Tsukuba system and CRAY XT5. In some benchmarks, we also show the MPI performance for comparison, showing that XMP performance is comparable to that of the MPI.
- We consider an extension of XMP for multicore nodes. The loop directives can be extended to describe multi-threaded execution in each nodes, by just adding simple clauses, as well as parallelization between nodes. While this extension was applied to the Laplace solver, unfortunately we could not obtain effective results this time.

## II. Overview of XcalableMP

In XMP, a programmer adds directives to a serial source code, as in OpenMP[2], to describe data/task parallelism and inter-node communication.

The features of XMP are as follows:

- XMP is defined as a language extension for familiar languages, such as C and Fortran, to reduce code-rewriting and educational costs.
- XcalableMP supports typical parallelization based on the data parallel paradigm and work mapping under ”global view programming model”, and enables parallelizing the original sequential code using minimal modification with simple directives, like OpenMP. Many ideas on ”global-view” programming are inherited from HPF (High Performance Fortran).
- The important design principle of XcalableMP is”performance-awareness”. All actions of communication and synchronization are taken by directives, different from automatic parallelizing compilers. The user should be aware of what happens by XcalableMP directives in the execution model on the distributed memory architecture.
- XcalableMP also includes a CAF-like PGAS (Partitioned Global Address Space) feature as ”local view” programming.

### A. Execution Model

The target of XMP is a distributed memory system. Each compute node, which may have several cores that share the main memory, has its own local memory, and each node is connected via a network. The compute node is the execution unit referred to as a “node” in XMP. Like MPI[8], the execution model of XMP is the Single Program Multiple Data (SPMD) model. In each node, a single thread starts program execution from the same main routine. By default, data declared in the program is allocated in each node and is referenced locally by threads executed in the node. When the thread encounters XMP directives, synchronization and communication occurs between nodes. In other words, no synchronization or communication occurs without directives. In this case, the program performs duplicate executions of the same program on local memory in

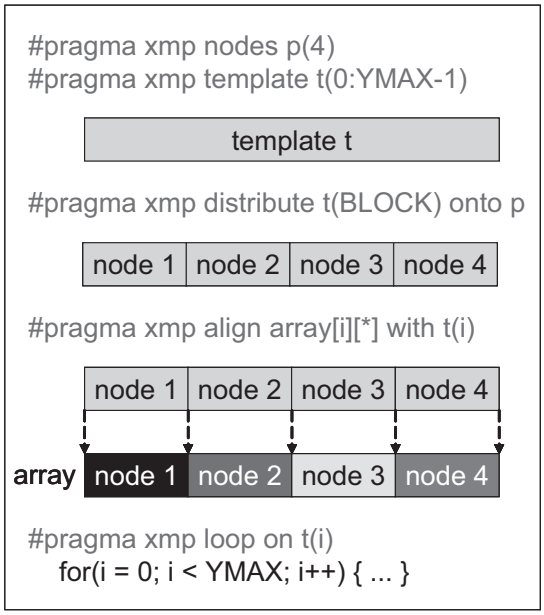


Figure 1. Data Distribution Using Template

```
01: int array[N];
02:
03: #pragma xmp nodes p(4)
04: #pragma xmp template t(0:N-1)
05: #pragma xmp distribute t(BLOCK) on p
06: #pragma xmp align array[i] with t(i)
07:
08: main(void) {
09:  int i, j, res = 0;
10:
11: #pragma xmp loop on t(i)
12:  for (i = 0; i < YMAX; i++) {
13:     array[i] = func(i);
14:     res += array[i];
15:  }
16:
17: #pragma xmp reduction (+:res)
18: }
```

Figure 2. Example of Global-view Model Programming

each node. XMP directives are used to specify the synchronization and communication explicitly whenever the programmer wants to communicate and synchronize data between nodes. XMP supports two models for viewing data: the global-view programming model and the local-view programming model. In the local-view model, accesses to data in remote nodes are performed explicitly by language extension for remote memory access with the node number of the target nodes, whereas reference to local data is executed implicitly.

### *B. Global-View Programming Model*

In the global-view programming model, programmers express their algorithm and data structure as a whole, mapping them to the node set. Programmers describe the data distribution and the work mapping to express how to distribute data and share work among nodes. The variables in the global-view model appear as a shared memory spanning nodes.

*1) Template and Data Distribution:* The global-view programming model supports typical communications for parallel programs. First, to parallelize the program in the global-view model, the data distribution for each global data is described. A template is used to specify the data distribution. The template is a dummy array used to express an index space associated with an array. Fig. 1 shows the concept of a template. The node directive declares the node array. The distribution of the template is described by a distribute directive; in this case, the distribute directive specifies the block distribution of the template. Finally, the data distribution of an array is specified by aligning the array to the template by align directives.

*2) Loop Construct and Parallelization of Loop:* The loop construct maps work iteratively on the node at which the computed data is located. The template is also used to describe the iteration space of the loop. A simple example of the global-view programming model of the C language is shown in Fig. 2. Line 3 defines the "node set" (four nodes in this case). Line 4 defines the template with the lower limit (0) and the upper limit (N – 1) of the index space. Line 5 specifies the distribution of the template on the node set as a block distribution, and Line 6 defines the distribution of the array by aligning the array with the template. For the distribution, block, cyclic, and gen-block distributions are supported. Line 11 is the loop construct for parallelizing the loop of statements from Line 12 to Line 15. In this case, Node 1 executes the iterations from 0 to N/4 – 1, and Node 2 executes the iterations from N/4 to N/2 – 1, independently. Line 17 executes the reduction operation on the variable *res* by the reduction directive. The reduction directive supports typical operators (such as "+" and " MAX").

*3) Communication in Global-View:* Global-view communication directives are used to synchronize nodes, maintain the consistency of a shadow area, and move distributed data globally.

The gmove construct is a powerful operation in global-view programming in XMP. The gmove construct copies the data of a distributed array. This directive is followed by the assignment statement of scalar value and array sections. The assignment

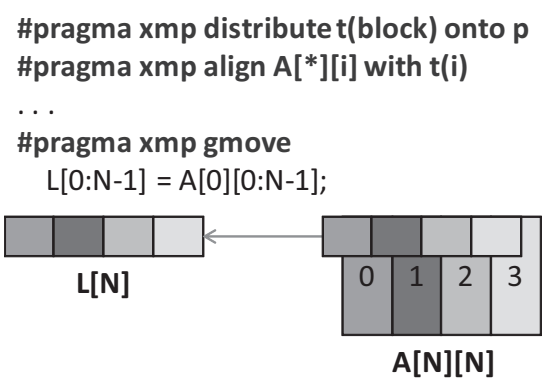


Figure 3. Example of Gmove Directive

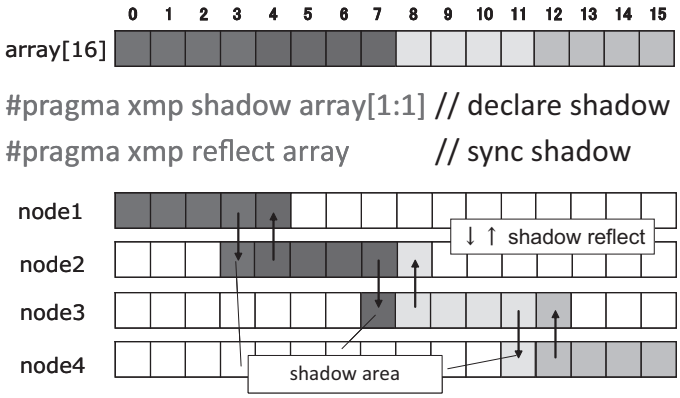


Figure 4. Example of Shadow and Reflect Directives

operation of the array sections of a distributed array may require communication between nodes. In XMP, the C language is extended to support array section notation in order to support the assignment of array objects. Fig. 3 shows a sample code of the gmove directive. In this example, an array *A* distributed on three nodes is gathered in a local array *B*.

The shadow directive specifies the shadow width for the area used to communicate the neighbor element of a block of a distributed array. The data stored in the storage area declared by the shadow directive is referred to as the shadow object. The reflect directive assigns the value of a reflection source to a shadow object for variables having the shadow attribute. Among the data allocated to a storage area other than a shadow area, the data representing the same array element as that of the shadow object is referred to as the reflection source of the shadow object. Fig. 4 shows an example of the shadow and reflect directives.

For collective communications, barrier, reduction, and broadcast operations are provided by the directives.

- #pragma xmp bcast *var* [, *var*...] [from *nodes-ref*] [on *nodes-ref* | *template-ref*]
  The "bcast" construct executes broadcast communication from one node.
- #pragma xmp barrier [on *nodes-ref* | *template-ref*]
  The "barrier" construct specifies an explicit barrier at the point.

The global-view programming model is useful when, starting from the sequential version of the program, the programmer parallelizes the program in the data-parallel model by adding directives incrementally with minimum modifications. Since these directives can be ignored as a comment by the compilers of base languages (C and Fortran), an XMP program derived from a sequential program can preserve the integrity of the original program when the program is run sequentially. The global-view model shares a number of major conceptual similarities with HPF.

### *C. Local-View Programming Model*

The local-view programming model attempts to increase performance by considering inter-node communication and the local memory of each node explicitly. The local data distributed on the node can be referred to using the node number.

XMP adopts coarray notation as an extension of languages for local-view programming. In the case of Fortran as the base language, most coarray notation is compatible to that of CAF, except that the task constructs are used for task parallelism. For example, in Fortran, to access an array element of *A(i)* located on computation node *N*, the expression of *A(i)[N]* is used. If the access is a value reference, then communication to obtain the value occurs. If the access updates the value, then communication to set a new value occurs.

To use coarray notation in C, we propose a language extension of the language. A coarray is declared by the coarray directive in C.

- #pragma xmp coarray *array-variable co-array-dimension*

An example of the local-view programming model of the C language is shown in Fig. 5.

The coarray object is referenced in the following expression:

- scalar-variable : [image-index]
- array-section-expression : [image-index]

To access a coarray, the node number (image-index) is specified by following an array reference by ":" Array section notation is notation to describe part of an array and is adopted in Fortran90. In C, we extend C to have an array section of the following form:

```
int array[N];

#pragma xmp coarray array[*]

array[a:b]:[1] = tmp[c:d];
```

Figure 5. Example of Local-view Model Programming

```
#pragma xmp loop on t(i) reduction(+:sum) threads private(x)
for(i = 0; i < N; i++) {
  x = func(..., i);
  . . .
  sum += x;
}
```

⟶

```
xmp_sched(&lb, &ub, &s, ...);
#pragma omp parallel for reduction(+:temp_sum) private(x)
for (i = lb; i < ub, i += s) {
  x = func(..., i);
  . . .
  temp_sum += x;
}
xmp_reduce(&sum, &temp_sum);
```

Figure 6. Sample Code of Threads Clause

- array_name [*lower_bound* : *upper_bound* : *step* ][...]

Either the lower bound or the upper bound can be omitted in the index range of a section, in which case the lower or upper bound defaults, respectively, to the lowest or highest values taken by the index of the array. Thus, *A[:]* is a section containing the whole of *A*.

Fig. 5 shows that the elements from *c* to *d* of array *tmp* are copied to form *a* to *b* of array *array* in node 1.

## III. Hybrid Parallel Programming extension in XcalableMP

One of interesting issues is hybrid programming on SMP clusters, because SMP clusters are now common platforms in HPC. SMP clusters have several multicore processor(s) sharing the local memory. To obtain the performance, we need to exploit resources inside a node. XMP has the similar programming model with OpenMP, directive-based approach. It is not difficult to allow OpenMP directives in XMP source codes.

In the loop specified *loop* directive, each iteration can be executed independently in each nodes since data is private in each node. To execute each iteration in parallel within the node, some variables should be declared as private to a thread in the nodes. The equivalent data-scope clauses of OpenMP are used in *loop* directive in XMP to do this.

Fig. 6 shows a sample code using a thread-private variable in XMP. After *threads* clause in loop directive, you can write OpenMP clauses such as *private*, *firstprivate* to declare thread-private variables. The *reduction* clause does not appear after *threads* clause because XMP compiler can generate it from the *reduction* clause in *loop* directive. Loop iteration, reduction variables are thread-private by default.

Note that if the users want to parallelize the inner loop by threads in the node, OpenMP can directly be used to parallelize the loop.

## IV. Performance Evaluation

We parallelized 5 benchmarks including RandomAccess, HPL and FFT in HPCC Benchmarks[9], CG in NAS Parallel Benchmarks[12], Laplace solver using simple Jacobi iteration. We used T2K-Tsukuba system[10] and Cray Inc. XT5[13] for evaluation. Table. I shows the node configuration of T2K-Tsukuba system and XT5. Each benchmark used 64 physical system nodes of T2K-Tsukuba system, and 16 physical system nodes of XT5 . In this evaluation, we assigned XMP processes(equivalent to MPI processes) to a physical node, a physical socket and a core. Moreover, we set the memory size of three(called Small, Middle, Large ). We used CLOC[11] by Al Danial to count Lines-Of-Code(LOC).

Table I
T2K TSUKUBA SYSTEM & XT5 NODE CONFIGURATIONS

| System | T2K Tsukuba System | XT5 |
|---|---|---|
| CPU | AMD Opteron Quad-core 8000 series 2.3Ghz x 4 sockets (16 cores) | AMD Opteron Shanghai 2.7 GHz x 2 sockets (8 cores) |
| Memory | 32GB | 32GB |
| Network | Infiniband DDR (4 rails) | SeaStar 3D-Torus interconnect |
| OS | Linux kernel 2.6.18 x86_64 | CLE 3.1 pre-release (based on Linux 2.6.27.48) |
| MPI | MVAPICH2 1.2 | Cray MPT(Message Passing Toolkit) 5.1 (based on MPICH2) |

```
#pragma xmp nodes p(NPCOL, NPROW)
#pragma xmp template t(0:XSIZE-1, 0:YSIZE-1)
#pragma xmp distribute t(block, block) onto p
#pragma xmp align u[j][i] with t(i, j)
#pragma xmp align uu[j][i] with t(i, j)
#pragma xmp shadow uu[1:1][1:1]
...

#pragma xmp loop (x, y) on t(y, x)
   for(x = 1; x < XSIZE; x++)
      for(y = 1; y < YSIZE; y++)
        uu[x][y] = u[x][y];

#pragma xmp reflect uu

#pragma xmp loop (x, y) on t(y, x)
for(x = 1; x < XSIZE; x++)
      for(y = 1; y < YSIZE; y++)
        u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
```

Figure 7. Parallel Code of Laplace

### A. Laplace solver

The Laplace solver is an implementation of Laplace equation using simple Jacobi iteration with 4 points-stencil operations in two dimensional block distribution. The LOC of Laplace is 100. We added 20 directives to parallelize Laplace. Fig. 7 shows an outline of parallelization of Laplace. This program is taken to demonstrate parallelization by shadow and reflect directives. The overlapped region between nodes are updated by using shadow and reflect directives. Note that the parallelization is very simple and straightforward from its sequential version.

Fig. 8 shows the performance of Laplace in GFlops. The result is verified by checking the residual after the fixed number of iterations.

The numbers of *u[][]* and *uu[][]* elements of Small size are 4,096, those of Middle size are 4,096 $\times$ 4 and those of Large size are 4,096 $\times$ 16. In Size large, the Laplace benchmark requires more than memory of 4 nodes. The results for Laplace indicate that the performance of XMP is similar to that of MPI.

### B. CG Benchmark

In the CG benchmark, the conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. We have implemented two-dimensional parallelization(equivalent to original MPI CG).

The LOC of CG is 545. We added 41 directives to parallelize CG. Fig. 9 shows an outline of two-dimensional parallelization of CG. The two-dimensional template is declared and distributed in a two-dimensional block distribution on the nodes. The one-dimensional vectors are aligned to one dimension of the template and are replicated in other dimensions in the two-dimensional node array. For example, the array w is distributed from p(*,1) to p(*,NPROW), where "*" stands for any number from 1 to NPCOL. This means that the array is replicated in the first dimension of nodes array *p*.

The subscript "*" in the template of the align directive and the loop directive specifies replication or replicated execution. In the figure, the first loop construct specifies align to t(*,j), which means that the iterations of this loop are mapped on the first dimension t(*,j). Moreover, index arrays *rowstr* and *colidx* are calculated to refer array *a* before entering the loop in each node. This operation enables the double loop sentence to be calculated and is equivalent to the MPI version of CG.

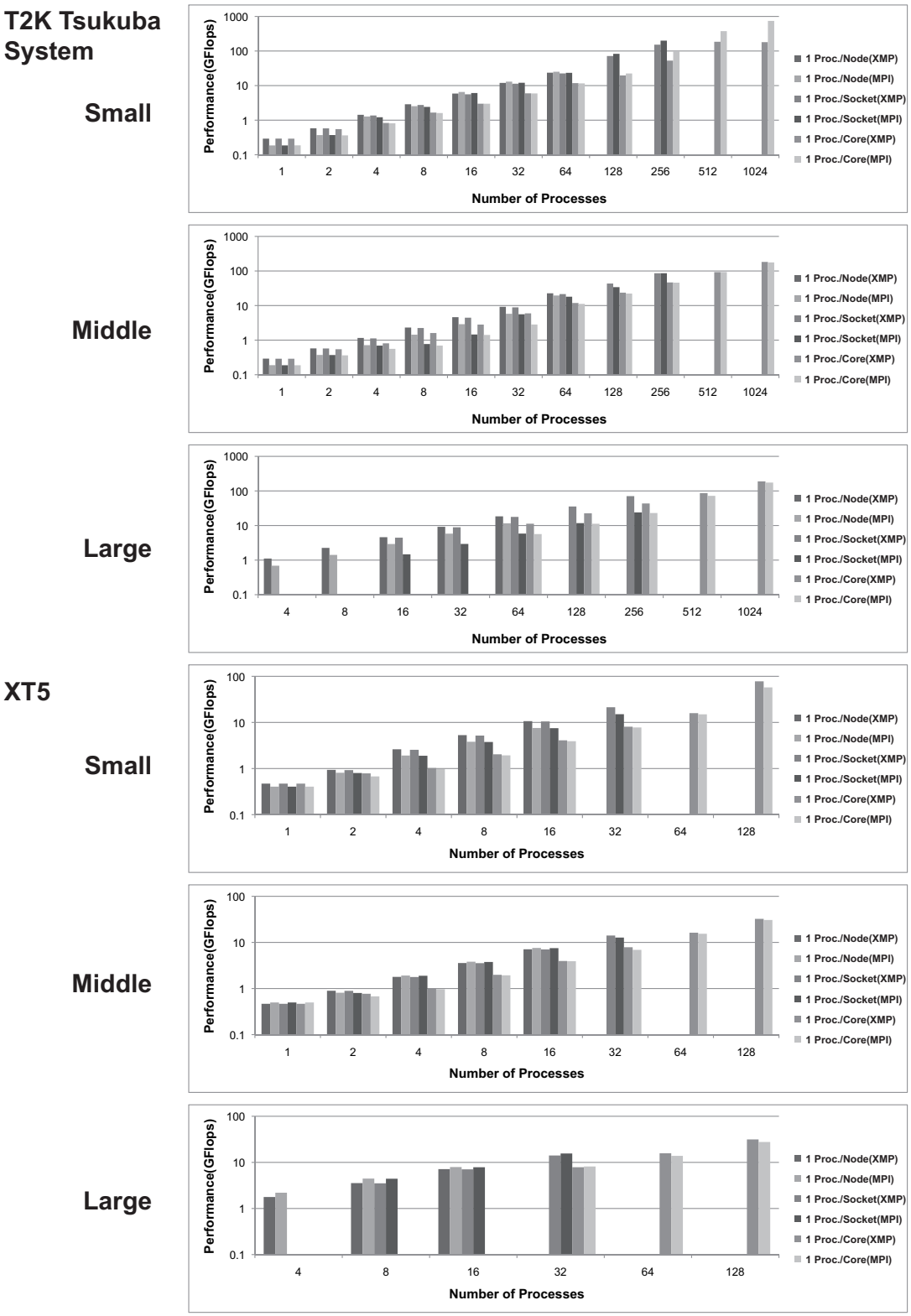


Figure 8. Performance of Laplace

```
#pragma xmp nodes pros(NPCOL, NPROW)
#pragma xmp template t(0:na+1, 0:na+1)
#pragma xmp distribute t(block, block) onto pros
#pragma xmp align x[i], z[i], p[i], q[i], r[i] with t(i, *)
#pragma xmp align w[i] with t(*, i)
        . . .
#pragma xmp loop on t(*, j)
  for (j = 1; j <= lastrow-firstrow+1; j++) {
    sum = 0.0;
    for (k = rowstr[j]; k < rowstr[j+1]; k++) {
      sum = sum + a[k]*p[colidx[k]];
    }
    w[j] = sum;
  }

#pragma xmp reduction(+:w) on pros(*, :)

#pragma xmp gmove
  q[:] = w[:];
        . . .
```

Figure 9. Parallel Code of CG

This replication rule is applied to the reduction operation. The following directive executes the reduction operation on distributed array *w*. The symbol ":" specifies the entire set of indices.

```
#pragma xmp reduction(+:w) on p(*,:)
```

The "gmove" directive copies arrays *w* and *q* in their entirety with different distributions.

Fig. 10 shows the performance of CG. The results are verified by the rule given by NPB.

Small size is CLASS A defined by NAS Parallel Benchmarks, Middle size is CLASS B and that of Large size is CLASS C. The results for CG indicate that the performance of XMP is similar to that of MPI.

### C. FFT Benchmark

FFT benchmark measures the floating point rate of execution for double precision complex 1-dimensional Discrete Fourier Transform. Fig. 11 shows the parallel code of FFT. The six-step FFT algorithm is used as in the MPI version. The main function invokes *zff1d* function. *in* and *out* are passed to *a* and *b*. Those 1-dimensional vectors are accessed as 2-dimensional matrices in *zff1d0* function. In *zff1d0* function, 1-dimensional FFT is performed on each dimension of the matrix. Therefore, matrix transpose operations are required during the calculation.

We parallelized *zff1d* and *zff1d0* function in global view model. Local *align* directives are describing distribution of parameter arrays. We distributed them along the second dimension in a block manner because FFT is done along the first dimension. Two templates of different sizes are used to distribute *a* and *b*. Because the block size is same($((N1 \times N2) \div p) = ((N2 \div p) \times N1) = ((N1 \div p) \times N2)$), the original 1-dimensional vectors can be accessed as 2-dimensional matrices in the parallel version. In six-step FFT, matrix transpose operation is done before 1-dimensional FFT. The matrix transpose is implemented by local memory copy between *a* and *b* in the sequential code. In the parallel version, the matrix transpose operation is implemented by the *gmove* directive and local memory copy. Fig. 12 shows how matrix transpose *a* to *b* is processed on node 1. The number is the node number where the block is allocated, and dotted lines show how the matrices are distributed. Since the distribution manner is different, node 1 does not have all the elements of matrix *a* which are needed for the transpose. At first, a *gmove* is written to collect those elements. A new array *a_work* is declared to store the elements. *a_work* is distributed by *t1* which was used to distribute *b*. Consequently, the local block of *a_work* and *b* have the same shape. By the all-to-all communication of the *gmove* directive, all elements needed for transpose are stored in local buffer. So we can copy it to *b* using the loop statement [1]. And then, FFTE routine(FFT235) is used for 1-dimensional FFT. The rest of the code is simple work-sharing parallelizing loop statements. The LOC of FFT is 217. We added 31 directives to parallelize FFT.

[1]we cannot describe the transpose only by the *gmove* directive because of the syntax of the array section statement

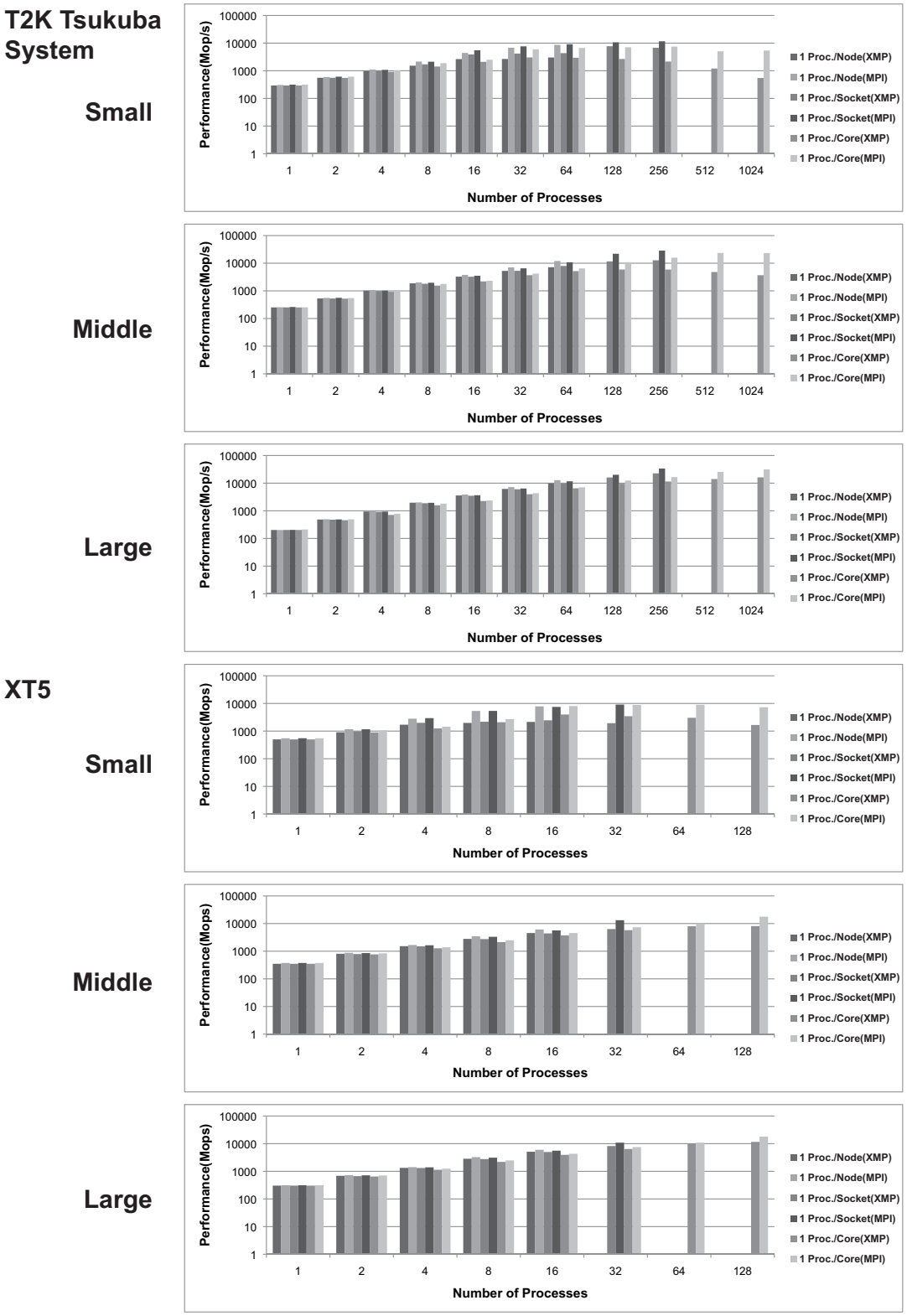


Figure 10. Performance of CG

```
#pragma xmp nodes p(*)
#pragma xmp template t0(0:(N1*N2)-1)
#pragma xmp template t1(0:N1-1)
#pragma xmp template t2(0:N2-1)
#pragma xmp distribute (BLOCK) onto p :: t0, t1, t2
fftw_complex in[N1*N2], out[N1*N2], a_work[N2][N1];
#pragma xmp align [i] with t0(i) :: in, out
#pragma xmp align a_work[*][i] with t1(i)
. . .
int zfft1d(fftw_complex a[N], fftw_complex b[N], ...) {
#pragma xmp align [i] with t0(i) :: a, b
  . . . zfft1d0((fftw_complex **)a, (fftw_complex **)b, ...); . . .
}
. . .
void zfft1d0(fftw_complex a[N2][N1], fftw_complex b[N1][N2], ...) {
#pragma xmp align a[i][*] with t2(i)
#pragma xmp align b[i][*] with t1(i)
  . . .
#pragma xmp gmove
  a_work[:][:] = a[:][:];
#pragma xmp loop on t1(i)
  for (i = 0; i < N1; i++)
    for (j = 0; j < N2; j++)
      c_assgn(b[i][j], a_work[j][i]);
#pragma xmp loop on t1(i)
  for(i = 0; i < N1; i++) HPCC_fft235(b[i], work, w2, N2, ip2);
  . . .
}
```

Figure 11. Parallel Code of FFT

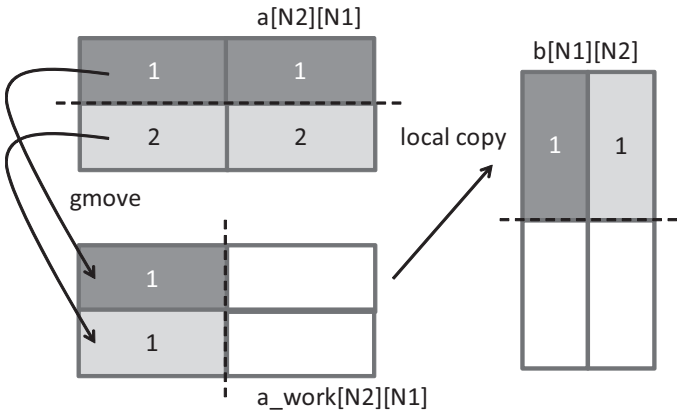


Figure 12. Matrix Transpose in FFT

Fig. 13 shows the performance of FFT. The numbers of *N1* and *N2* elements of Small size are 1,024, those of Middle size are 1,024 $\times$ 2 and those of Large size are 1,024 $\times$ 4. The most of the overhead is all-to-all communication by the *gmove* directive. We have to improve *gmove* runtime functions get better performance. Or, users can describe overlapping gmove communication with local copy using co-array functions.

### *D. Linpack Benchmark*

Linpack benchmark measures the floating point rate of execution for solving a dense system of linear equations. Fig. 14 shows the parallel code of Linpack. We parallelized simple sequential Linpack routines(*dgefa* and *dgesl*) in global view model. Matrix *a* is distributed along the first dimension in a cyclic manner. Pivot vector *pvt_v* is an *N* sized local array duplicated on each node.

When a function processes distributed arrays(distributed by the global view directives), the function should also be parallelized. For example, *A_daxpy* in Fig. 14 calculates $dy = dyda \times dx$. *dx* and *dy* are pointers referring arrays. When the referred arrays are distributed, the loop statement should be parallelized not to access unallocated area of the arrays. The *align* directive is often used in global scope to describe that the target (global) array is distributed. And the compiler creates the array descriptor and reallocates the target array. In a function's local scope, the *align* directive can be used for the function's parameters. It tells that the target parameter array is distributed in global scope (we assume that the target array is already distributed by the *align* directive in global scope), and creates the array descriptor which is used to parallelize the function. In *A_daxpy*, the descriptor is used to parallelize the loop. The size of parameter arrays should be written explicitly to create the array descriptor. Current prototype compiler only allows a constant number for the parameter size. We are fixing this to allow variable length arrays as parameters.

The main issue on parallelizing Linpack is exchanging the pivot vector. On iteration *k*, row *l* is selected as a pivot vector and row *k* is exchanged with row *l*. This is a typical operation of Gaussian elimination with partial pivoting. The problem is matrix *a* is distributed among nodes, that is, the row exchange requires inter-node communication. We used the *gmove* directive to describe the communication. The first *gmove* directive copies the *l*-th row to *pvt_v*. Then, the owner of row *l* broadcasts row data to *pvt_v*. The second *gmove* directive generates send/recv communication. The owner of row *l* receives

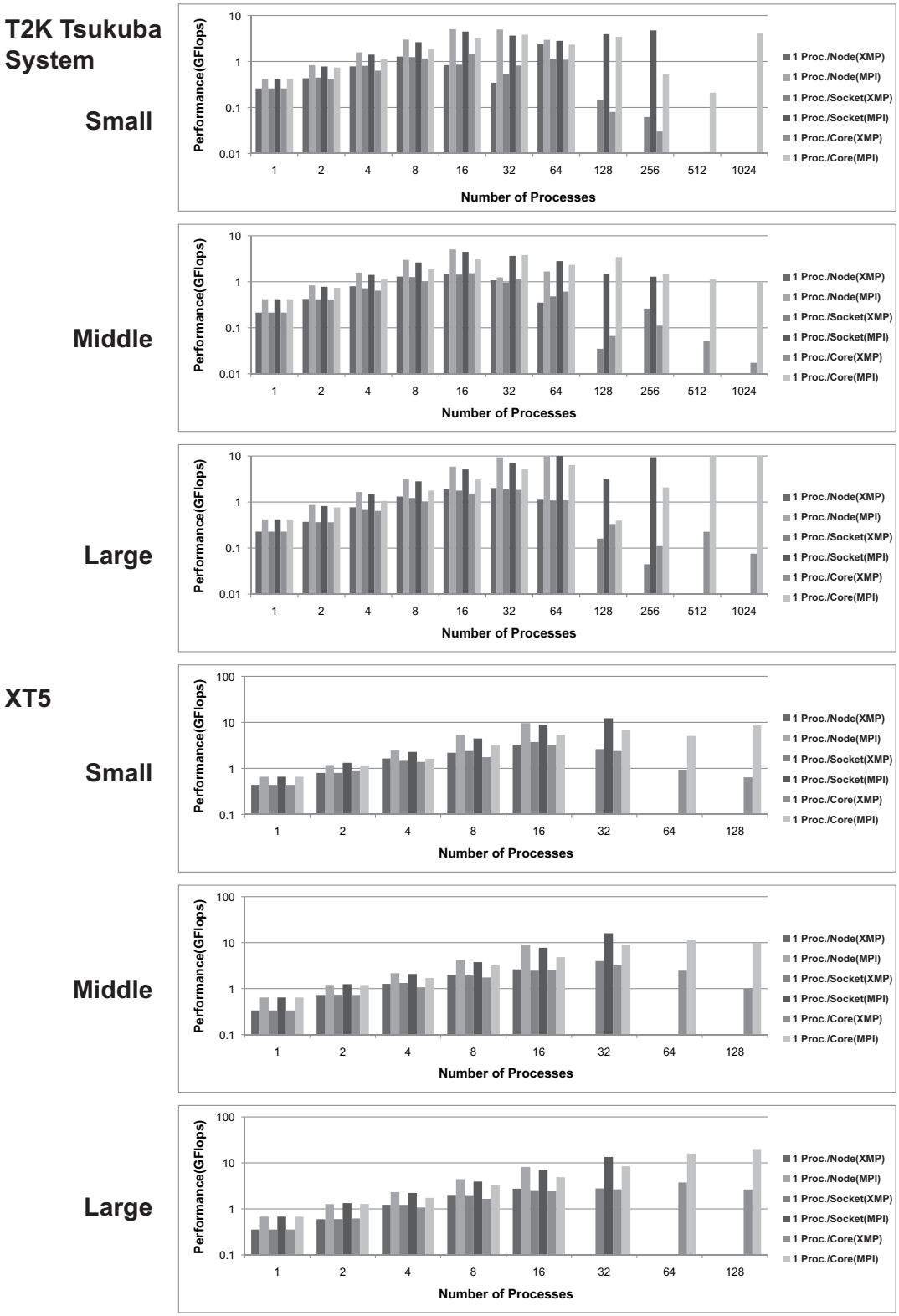


Figure 13. Performance of FFT

row *k* from its owner. The third *gmove* directive is a local memory copy operation. The owner of row *k* copies *pvt_t* to row *k*. Consequently, row exchanging is completed using the pivot buffer. Those communications are generated by the compiler with the *gmove* directive descriptions, and users do not need to consider the owner node of each data. The LOC of Linpack is 243. We added 35 directives to the sequential code to parallelize.

```
#pragma xmp nodes p(*)
#pragma xmp template t(0:N-1)
#pragma xmp distribute t(CYCLIC) onto p
double a[N][N], pvt_v[N];
#pragma xmp align a[*][i] with t(i)
. . .
void dgefa(double a[N][N], int n, int ipvt[N]) {
#pragma xmp align a[*][i] with t(i)
  . . .
  for (k = 0; k < nm1; k++) {
    . . .
#pragma xmp gmove
    pvt_v[k:n-1] = a[k:n-1][l];
    if (l != k) {
#pragma xmp gmove
      a[k:n-1][l] = a[k:n-1][k];
#pragma xmp gmove
      a[k:n-1][k] = pvt_v[k:n-1];
    }
    . . .
    for (j = kp1; j < n; j++) {
      t = pvt_v[j];
      A_daxpy(k+1, n-(k+1), t, a[k], a[j]);
    }
    . . .
}
void A_daxpy(int b, int n, double da, double dx [N], double dy[N]) {
#pragma xmp align [i] with t(i) :: dx, dy
  . . .
#pragma xmp loop on t(i)
  for (i = b; i < b+n; i ++)
    dy[i] = dy[i] + da*dx[i];
}
```

Figure 14. Parallel Code of Linpack

Fig. 15 shows the performance of Linpack. The number of *a[]* elements of Small size is 1,024, that of Middle size is 1,024 $\times$ 4 and that of Large size is 1,024 $\times$ 16. The performance is not satisfying. One of the reasons is its simple parallelization scheme, 1-dimensional array distribution. Another reason of bad scalability is communication in *dgesl* function, diagonal elements of matrix *a* should be broadcasted when solving *b*. Although the performance is not that good, Linpack can be described with a few directives and we can optimize the performance based on this parallel code. 2-dimensional array distribution and cache blocking will be the next step to achieve better performance.

### *E. RandomAccess Benchmark*

RandomAccess benchmark measures the performance of random integer updates of memory. The measurement is Giga UPdates per Second(GUPS). Fig. 16 shows the parallel code of RandomAccess. It updates arbitrary array elements for each iteration. The array update is written like the following statement in the sequential version.

- Table[temp] ^= temp;

When an element to update is allocated on a remote node, the operation should be done using inter-node communication. Message passing with send/recv is not suitable for describing this kind of applications. The MPI version of RandomAccess creates a message handler processing remote update requests with long and complicated descriptions of MPI functions such as *MPI_Isend()*, *MPI_Irecv()* and *MPI_Test()*. Fundamentally, this is remote memory access. RandomAccess can be easily described with one-sided communication support.

We parallelized RandomAccess in the local view model. In the local view model, data distribution is more explicit than the global view model. The updated array *Table* is divided by the number of nodes. We manually redefined the size of *Table* in the source code, and *SIZE* sized array is allocated on each node. Only the local *Table* can be accessed using local indices now. To enable remote memory access, we declared *Table* as a co-array by describing the *co-array* directive. RandomAccess create a random index *temp*(The range is from 0 to (TABLE_SIZE - 1)) on each iteration. The information for remote memory access is manually calculated and used in the co-array notation. ((temp%TABLE_SIZE)/SIZE) indicates the node number to access, and (temp%SIZE) indicates the local index of *temp* on the target node. Consequently, accumulation(BIT XOR) to *Table*[*temp*] can be done remotely. Remote memory access with co-array notation is asynchronous in XMP. Barrier synchronization is taken to complete requested remote memory access(*barrier* directive). This is for the performance evaluation. The LOC of RandomAccess is 77. We added 4 directives and modified 3 lines(array declaration and update) to the sequential code to parallelize RandomAccess.

Fig. 17 shows the performance of RandomAccess. The number of *Table[]* elements in Small size of 131,072 and that of Large size is 131,072 $\times$ 64. In this experimentation, we used from 2 to 32 processes because XMP execution required a lot of time. We couldn't achieve good performance on RandomAccess, and it shows bad scalability. The most of the overhead is barrier of remote memory access processing a large number of asynchronous messages. We are using MPI-2 functions

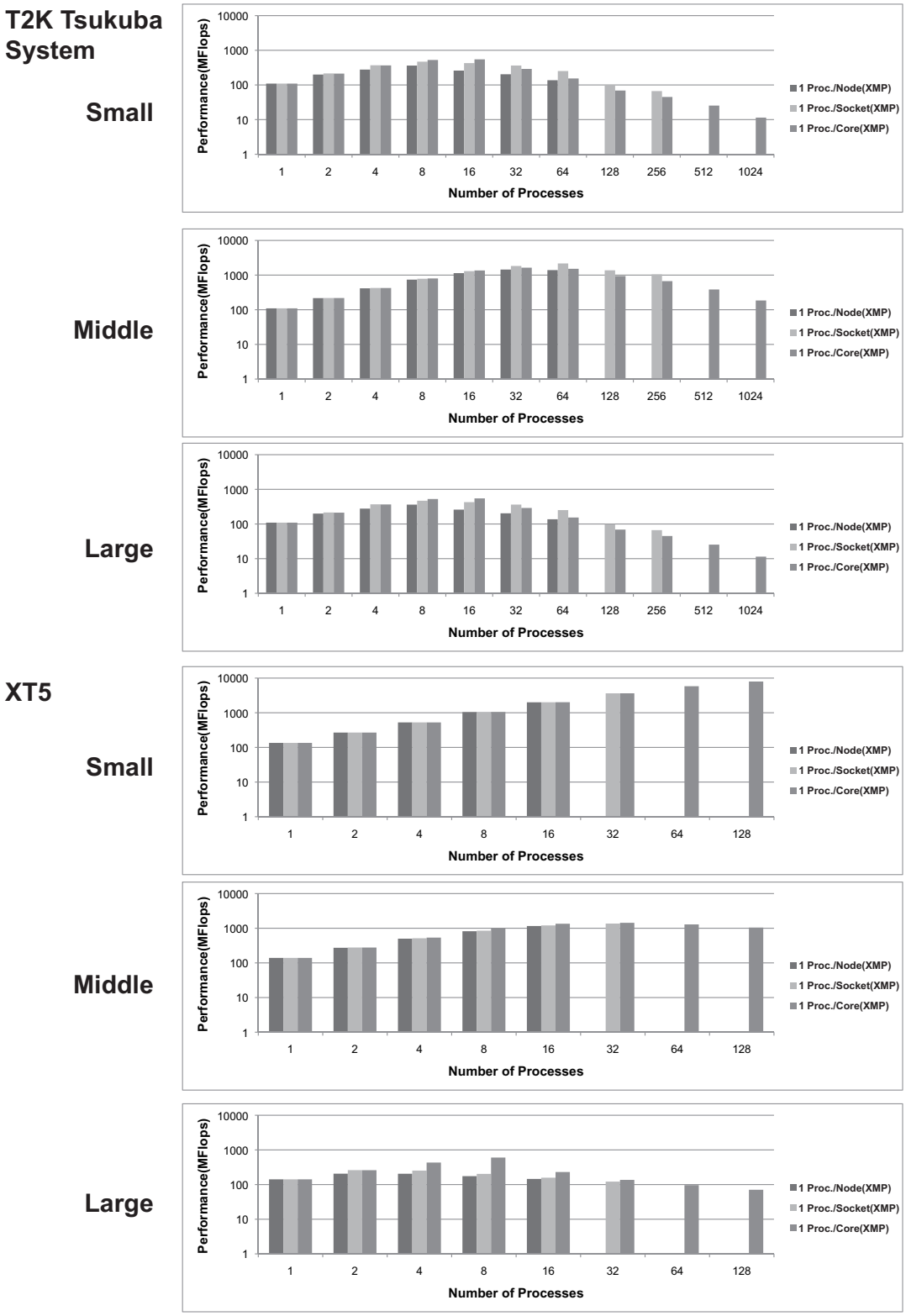


Figure 15. Performance of Linpack

```
#define SIZE TABLE_SIZE/PROCS
u64Int Table[SIZE] ;
#pragma xmp nodes p(PROCS)
#pragma xmp coarray Table [PROCS]
. . .
for (i = 0; i < SIZE; i++) Table[i] = b + i ;
. . .
for (i = 0; i < NUPDATE; i++) {
  temp = (temp << 1) ^ ((s64Int)temp < 0 ? POLY : 0);
  Table[temp%SIZE]:[(temp%TABLE_SIZE)/SIZE] ^= temp;
}
#pragma xmp barrier
```

Figure 16. Parallel Code of RandomAccess

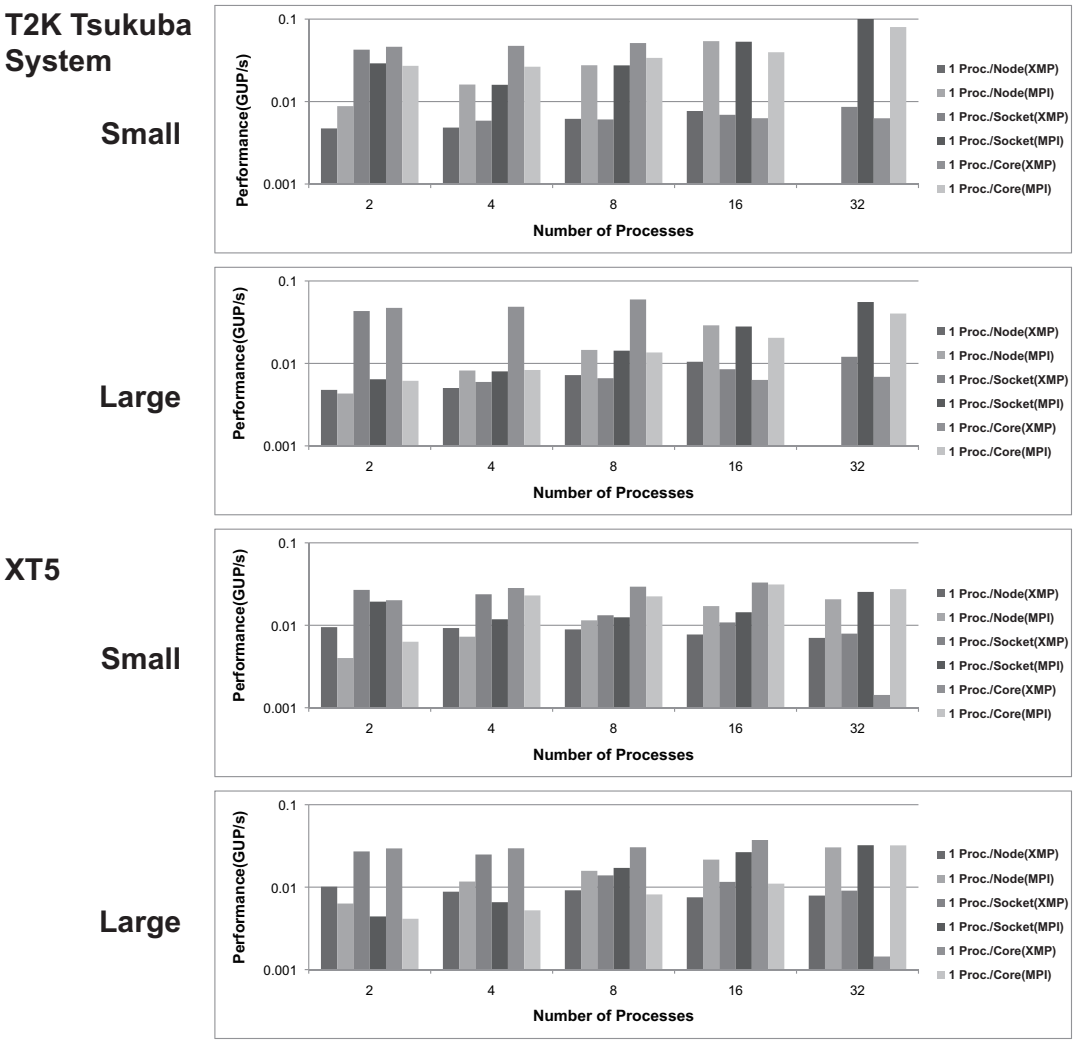


Figure 17. Performance of RandomAccess

```
#pragma xmp loop (x, y) on t(y, x) threads private(y)
   for(x = 1; x < XSIZE; x++)
      for(y = 1; y < YSIZE; y++)
        uu[x][y] = u[x][y];

#pragma xmp reflect uu

#pragma xmp loop (x, y) on t(y, x) threads private(y)
for(x = 1; x < XSIZE; x++)
      for(y = 1; y < YSIZE; y++)
        u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
```

Figure 18. Parallel Code of Hybrid Parallel Laplace

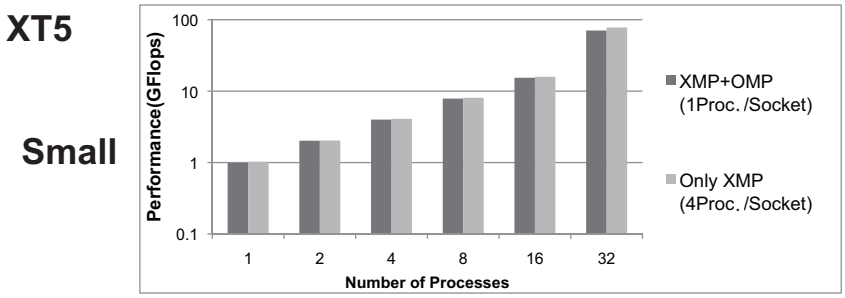


Figure 19. Performance of Hybrid Parallel Laplace

such as *MPI_Get()*, *MPI_Put()* and *MPI_Fence()* to implement one-sided communication; the co-array notations and barriers are translated those function calls. This benchmark shows the performance of remote memory access, and it tells we need more efficient implementation of one-sided communication.

### *F. Hybrid Laplace solver*

Here, we parallelized Laplace in Section IV-A in thread-level using the XMP extension which is introduced in Section III. Fig. 18 shows the hybrid parallel code of Laplace. Each loop statement with *loop* directive is parallelized in thread-level by the XMP compiler. Because the loop iteration counter *y* has to be thread-private, we added *private* cluase after *threads* clause (*x* is declared as a thread-private variable by XMP compiler).

Fig. 19 shows the performance of Hybrid Parallel Laplace which is evaluated on Cray XT5. Each XT5 node has 2 quad-core processors(socket). We executed a XMP process(equivalent to a MPI process) with 4 threads on each socket for Hybrid Parallel Laplace. In this experimentation, the performance did not change compared to the flat-MPI(4 XMP processes on each socket). This result is reasonable because the more the Laplace uses processes, the less communication between processes is.

## REFERENCES


[1] XcalableMP, http://www.xcalablemp.org/

[2] OpenMP, http://www.openmp.org/

[3] Unified Parallel C, http://upc.gwu.edu/

[4] Co-Array Fortran, http://www.co-array.org/

[5] High Performance Fortran, http://hpff.rice.edu/

[6] Yuanyuan Zhang, Hidetoshi Iwashita, Kuninori Ishii, Masanori Kaneko, Tomotake Nakamura and Kohichiro Hotta, ”Hybrid Parallel Programming on SMP Clusters using XPFortran”, Proceedings of the 6th International Workshop on OpenMP, pp.133-148, 2010.

[7] Jinpil Lee, Mitsuhisa Sato and Taisuke Boku, ”OpenMPD: A Directive Based Data Parallel Language Extensions for Distributed Memory Systems”, Proceedings of the 37th International Conference on Parallel Processing, pp.121-128, 2008.

[8] Message Passing Interface (MPI) Forum, http://www.mpi-forum.org/

[9] High Performance Computing Challenge Benchmark, http://icl.cs.utk.edu/hpcc/

[10] T2K Open Supercomputer, http://www.open-supercomputer.org/

[11] CLOC, http://cloc.sourceforge.net/

[12] Bailey, D.H. and et al.THE NAS PARALLEL BENCHMARKS, Technical Report NAS-94-007,Nasa Ames Research Center (1994)

[13] http://www.cray.com/Products/XT/Systems/XT5.aspx

## APPENDIX

- Source Code of Laplace

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <time.h>
4 #include <stdlib.h>
5 #ifdef _XCALABLEMP
6 #include <xmp.h>
7 #endif
8
9 #define XSIZE (4096)
10 #define YSIZE XSIZE
11 #define PI M_PI
12 double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
13
14 #pragma xmp nodes p(NPCOL, NPROW)
15 #pragma xmp template t(0:XSIZE-1, 0:YSIZE-1)
16 #pragma xmp distribute t(block, block) onto p
17 #pragma xmp align u[j][i] with t(i, j)
18 #pragma xmp align uu[j][i] with t(i, j)
19 #pragma xmp shadow uu[1:1][1:1]
20 double time;
21 int rank, niter;
22
23 static void lap_main(void);
24 static void verify(void);
25
26 int main(){
27   int x, y;
28
29 #ifdef _XCALABLEMP
30   rank = xmp_get_rank();
31 #else
32   rank = 0;
33 #endif
34
35   /* initalize */
36 #pragma xmp loop (x, y) on t(y, x)
37   for(x = 1; x < XSIZE-1; x++)
38     for(y = 1; y < YSIZE-1; y++)
39       u[x][y] = sin((double)(x-1)/XSIZE*PI) + cos((double)(y-1)/YSIZE*PI);
40
41 #pragma xmp task on t(0, *)
```

```
42   {
43 #pragma xmp loop on t(*, x)
44     for(x = 0; x < (XSIZE); x++){
45       u[x][0] = 0.0;
46       uu[x][0] = 0.0;
47     }
48   }
49
50 #pragma xmp task on t((YSIZE)-1, *)
51   {
52 #pragma xmp loop on t(*, x)
53     for(x = 0; x < (XSIZE); x++){
54       u[x][YSIZE-1] = 0.0;
55       uu[x][YSIZE-1] = 0.0;
56     }
57   }
58
59 #pragma xmp task on t(*, 0)
60   {
61 #pragma xmp loop on t(y, *)
62     for(y = 0; y < (YSIZE); y++){
63       u[0][y] = 0.0;
64       uu[0][y] = 0.0;
65     }
66   }
67
68 #pragma xmp task on t(*, (XSIZE)-1)
69   {
70 #pragma xmp loop on t(y, *)
71     for(y = 0; y < (YSIZE); y++){
72       u[XSIZE-1][y] = 0.0;
73       uu[XSIZE-1][y] = 0.0;
74     }
75   }
76
77   time = -xmp_get_second();
78   lap_main();
79   time += xmp_get_second();
80   verify();
81
82 #pragma xmp reduction(MAX:time) on p
83   if(rank == 0){
84     fprintf(stderr, "Per. = %.3f GFlops\n",
85     (double)niter*(XSIZE-2)*(YSIZE-2)*5/time/1000/1000/1000);
86   }
87
88   return 0;
89 }
90
91 void lap_main(void){
92   int x, y, k;
93   double sum;
94
95         for(k = 0; k < niter; k++){
```

```
96
97        /* old <- new */
98 #pragma xmp loop (x, y) on t(y, x)
99      for(x = 1; x < XSIZE-1; x++)
100           for(y = 1; y < YSIZE-1; y++)
101               uu[x][y] = u[x][y];
102
103 #pragma xmp reflect uu
104
105       /* update */
106 #pragma xmp loop (x, y) on t(y, x)
107       for(x = 1; x < XSIZE-1; x++)
108             for(y = 1; y < YSIZE-1; y++)
109                 u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
110
111   } // end of niter
112 }
113
114 void verify(void){
115   int x, y;
116   double sum = 0.0;
117
118 #pragma xmp loop (x, y) on t(y, x) reduction(+:sum)
119   for(x = 1; x < XSIZE-1; x++)
120     for(y = 1; y < YSIZE-1; y++)
121       sum += (uu[x][y]-u[x][y]);
122
123   if(rank == 0){
124     fprintf(stderr, "Verification Value = %f\n", sum);
125   }
126
127 }
```

- Source Code of CG

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #ifdef _XCALABLEMP
5 #include <xmp.h>
6 #endif
7
8 #define NZ       NA*(NONZER+1)*(NONZER+1)+NA*(NONZER+2)
9 #define NA       75000
10 #define NONZER   13
11 #define NITER    75
12 #define SHIFT    60.0
13 #define RCOND    1.0e-1
14 #define COMPILETIME "30 Jun 2010"
15 #define NPBVERSION "2.3"
16 #define CS1 "cc"
17 #define CS2 "cc"
18 #define CS3 "(none)"
19 #define CS4 "-I../common"
20 #define CS5 "-O3 "
```

```
21 #define CS6 "(none)"
22 #define CS7 "randdp"
23
24 #define NUM_COL 4
25 #define NUM_ROW 2
26 #pragma xmp nodes pros(NUM_COL, NUM_ROW)
27 #pragma xmp template t(0:NA,0:NA)
28 #pragma xmp distribute t(block,block) onto pros
29 #pragma xmp align x[i] with t(i,*)
30 #pragma xmp align z[i] with t(i,*)
31 #pragma xmp align p[i] with t(i,*)
32 #pragma xmp align q[i] with t(i,*)
33 #pragma xmp align r[i] with t(i,*)
34 #pragma xmp align w[i] with t(*,i)
35
36 static int naa, nzz, firstrow, lastrow, firstcol, lastcol;
37 static int colidx[NZ+1];        /* colidx[1:NZ] */
38 static int rowstr[NA+1+1];      /* rowstr[1:NA+1] */
39 static int iv[2*NA+1+1];        /* iv[1:2*NA+1] */
40 static int arow[NZ+1];          /* arow[1:NZ] */
41 static int acol[NZ+1];          /* acol[1:NZ] */
42 static double v[NA+1+1];        /* v[1:NA+1] */
43 static double aelt[NZ+1];       /* aelt[1:NZ] */
44 static double a[NZ+1];          /* a[1:NZ] */
45 static double x[NA+1], z[NA+1], p[NA+1], q[NA+1], r[NA+1], w[NA+1];
46 static double amult;
47 static double tran;
48 static int rank, size;
49 static double tmp;
50 static int tmp_i[NA+1];
51
52 static void conj_grad(int colidx[], int rowstr[], double x[], double z[], double a[],
53                       double p[], double q[], double r[], double w[], double *rnorm);
54 static void makea(int n, int nz, double a[], int colidx[], int rowstr[],
55                   int nonzer, int firstrow, int lastrow, int firstcol,
56                   int lastcol, double rcond, int arow[], int acol[],
57                   double aelt[], double v[], int iv[], double shift );
58 static void sparse(double a[], int colidx[], int rowstr[], int n,
59                    int arow[], int acol[], double aelt[], int firstrow, int lastrow,
60                    double x[], boolean mark[], int nzloc[], int nnza);
61 static void sprnvc(int n, int nz, double v[], int iv[], int nzloc[], int mark[]);
62 static int icnvrt(double x, int ipwr2);
63 static void vecset(int n, double v[], int iv[], int *nzv, int i, double val);
64 static void setup_submatrix_info();
65 extern void timer_clear(int);
66 extern void timer_start(int);
67 extern void timer_stop(int);
68 extern double timer_read(int);
69
70 /*--------------------------------------------------------------------
71       program cg
72 --------------------------------------------------------------------*/
73 int main(int argc, char **argv) {
74     int i, j, k, it;
```

```
75      int nthreads = 1;
76      double zeta;
77      double rnorm;
78      double norm_temp11, norm_temp12;
79      double t, mflops;
80      char class;
81      boolean verified;
82      double zeta_verify_value, epsilon;
83
84  #ifdef _XCALABLEMP
85      rank = xmp_get_rank();
86      size = xmp_get_size();
87  #else
88      rank = 0;
89      size = 1;
90  #endif
91
92  #pragma xmp loop on t(i, *)
93      for(i=0;i<NA;i++){
94        x[i] = 0.0; z[i] = 0.0;
95        p[i] = 0.0; q[i] = 0.0;
96        r[i] = 0.0;
97      }
98
99  #pragma xmp loop on t(*, i)
100     for(i=0;i<NA;i++){
101       w[i] = 0.0;
102     }
103
104     firstrow = 1;
105     lastrow  = NA;
106     firstcol = 1;
107     lastcol  = NA;
108
109     if (NA == 1400 && NONZER == 7 && NITER == 15 && SHIFT == 10.0) {
110         class = 'S';
111         zeta_verify_value = 8.5971775078648;
112     } else if (NA == 7000 && NONZER == 8 && NITER == 15 && SHIFT == 12.0) {
113         class = 'W';
114         zeta_verify_value = 10.362595087124;
115     } else if (NA == 14000 && NONZER == 11 && NITER == 15 && SHIFT == 20.0) {
116         class = 'A';
117         zeta_verify_value = 17.130235054029;
118     } else if (NA == 75000 && NONZER == 13 && NITER == 75 && SHIFT == 60.0) {
119         class = 'B';
120         zeta_verify_value = 22.712745482631;
121     } else if (NA == 150000 && NONZER == 15 && NITER == 75 && SHIFT == 110.0) {
122         class = 'C';
123         zeta_verify_value = 28.973605592845;
124     } else if(NA == 1500000 && NONZER == 21 && NITER == 100 && SHIFT == 500.0){
125         class = 'D';
126         zeta_verify_value = 52.514532105794;
127     } else {
128         class = 'U';
```

```
129     }
130
131     if(rank == 0){
132       printf("\n\n NAS Parallel Benchmarks 2.3 OpenMP C version"
133              " - CG Benchmark\n");
134       printf(" Size: %10d\n", NA);
135       printf(" Iterations: %5d\n", NITER);
136     }
137
138     naa = NA;
139     nzz = NZ;
140
141 /*--------------------------------------------------------------------
142 c  Initialize random number generator
143 c-------------------------------------------------------------------*/
144     tran    = 314159265.0;
145     amult   = 1220703125.0;
146     zeta    = randlc( &tran, amult );
147
148 /*--------------------------------------------------------------------
149 c
150 c-------------------------------------------------------------------*/
151     setup_submatrix_info();
152
153     makea(naa, nzz, a, colidx, rowstr, NONZER,
154           firstrow, lastrow, firstcol, lastcol,
155           RCOND, arow, acol, aelt, v, iv, SHIFT);
156
157 /*---------------------------------------------------------------------
158 c  Note: as a result of the above call to makea:
159 c        values of j used in indexing rowstr go from 1 --> lastrow-firstrow+1
160 c        values of colidx which are col indexes go from firstcol --> lastcol
161 c        So:
162 c        Shift the col index vals from actual (firstcol --> lastcol )
163 c        to local, i.e., (1 --> lastcol-firstcol+1)
164 c---------------------------------------------------------------------*/
165 #pragma xmp loop on t(*, j)
166   for (j = 1; j <= NA; j++) {
167     tmp_i[j] = rowstr[j-firstrow+1];
168   }
169   tmp_i[j] = rowstr[j-firstrow+1];
170
171 #pragma xmp loop on t(*, j)
172   for (j = 1; j <= NA; j++)
173     rowstr[j] = tmp_i[j];
174   rowstr[j] = tmp_i[j];
175
176 /*--------------------------------------------------------------------
177 c  set starting vector to (1, 1, .... 1)
178 c-------------------------------------------------------------------*/
179 #pragma xmp loop on t(i,*)
180   for (i = 1; i <= NA; i++) {
181         x[i] = 1.0;
182     }
```

```
183
184       zeta  = 0.0;
185
186 /*---------------------------------------------------------------------
187 c
188 c  Do one iteration untimed to init all code and data page tables
189 c                        (then reinit, start timing, to niter its)
190 c---------------------------------------------------------------------*/
191       for (it = 1; it <= 1; it++) {
192
193 /*---------------------------------------------------------------------
194 c  The call to the conjugate gradient routine:
195 c---------------------------------------------------------------------*/
196           conj_grad( colidx, rowstr, x, z, a, p, q, r, w, &rnorm );
197
198 /*---------------------------------------------------------------------
199 c  zeta = shift + 1/(x.z)
200 c  So, first: (x.z)
201 c  Also, find norm of z
202 c  So, first: (z.z)
203 c---------------------------------------------------------------------*/
204           norm_temp11 = 0.0;
205           norm_temp12 = 0.0;
206
207 #pragma xmp loop on t(j, *) reduction(+:norm_temp11,norm_temp12)
208   for (j = 1; j <= NA; j++) {
209     norm_temp11 = norm_temp11 + x[j]*z[j];
210     norm_temp12 = norm_temp12 + z[j]*z[j];
211   }
212
213   norm_temp12 = 1.0 / sqrt( norm_temp12 );
214
215 /*---------------------------------------------------------------------
216 c  Normalize z to obtain x
217 c---------------------------------------------------------------------*/
218 #pragma xmp loop on t(j,*)
219   for (j = 1; j <= NA; j++) {
220     x[j] = norm_temp12*z[j];
221   }
222
223       } /* end of do one iteration untimed */
224
225 /*---------------------------------------------------------------------
226 c  set starting vector to (1, 1, .... 1)
227 c---------------------------------------------------------------------*/
228 #pragma xmp loop on t(i,*)
229       for (i = 1; i <= NA+1; i++) {
230           x[i] = 1.0;
231       }
232
233       zeta  = 0.0;
234
235       timer_clear( 1 );
236       timer_start( 1 );
```

```
237
238 /*---------------------------------------------------------------------
239 c--->
240 c  Main Iteration for inverse power method
241 c--->
242 c---------------------------------------------------------------------*/
243     for (it = 1; it <= NITER; it++) {
244
245 /*---------------------------------------------------------------------
246 c  The call to the conjugate gradient routine:
247 c---------------------------------------------------------------------*/
248         conj_grad( colidx, rowstr, x, z, a, p, q, r, w, &rnorm);
249
250 /*---------------------------------------------------------------------
251 c  zeta = shift + 1/(x.z)
252 c  So, first: (x.z)
253 c  Also, find norm of z
254 c  So, first: (z.z)
255 c---------------------------------------------------------------------*/
256         norm_temp11 = 0.0;
257         norm_temp12 = 0.0;
258
259 #pragma xmp loop on t(j, *) reduction(+:norm_temp11,norm_temp12)
260  for (j = 1; j <= NA; j++) {
261    norm_temp11 = norm_temp11 + x[j]*z[j];
262    norm_temp12 = norm_temp12 + z[j]*z[j];
263  }
264
265  norm_temp12 = 1.0 / sqrt( norm_temp12 );
266  zeta = SHIFT + 1.0 / norm_temp11;
267
268  if( rank == 0 ){
269     if( it == 1 ) {
270       printf("   iteration           ||r||                 zeta\n");
271     }
272     printf("    %5d       %20.14e%20.13e\n", it, rnorm, zeta);
273   }
274
275 /*---------------------------------------------------------------------
276 c  Normalize z to obtain x
277 c---------------------------------------------------------------------*/
278 #pragma xmp loop on t(j,*)
279 for (j = 1; j <= NA; j++) {
280   x[j] = norm_temp12*z[j];
281  }
282 } /* end of main iter inv pow meth */
283
284     timer_stop( 1 );
285
286 /*---------------------------------------------------------------------
287 c  End of timed section
288 c---------------------------------------------------------------------*/
289     t = timer_read( 1 );
290
```

```
291 if(rank == 0){
292     printf(" Benchmark completed\n");
293
294     epsilon = 1.0e-10;
295     if (class != 'U') {
296         if (fabs(zeta - zeta_verify_value) <= epsilon) {
297             verified = TRUE;
298             printf(" VERIFICATION SUCCESSFUL\n");
299             printf(" Zeta is     %20.12e\n", zeta);
300             printf(" Error is    %20.12e\n", zeta - zeta_verify_value);
301         } else {
302           verified = FALSE;
303           printf(" VERIFICATION FAILED\n");
304           printf(" Zeta                %20.12e\n", zeta);
305           printf(" The correct zeta is %20.12e\n", zeta_verify_value);
306         }
307       } else {
308         verified = FALSE;
309         printf(" Problem size unknown\n");
310         printf(" NO VERIFICATION PERFORMED\n");
311       }
312
313       if ( t != 0.0 ) {
314         mflops = (2.0*NITER*NA)
315           * (3.0+(NONZER*(NONZER+1)) + 25.0*(5.0+(NONZER*(NONZER+1))) + 3.0 )
316           / t / 1000000.0;
317       } else {
318         mflops = 0.0;
319       }
320
321       c_print_results("CG", class, NA, 0, 0, NITER, nthreads, t,
322                       mflops, "          floating point",
323                       verified, NPBVERSION, COMPILETIME,
324                       CS1, CS2, CS3, CS4, CS5, CS6, CS7);
325     }
326 }
327
328 static void setup_submatrix_info(){
329   int col_size, row_size;
330   int npcols = NUM_COL, nprows = NUM_ROW;
331   int proc_row = rank / npcols;
332   int proc_col = rank - proc_row * npcols;
333
334 /*---------------------------------------------------------------------
335   If naa evenly divisible by npcols, then it is evenly divisible
336   by nprows
337 ---------------------------------------------------------------------*/
338   col_size = NA / npcols;
339   firstcol = proc_col * col_size;
340   lastcol  = firstcol - 1 + col_size;
341   row_size = NA / nprows;
342   firstrow = proc_row * row_size;
343   lastrow  = firstrow - 1 + row_size;
344 }
```

```
345
346 /*---------------------------------------------------------------------
347 c---------------------------------------------------------------------*/
348 static void conj_grad (
349     int colidx[],       /* colidx[1:nzz] */
350     int rowstr[],       /* rowstr[1:naa+1] */
351     double x[NA],
352     double z[NA],
353     double a[],         /* a[1:nzz] */
354     double p[NA],
355     double q[NA],
356     double r[NA],
357     double w[NA],
358     double *rnorm )
359 /*---------------------------------------------------------------------
360 c---------------------------------------------------------------------*/
361
362 /*---------------------------------------------------------------------
363 c  Floaging point arrays here are named as in NPB1 spec discussion of
364 c  CG algorithm
365 c---------------------------------------------------------------------*/
366 {
367     static double d, sum, rho, rho0, alpha, beta;
368     int i, j, k;
369     int cgit, cgitmax = 25;
370     int count;
371
372 #pragma xmp align x[i] with t(i,*)
373 #pragma xmp align z[i] with t(i,*)
374 #pragma xmp align p[i] with t(i,*)
375 #pragma xmp align q[i] with t(i,*)
376 #pragma xmp align r[i] with t(i,*)
377 #pragma xmp align w[i] with t(*,i)
378
379     rho = 0.0;
380
381 /*---------------------------------------------------------------------
382 c  Initialize the CG algorithm:
383 c---------------------------------------------------------------------*/
384 #pragma xmp loop on t(j,*)
385     for (j = 1; j <= NA; j++) {
386       q[j] = 0.0;
387       z[j] = 0.0;
388       r[j] = x[j];
389       p[j] = r[j];
390     }
391
392 #pragma xmp loop on t(*,j)
393     for (j = 1; j <= NA; j++) {
394       w[j] = 0.0;
395     }
396
397 /*---------------------------------------------------------------------
398 c  rho = r.r
```

```
399 c  Now, obtain the norm of r: First, sum squares of r elements locally...
400 c---------------------------------------------------------------------*/
401 #pragma xmp loop on t(j,*) reduction(+:rho)
402     for (j = 1; j <= NA; j++) {
403         rho = rho + x[j]*x[j];
404     }
405
406 /*---------------------------------------------------------------------
407 c  The conj grad iteration loop
408 c---------------------------------------------------------------------*/
409     for (cgit = 1; cgit <= cgitmax; cgit++) {
410       rho0 = rho;
411       d = 0.0;
412       rho = 0.0;
413
414 /*---------------------------------------------------------------------
415 c  q = A.p
416 c  The partition submatrix-vector multiply: use workspace w
417 c---------------------------------------------------------------------*/
418 C
419 C  NOTE: this version of the multiply is actually (slightly: maybe %5)
420 C        faster on the sp2 on 16 nodes than is the unrolled-by-2 version
421 C        below.   On the Cray t3d, the reverse is true, i.e., the
422 C        unrolled-by-two version is some 10% faster.
423 C        The unrolled-by-8 version below is significantly faster
424 C        on the Cray t3d - overall speed of code is 1.5 times faster.
425 */
426 #pragma xmp loop on t(*, j)
427  for (j = 1; j <= NA; j++) {
428    sum = 0.0;
429    for (k = rowstr[j]; k < rowstr[j+1]; k++) {
430      sum = sum + a[k]*p[colidx[k]];
431    }
432    w[j] = sum;
433  }
434
435 #pragma xmp reduction(+:w) on pros(:,*)
436
437 #pragma xmp gmove
438        q[:] = w[:];
439
440 /*---------------------------------------------------------------------
441 c  Clear w for reuse...
442 c---------------------------------------------------------------------*/
443 #pragma xmp loop on t(*, j)
444  for (j = 1; j <= NA; j++) {
445    w[j] = 0.0;
446  }
447
448 /*---------------------------------------------------------------------
449 c  Obtain p.q
450 c---------------------------------------------------------------------*/
451 #pragma xmp loop on t(j, *)
452  for (j = 1; j <= NA; j++) {
```

```
453     d = d + p[j]*q[j];
454   }
455
456 #pragma xmp reduction(+:d) on pros(:,*)
457
458 /*---------------------------------------------------------------------
459 c   Obtain alpha = rho / (p.q)
460 c---------------------------------------------------------------------*/
461           alpha = rho0 / d;
462
463 /*---------------------------------------------------------------------
464 c   Save a temporary of rho
465 c---------------------------------------------------------------------*/
466           rho0 = rho;
467
468 /*---------------------------------------------------------------------
469 c   Obtain z = z + alpha*p
470 c   and    r = r - alpha*q
471 c---------------------------------------------------------------------*/
472 #pragma xmp loop on t(j, *)
473           for (j = 1; j <= NA; j++) {
474               z[j] = z[j] + alpha*p[j];
475               r[j] = r[j] - alpha*q[j];
476           }
477
478 /*---------------------------------------------------------------------
479 c   rho = r.r
480 c   Now, obtain the norm of r: First, sum squares of r elements locally...
481 c---------------------------------------------------------------------*/
482 #pragma xmp loop on t(j, *) reduction(+:rho)
483           for (j = 1; j <= NA; j++) {
484               rho = rho + r[j]*r[j];
485           }
486
487 /*---------------------------------------------------------------------
488 c   Obtain beta:
489 c---------------------------------------------------------------------*/
490           beta = rho / rho0;
491
492 /*---------------------------------------------------------------------
493 c   p = r + beta*p
494 c---------------------------------------------------------------------*/
495 #pragma xmp loop on t(j,*)
496           for (j = 1; j <= NA; j++) {
497               p[j] = r[j] + beta*p[j];
498           }
499       } /* end of do cgit=1,cgitmax */
500
501 /*---------------------------------------------------------------------
502 c   Compute residual norm explicitly:  ||r|| = ||x - A.z||
503 c   First, form A.z
504 c   The partition submatrix-vector multiply
505 c---------------------------------------------------------------------*/
506       sum = 0.0;
```

```
507
508 #pragma xmp loop on t(*, j)
509     for (j = 1; j <= NA; j++) {
510       d = 0.0;
511       for (k = rowstr[j]; k <= rowstr[j+1]-1; k++) {
512         d = d + a[k]*z[colidx[k]];
513       }
514       w[j] = d;
515     }
516
517 #pragma xmp reduction(+:w) on pros(:,*)
518
519 #pragma xmp gmove
520     r[:] = w[:];
521
522 /*---------------------------------------------------------------------
523 c  At this point, r contains A.z
524 c-------------------------------------------------------------------*/
525 #pragma xmp loop on t(j,*) reduction(+:sum)
526     for (j = 1; j <= NA; j++) {
527         d = x[j] - r[j];
528         sum = sum + d*d;
529     }
530
531     (*rnorm) = sqrt(sum);
532 }
533
534 /*---------------------------------------------------------------------
535 c       generate the test problem for benchmark 6
536 c       makea generates a sparse matrix with a
537 c       prescribed sparsity distribution
538 c
539 c       parameter    type        usage
540 c
541 c       input
542 c
543 c       n            i           number of cols/rows of matrix
544 c       nz           i           nonzeros as declared array size
545 c       rcond        r*8         condition number
546 c       shift        r*8         main diagonal shift
547 c
548 c       output
549 c
550 c       a            r*8         array for nonzeros
551 c       colidx       i           col indices
552 c       rowstr       i           row pointers
553 c
554 c       workspace
555 c
556 c       iv, arow, acol i
557 c       v, aelt        r*8
558 c-------------------------------------------------------------------*/
559 static void makea(
560     int n,
```

```
561     int nz,
562     double a[],          /* a[1:nz] */
563     int colidx[],        /* colidx[1:nz] */
564     int rowstr[],        /* rowstr[1:n+1] */
565     int nonzer,
566     int firstrow,
567     int  lastrow,
568     int firstcol,
569     int lastcol,
570     double rcond,
571     int arow[],          /* arow[1:nz] */
572     int acol[],          /* acol[1:nz] */
573     double aelt[],       /* aelt[1:nz] */
574     double v[],          /* v[1:n+1] */
575     int iv[],            /* iv[1:2*n+1] */
576     double shift )
577 {
578     int i, nnza, iouter, ivelt, ivelt1, irow, nzv;
579
580 /*--------------------------------------------------------------------
581 c       nonzer is approximately  (int(sqrt(nnza /n)));
582 c-------------------------------------------------------------------*/
583     double size, ratio, scale;
584     int jcol;
585
586     size = 1.0;
587     ratio = pow(rcond, (1.0 / (double)n));
588     nnza = 0;
589
590 /*--------------------------------------------------------------------
591 c  Initialize colidx(n+1 .. 2n) to zero.
592 c  Used by sprnvc to mark nonzero positions
593 c-------------------------------------------------------------------*/
594     for (i = 1; i <= n; i++) {
595         colidx[n+i] = 0;
596     }
597     for (iouter = 1; iouter <= n; iouter++) {
598         nzv = nonzer;
599         sprnvc(n, nzv, v, iv, &(colidx[0]), &(colidx[n]));
600         vecset(n, v, iv, &nzv, iouter, 0.5);
601         for (ivelt = 1; ivelt <= nzv; ivelt++) {
602             jcol = iv[ivelt];
603             if (jcol >= firstcol && jcol <= lastcol) {
604                 scale = size * v[ivelt];
605                 for (ivelt1 = 1; ivelt1 <= nzv; ivelt1++) {
606                     irow = iv[ivelt1];
607                     if (irow >= firstrow && irow <= lastrow) {
608                         nnza = nnza + 1;
609                         if (nnza > nz) {
610                             printf("Space for matrix elements exceeded in"
611                                    " makea\n");
612                             printf("nnza, nzmax = %d, %d\n", nnza, nz);
613                             printf("iouter = %d\n", iouter);
614                             exit(1);
```

```
615                         }
616                         acol[nnza] = jcol;
617                         arow[nnza] = irow;
618                         aelt[nnza] = v[ivelt1] * scale;
619                     }
620                 }
621             }
622         }
623         size = size * ratio;
624     }
625
626 /*-------------------------------------------------------------------------
627 c       ... add the identity * rcond to the generated matrix to bound
628 c           the smallest eigenvalue from below by rcond
629 c-------------------------------------------------------------------------*/
630     for (i = firstrow; i <= lastrow; i++) {
631         if (i >= firstcol && i <= lastcol) {
632             iouter = n + i;
633             nnza = nnza + 1;
634             if (nnza > nz) {
635                 printf("Space for matrix elements exceeded in makea\n");
636                 printf("nnza, nzmax = %d, %d\n", nnza, nz);
637                 printf("iouter = %d\n", iouter);
638                 exit(1);
639             }
640             acol[nnza] = i;
641             arow[nnza] = i;
642             aelt[nnza] = rcond - shift;
643         }
644     }
645
646 /*-------------------------------------------------------------------------
647 c       ... make the sparse matrix from list of elements with duplicates
648 c           (v and iv are used as  workspace)
649 c-------------------------------------------------------------------------*/
650     sparse(a, colidx, rowstr, n, arow, acol, aelt,
651            firstrow, lastrow, v, &(iv[0]), &(iv[n]), nnza);
652 }
653
654 /*---------------------------------------------------
655 c       generate a sparse matrix from a list of
656 c       [col, row, element] tri
657 c---------------------------------------------------*/
658 static void sparse(
659     double a[],         /* a[1:*] */
660     int colidx[],       /* colidx[1:*] */
661     int rowstr[],       /* rowstr[1:*] */
662     int n,
663     int arow[],         /* arow[1:*] */
664     int acol[],         /* acol[1:*] */
665     double aelt[],      /* aelt[1:*] */
666     int firstrow,
667     int lastrow,
668     double xx[],                /* xx[1:n] */
```

```
669     boolean mark[],      /* mark[1:n] */
670     int nzloc[],         /* nzloc[1:n] */
671     int nnza)
672 /*---------------------------------------------------------------------
673 c       rows range from firstrow to lastrow
674 c       the rowstr pointers are defined for nrows = lastrow-firstrow+1 values
675 c---------------------------------------------------------------------*/
676 {
677     int nrows;
678     int i, j, jajp1, nza, k, nzrow;
679     double xi;
680
681 /*---------------------------------------------------------------------
682 c    how many rows of result
683 c---------------------------------------------------------------------*/
684     nrows = lastrow - firstrow + 1;
685
686 /*---------------------------------------------------------------------
687 c      ...count the number of triples in each row
688 c---------------------------------------------------------------------*/
689     for (j = 1; j <= n; j++) {
690         rowstr[j] = 0;
691         mark[j] = FALSE;
692     }
693     rowstr[n+1] = 0;
694
695     for (nza = 1; nza <= nnza; nza++) {
696         j = (arow[nza] - firstrow + 1) + 1;
697         rowstr[j] = rowstr[j] + 1;
698     }
699
700     rowstr[1] = 1;
701     for (j = 2; j <= nrows+1; j++) {
702         rowstr[j] = rowstr[j] + rowstr[j-1];
703     }
704
705 /*---------------------------------------------------------------------
706 c      ... rowstr(j) now is the location of the first nonzero
707 c            of row j of a
708 c---------------------------------------------------------------------*/
709
710 /*---------------------------------------------------------------------
711 c      ... do a bucket sort of the triples on the row index
712 c---------------------------------------------------------------------*/
713     for (nza = 1; nza <= nnza; nza++) {
714         j = arow[nza] - firstrow + 1;
715         k = rowstr[j];
716         a[k] = aelt[nza];
717         colidx[k] = acol[nza];
718         rowstr[j] = rowstr[j] + 1;
719     }
720
721 /*---------------------------------------------------------------------
722 c       ... rowstr(j) now points to the first element of row j+1
```

```
c-------------------------------------------------------------------------*/
    for (j = nrows; j >= 1; j--) {
        rowstr[j+1] = rowstr[j];
    }
    rowstr[1] = 1;

/*--------------------------------------------------------------------
c       ... generate the actual output rows by adding elements
c-------------------------------------------------------------------*/
    nza = 0;

    for (i = 1; i <= n; i++) {
        xx[i] = 0.0;
        mark[i] = FALSE;
    }

    jajp1 = rowstr[1];
    for (j = 1; j <= nrows; j++) {
        nzrow = 0;

/*--------------------------------------------------------------------
c          ...loop over the jth row of a
c-------------------------------------------------------------------*/
        for (k = jajp1; k < rowstr[j+1]; k++) {
            i = colidx[k];
            xx[i] = xx[i] + a[k];
            if ( mark[i] == FALSE && xx[i] != 0.0) {
                mark[i] = TRUE;
                nzrow = nzrow + 1;
                nzloc[nzrow] = i;
            }
        }

/*--------------------------------------------------------------------
c          ... extract the nonzeros of this row
c-------------------------------------------------------------------*/
        for (k = 1; k <= nzrow; k++) {
            i = nzloc[k];
            mark[i] = FALSE;
            xi = xx[i];
            xx[i] = 0.0;
            if (xi != 0.0) {
                nza = nza + 1;
                a[nza] = xi;
                colidx[nza] = i;
            }
        }
        jajp1 = rowstr[j+1];
        rowstr[j+1] = nza + rowstr[1];
    }
}

/*--------------------------------------------------------------------
c       generate a sparse n-vector (v, iv)
```

```
c        having nzv nonzeros
c
c        mark(i) is set to 1 if position i is nonzero.
c        mark is all zero on entry and is reset to all zero before exit
c        this corrects a performance bug found by John G. Lewis, caused by
c        reinitialization of mark on every one of the n calls to sprnvc
c-----------------------------------------------------------------------*/
static void sprnvc(
    int n,
    int nz,
    double v[],         /* v[1:*] */
    int iv[],           /* iv[1:*] */
    int nzloc[],        /* nzloc[1:n] */
    int mark[] )        /* mark[1:n] */
{
    int nn1;
    int nzrow, nzv, ii, i;
    double vecelt, vecloc;

    nzv = 0;
    nzrow = 0;
    nn1 = 1;
    do {
        nn1 = 2 * nn1;
    } while (nn1 < n);

/*--------------------------------------------------------------------
c    nn1 is the smallest power of two not less than n
c-------------------------------------------------------------------*/
    while (nzv < nz) {
        vecelt = randlc(&tran, amult);

/*--------------------------------------------------------------------
c   generate an integer between 1 and n in a portable manner
c-------------------------------------------------------------------*/
        vecloc = randlc(&tran, amult);
        i = icnvrt(vecloc, nn1) + 1;
        if (i > n) continue;

/*--------------------------------------------------------------------
c  was this integer generated already?
c-------------------------------------------------------------------*/
        if (mark[i] == 0) {
            mark[i] = 1;
            nzrow = nzrow + 1;
            nzloc[nzrow] = i;
            nzv = nzv + 1;
            v[nzv] = vecelt;
            iv[nzv] = i;
        }
    }

    for (ii = 1; ii <= nzrow; ii++) {
        i = nzloc[ii];
```

```
831         mark[i] = 0;
832     }
833 }
834
835 /*---------------------------------------------------------------------
836 * scale a double precision number x in (0,1) by a power of 2 and chop it
837 *---------------------------------------------------------------------*/
838 static int icnvrt(double x, int ipwr2) {
839     return ((int)(ipwr2 * x));
840 }
841
842 /*---------------------------------------------------------------------
843 c       set ith element of sparse vector (v, iv) with
844 c       nzv nonzeros to val
845 c---------------------------------------------------------------------*/
846 static void vecset(
847     int n,
848     double v[], /* v[1:*] */
849     int iv[],   /* iv[1:*] */
850     int *nzv,
851     int i,
852     double val)
853 {
854     int k;
855     boolean set;
856
857     set = FALSE;
858     for (k = 1; k <= *nzv; k++) {
859         if (iv[k] == i) {
860             v[k] = val;
861             set  = TRUE;
862         }
863     }
864     if (set == FALSE) {
865         *nzv = *nzv + 1;
866         v[*nzv] = val;
867         iv[*nzv] = i;
868     }
869 }
```

- Source Code of FFT
- hpccfft.h

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <malloc.h>
 4 #include <math.h>
 5
 6 #ifdef LONG_IS_64BITS
 7 typedef unsigned long u64Int_t;
 8 typedef long s64Int_t;
 9 #else
10 typedef unsigned long long u64Int_t;
11 typedef long long s64Int_t;
```

```
12 #endif
13
14 typedef double fftw_real;
15
16 typedef struct {
17      fftw_real re, im;
18 } fftw_complex;
19
20 #define c_re(c)  ((c).re)
21 #define c_im(c)  ((c).im)
22
23 #define ARR2D(a,i,j,lda) a[(i)+(j)*(lda)]
24 #define ARR3D(a,i,j,k,lda1,lda2) a[(i)+(lda1)*((j)+(k)*(lda2))]
25 #define ARR4D(a,i,j,k,l,lda1,lda2,lda3) a[(i)+(lda1)*((j)+(lda2)*((k)+(lda3)*(l))))]
26 #define c_mul3v(v,v1,v2) \
27 { c_re(v) = c_re(v1)*c_re(v2) - c_im(v1)*c_im(v2);
28 c_im(v) = c_re(v1)* c_im(v2) + c_im(v1)*c_re(v2); }
29 #define c_assgn(d,s) { c_re(d)=c_re(s);c_im(d)=c_im(s); }
```

- fft_xmp.h

```
1 #pragma xmp nodes p(*)
2
3 #define N1 (10*1024)
4 #define N2 (10*1024)
5 #define N (N1*N2)
6 #define N_MAX (N1 > N2 ? N1 : N2)
7
8 #pragma xmp template t0(0:((10*1024)*(10*1024))-1)
9 #pragma xmp template t1(0:(10*1024)-1)
10 #pragma xmp template t2(0:(10*1024)-1)
11
12 #pragma xmp distribute t0(block) onto p
13 #pragma xmp distribute t1(block) onto p
14 #pragma xmp distribute t2(block) onto p
15
16 extern fftw_complex in[N], out[N], tbl_ww[N];
17 extern fftw_complex tbl_w1[N1], tbl_w2[N2], work[N_MAX];
18
19 #pragma xmp align in[i] with t0(i)
20 #pragma xmp align out[i] with t0(i)
21 #pragma xmp align tbl_ww[i] with t0(i)
```

- zfft1d-sixstep-xmp.c

```
1 #include "hpccfft.h"
2 #include "fft_xmp.h"
3
4 fftw_complex a_work[N2][N1];
5 #pragma xmp align a_work[*][i] with t1(i)
6
7 void settbl2(fftw_complex w[N1][N2])
8 {
9   int i,j;
```

```
10   double pi, px;
11 #pragma xmp align w[i][*] with t1(i)
12
13   pi=4.0*atan(1.0);
14   px=-2.0*pi/((double)(N1*N2));
15
16 #pragma xmp loop on t1(i)
17   for(i = 0; i < N1; i++) {
18     for(j = 0; j < N2; j++) {
19       c_re(w[i][j]) = cos(px*((double)(i))*((double)(j)));
20       c_im(w[i][j]) = sin(px*((double)(i))*((double)(j)));
21     }
22   }
23 }
24
25 void zfft1d0(fftw_complex a[N2][N1], fftw_complex b[N1][N2],
26              fftw_complex ww[N1][N2], fftw_complex w1[N1], fftw_complex w2[N2],
27              fftw_complex *work, int ip1[3], int ip2[3])
28 {
29   int i, j;
30   fftw_complex ztmp1,ztmp2,ztmp3;
31 #pragma xmp align a[i][*] with t2(i)
32 #pragma xmp align b[i][*] with t1(i)
33 #pragma xmp align ww[i][*] with t1(i)
34
35 #pragma xmp gmove
36   a_work[:][:] = a[:][:];
37
38 #pragma xmp loop on t1(i)
39   for(i = 0; i < N1; i++) {
40     for(j = 0; j < N2; j++) {
41       c_assgn(b[i][j],a_work[j][i]);
42     }
43   }
44
45 #pragma xmp loop on t1(i)
46   for(i = 0; i < N1; i++)
47     HPCC_fft235(b[i],work,w2,N2,ip2);
48
49 #pragma xmp loop on t1(i)
50   for(i = 0; i < N1; i++){
51     for(j = 0; j < N2; j++) {
52       c_assgn(ztmp1,b[i][j]);
53       c_assgn(ztmp2,ww[i][j]);
54       c_mul3v(ztmp3, ztmp1, ztmp2);
55       c_assgn(b[i][j],ztmp3);
56     }
57   }
58
59 #pragma xmp loop on t1(i)
60   for(i = 0; i < N1; i++) {
61     for(j = 0; j < N2; j++){
62       c_assgn(a_work[j][i],b[i][j]);
63     }
```

```
64    }
65
66  #pragma xmp gmove
67    a[:][:] = a_work[:][:];
68
69  #pragma xmp loop on t2(j)
70    for(j = 0; j < N2; j++) {
71      HPCC_fft235(a[j],work,w1,N1,ip1);
72    }
73
74  #pragma xmp gmove
75    a_work[:][:] = a[:][:];
76
77  #pragma xmp loop on t1(i)
78    for(i =0; i < N1; i++){
79      for(j=0; j < N2; j++){
80        c_assgn(b[i][j],a_work[j][i]);
81      }
82    }
83  }
84
85  int
86  zfft1d(fftw_complex a[N], fftw_complex b[N], int iopt, int n1, int n2)
87  {
88    int i;
89    double dn;
90    int ip1[3], ip2[3];
91  #pragma xmp align a[i] with t0(i)
92  #pragma xmp align b[i] with t0(i)
93
94    if (0 == iopt) {
95      HPCC_settbl(tbl_w1, n1);
96      HPCC_settbl(tbl_w2, n2);
97      settbl2((fftw_complex **)tbl_ww);
98      return 0;
99    }
100
101   HPCC_factor235( n1, ip1 );
102   HPCC_factor235( n2, ip2 );
103
104   if (1 == iopt){
105 #pragma xmp loop on t0(i)
106     for (i = 0; i < N; ++i) {
107       c_im( a[i] ) = -c_im( a[i] );
108     }
109   }
110
111   zfft1d0((fftw_complex **)a, (fftw_complex **)b, (fftw_complex **)tbl_ww,
112           (fftw_complex *)tbl_w1, (fftw_complex *)tbl_w2, (fftw_complex *)work,
113           ip1, ip2);
114
115   if (1 == iopt) {
116     dn = 1.0 / N;
117 #pragma xmp loop on t0(i)
```

```
118     for (i = 0; i < N; ++i) {
119       c_re( b[i] ) *= dn;
120       c_im( b[i] ) *= -dn;
121     }
122   }
123
124   return 0;
125 }
```

- fft_main_xmp.c

```
 1 #include "hpccfft.h"
 2 #include "fft_xmp.h"
 3
 4 fftw_complex in[N], out[N], tbl_ww[N];
 5 fftw_complex tbl_w1[N1], tbl_w2[N2], work[N_MAX];
 6
 7 double HPL_timer_cputime( void );
 8
 9 int
10 main(void)
11 {
12   int rv, n, failure = 1;
13   double Gflops = -1.0;
14   FILE *outFile;
15   int doIO = 0;
16   double maxErr, tmp1, tmp2, tmp3, t0, t1, t2, t3;
17   int i, n1, n2;
18   int rank;
19
20 #ifdef _XMP
21   rank = xmp_get_rank();
22 #else
23   rank = 0;
24 #endif
25
26   doIO = (rank == 0);
27
28   outFile = stdout;
29   srand(time(NULL));
30
31   n = N;
32   n1 = N1;
33   n2 = N2;
34
35   t0 = -HPL_timer_cputime();
36   HPCC_bcnrand( 0, in );
37   t0 += HPL_timer_cputime();
38
39   t1 = -HPL_timer_cputime();
40   zfft1d( in, out, 0, n1, n2 );
41   t1 += HPL_timer_cputime();
42
43   t2 = -HPL_timer_cputime();
```

```
44   zfft1d( in, out, -1, n1, n2 );
45   t2 += HPL_timer_cputime();
46
47   t3 = -HPL_timer_cputime();
48   zfft1d( out, in, +1, n1, n2 );
49   t3 += HPL_timer_cputime();
50
51   HPCC_bcnrand( 0, out );
52
53   maxErr = 0.0;
54 #pragma xmp loop on t0(i)
55   for (i = 0; i < N; i++) {
56     tmp1 = c_re( in[i] ) - c_re( out[i] );
57     tmp2 = c_im( in[i] ) - c_im( out[i] );
58     tmp3 = sqrt( tmp1*tmp1 + tmp2*tmp2 );
59     maxErr = maxErr >= tmp3 ? maxErr : tmp3;
60   }
61
62 #pragma xmp reduction(max:maxErr)
63
64   if (doIO) {
65     fprintf( outFile, "Vector size: %d\n", n );
66     fprintf( outFile, "Generation time: %9.3f\n", t0 );
67     fprintf( outFile, "Tuning: %9.3f\n", t1 );
68     fprintf( outFile, "Computing: %9.3f\n", t2 );
69     fprintf( outFile, "Inverse FFT: %9.3f\n", t3 );
70     fprintf( outFile, "max(|x-x0|): %9.3e\n", maxErr );
71   }
72
73   if (t2 > 0.0) Gflops = 1e-9 * (5.0 * n * log(n) / log(2.0)) / t2;
74
75   if (doIO)
76     fprintf(outFile, "Single FFT Gflop/s %.6f\n", Gflops);
77
78   return 0;
79 }
80
81 #include <sys/time.h>
82 #include <sys/resource.h>
83
84 #ifdef HPL_STDC_HEADERS
85 double HPL_timer_cputime( void )
86 #else
87 double HPL_timer_cputime()
88 #endif
89 {
90   struct rusage               ruse;
91
92   (void) getrusage( RUSAGE_SELF, &ruse );
93   return( (double)( ruse.ru_utime.tv_sec  ) +
94           ( (double)( ruse.ru_utime.tv_usec ) / 1000000.0 ) );
95 }
```

- Source Code of Linpack

```
1 #include <math.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define ZERO 0.0
6 #define ONE 1.0
7 #define EPS 1.0e-8
8
9 #define N 1024*8
10
11 double a[N][N], b[N], x[N], pvt_v[N];
12 int ipvt[N], n, rank, size;
13
14 #pragma xmp nodes p(*)
15 #pragma xmp template t(0:(1024*8)-1)
16 #pragma xmp distribute t(cyclic) onto p
17 #pragma xmp align a[*][i] with t(i)
18 #pragma xmp align b[i] with t(i)
19 #pragma xmp align x[i] with t(i)
20
21 void
22 matgen(double a[N][N], int n, double b[N] , double *norma)
23 {
24   int init, i, j;
25   double norma_temp;
26 #pragma xmp align a[*][i] with t(i)
27 #pragma xmp align b[i] with t(i)
28
29   srand48(rank);
30   *norma = 0.0;
31
32   for (j = 0; j < n; j++) {
33 #pragma xmp loop on t(i)
34     for (i = 0; i < n; i++) {
35       a[j][i] = drand48();
36       norma_temp = (a[j][i] > norma_temp) ? a[j][i] : norma_temp;
37     }
38   }
39 #pragma xmp reduction(max:norma_temp)
40
41   *norma = norma_temp;
42
43 #pragma xmp loop on t(i)
44   for (i = 0; i < n; i++) {
45     b[i] = 0.0;
46   }
47
48   for (j = 0; j < n; j++) {
49 #pragma xmp loop on t(i)
50     for (i = 0; i < n; i++) {
51       b[i] = b[i] + a[j][i];
52     }
53   }
54 }
```

```
55
56 int
57 A_idamax(int b, int n, double dx[N])
58 {
59   double dmax, g_dmax, temp;
60   int i, ix, itemp;
61 #pragma xmp align dx[i] with t(i)
62
63   if (n < 1) return -1;
64   if (n == 1) return 0;
65
66   itemp = 0;
67   dmax = 0.0;
68
69 #pragma xmp loop on t(i) reduction(lastmax:dmax/itemp/)
70   for (i = b; i < b+n; i++) {
71     temp = dx[i];
72     if(fabs(temp) >= dmax) {
73       itemp = i;
74       dmax = fabs(temp);
75     }
76   }
77
78   return itemp;
79 }
80
81 void
82 A_dscal(int b, int n, double da, double dx[N])
83 {
84   int i, m, mp1, nincx;
85 #pragma xmp align dx[i] with t(i)
86
87   if (n <= 0) return;
88
89 #pragma xmp loop on t(i)
90   for (i = b; i < b+n; i++)
91     dx[i] = da*dx[i];
92 }
93
94 void
95 A_daxpy(int b, int n, double da, double dx[N], double dy[N])
96 {
97   int i;
98 #pragma xmp align dx[i] with t(i)
99 #pragma xmp align dy[i] with t(i)
100
101   if (n <= 0) return;
102   if (da == ZERO) return;
103
104 #pragma xmp loop on t(i)
105   for (i = b; i < b+n; i++) {
106     dy[i] = dy[i] + da*dx[i];
107   }
108 }
```

```
109
110 void
111 dgefa(double a[N][N], int n, int ipvt[N])
112 {
113   double t;
114   int j, k, kp1, l, nm1, i;
115   double x_pvt;
116 #pragma xmp align a[*][i] with t(i)
117
118   nm1 = n-1;
119
120   for (k = 0; k < nm1; k++) {
121
122     kp1 = k+1;
123     l = A_idamax(k, n-k, a[k]);
124     ipvt[k] = l;
125
126 #pragma xmp task on t(l)
127     if(a[k][l] == ZERO) {
128       printf("ZERO is detected\n");
129       exit(1);
130     }
131
132 #pragma xmp gmove
133     pvt_v[k:n-1] = a[k:n-1][l];
134
135     if (l != k) {
136 #pragma xmp gmove
137       a[k:n-1][l] = a[k:n-1][k];
138 #pragma xmp gmove
139       a[k:n-1][k] = pvt_v[k:n-1];
140     }
141
142     t = -ONE/pvt_v[k];
143     A_dscal(k+1, n-(k+1), t, a[k]);
144
145     for (j = kp1; j < n; j++) {
146       t = pvt_v[j];
147       A_daxpy(k+1, n-(k+1), t, a[k], a[j]);
148     }
149   }
150
151   ipvt[n-1] = n-1;
152 }
153
154 void
155 dgesl(double a[N][N], int n, int ipvt[N], double b[N])
156 {
157   double t;
158   int k, kb, l, nm1;
159 #pragma xmp align a[*][i] with t(i)
160 #pragma xmp align b[i] with t(i)
161
162   nm1 = n-1;
```

```
163
164   for (k = 0; k < nm1; k++) {
165
166     l = ipvt[k];
167 #pragma xmp gmove
168     t = b[l];
169
170     if (l != k) {
171 #pragma xmp gmove
172       b[l] = b[k];
173 #pragma xmp gmove
174       b[k] = t;
175     }
176
177     A_daxpy(k+1, n-(k+1), t, a[k], b);
178   }
179
180   for (kb = 0; kb < n; kb++) {
181     k = n - (kb+1);
182 #pragma xmp task on t(k)
183 {
184     b[k] = b[k]/a[k][k];
185     t = -b[k];
186 }
187 #pragma xmp bcast t from t(k)
188
189     A_daxpy(0, k, t, a[k], b);
190   }
191 }
192
193 double
194 epslon(double x)
195 {
196   double a, b, c, eps;
197   a = 4.0e0/3.0e0;
198   eps = ZERO;
199
200   while (eps == ZERO) {
201     b = a - ONE;
202     c = b + b + b;
203     eps = fabs(c-ONE);
204   }
205
206   return(eps*fabs(x));
207 }
208
209 double buffer[N];
210
211 void
212 dmxpy(int n, double y[N], double *x, double m[N][N])
213 {
214   int i, j;
215   double temp;
216 #pragma xmp align m[*][i] with t(i)
```

```
217 #pragma xmp align y[i] with t(i)
218
219 #pragma xmp gmove
220   buffer[:] = x[:];
221
222 #pragma xmp loop on t(i)
223   for (i = 0; i < n; i++) {
224     temp = 0;
225     for (j = 0; j < n; j++) {
226       temp = temp + m[j][i]*buffer[j];
227     }
228     y[i] = y[i] + temp;
229   }
230 }
231
232 void
233 check_sol(double a[N][N], int n, double b[N], double x[N])
234 {
235   int i;
236   double norma, normx, residn, resid, eps, temp_b, temp_x;
237 #pragma xmp align a[*][i] with t(i)
238 #pragma xmp align b[i] with t(i)
239 #pragma xmp align x[i] with t(i)
240
241 #pragma xmp loop on t(i)
242   for (i = 0; i < n; i++) {
243     x[i] = b[i];
244   }
245
246   matgen(a, n, b, &norma);
247
248 #pragma xmp loop on t(i)
249   for (i = 0; i < n; i++) {
250     b[i] = -b[i];
251   }
252
253   dmxpy(n, b, x, a);
254
255   resid = 0.0;
256   normx = 0.0;
257
258 #pragma xmp loop on t(i) reduction(max:resid, normx)
259   for (i = 0; i < n; i++) {
260     temp_b = b[i];
261     temp_x = x[i];
262     resid = (resid > fabs(temp_b)) ? resid : fabs(temp_b);
263     normx = (normx > fabs(temp_x)) ? normx : fabs(temp_x);
264   }
265
266   eps = epslon((double)ONE);
267   residn = resid/(n*norma*normx*eps);
268
269   if(rank == 0) {
270     printf("   norm. resid      resid           machep\n");
```

```
271     printf("%8.1f       %16.8e%16.8e\n",
272             residn, resid, eps);
273   }
274 }
275
276 int
277 main(void)
278 {
279   int i,j;
280   double ops,norma, t0,t1, dn;
281
282   rank = xmp_get_rank();
283   size = xmp_get_size();
284
285   if(rank == 0)
286     printf("Linpack ...\n");
287
288   n = N;
289   dn = N;
290   ops = (2.0e0*(dn*dn*dn))/3.0 + 2.0*(dn*dn);
291
292   matgen(a, n, b, &norma);
293
294   t0 = xmp_get_second();
295   dgefa(a, n, ipvt);
296   dgesl(a, n, ipvt, b);
297   t1 = xmp_get_second();
298
299   if(rank == 0)
300     printf("time=%g, %g MFlops\n",t1-t0, ops/((t1-t0)*1.0e6));
301
302   check_sol(a, n, b, x);
303
304   if(rank == 0)
305     printf("end ...\n");
306
307   return 0;
308 }
```

- Source Code of RandomAccess

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef unsigned long long      u64Int;
5 typedef signed long long        s64Int;
6
7 #define POLY                    0x0000000000000007ULL
8 #define PERIOD                  1317624576693539401LL
9 #define NUPDATE                 (4 * XMP_TABLE_SIZE)
10
11 #define XMP_TABLE_SIZE          131072
12 #define PROCS                   2
13 #define LOCAL_SIZE              XMP_TABLE_SIZE/PROCS
14
```

```
15 u64Int Table[LOCAL_SIZE];
16
17 #pragma xmp nodes p(*)
18 #pragma xmp coarray Table[*]
19
20 u64Int
21 HPCC_starts(s64Int n)
22 {
23   int i, j;
24   u64Int m2[64];
25   u64Int temp, ran;
26
27   while(n < 0) n += PERIOD;
28   while(n > PERIOD) n -= PERIOD;
29   if(n == 0) return 0x1;
30
31   temp = 0x1;
32   for(i = 0; i < 64; i++) {
33     m2[i] = temp;
34     temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
35     temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
36   }
37
38   for(i = 62; i >= 0; i--)
39     if((n >> i) & 1) break;
40
41   ran = 0x2;
42   while(i > 0) {
43     temp = 0;
44     for(j = 0; j < 64; j++)
45       if((ran >> j) & 1) temp ^= m2[j];
46     ran = temp;
47     i -= 1;
48     if((n >> i) & 1)
49       ran = (ran << 1) ^ ((s64Int) ran < 0 ? POLY : 0);
50   }
51
52   return ran;
53 }
54
55 static void
56 RandomAccessUpdate(u64Int s)
57 {
58   u64Int i, temp;
59
60   temp = s;
61   for(i = 0; i < NUPDATE/128; i++) {
62       temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
63       Table[temp%LOCAL_SIZE][(temp%XMP_TABLE_SIZE)/LOCAL_SIZE] ^= temp;
64   }
65 #pragma xmp barrier
66 }
67
68 int
```

```
69 main(void)
70 {
71   int rank, size;
72   u64Int i, b, s;
73   double time, GUPs;
74
75   rank = xmp_get_rank();
76   size = xmp_get_size();
77   b = (u64Int)rank * LOCAL_SIZE;
78
79   for(i = 0; i < LOCAL_SIZE; i++) Table[i] = b + i;
80   s = HPCC_starts((s64Int)rank);
81
82   time = -xmp_get_second();
83   RandomAccessUpdate(s);
84   time += xmp_get_second();
85
86   GUPs = (time > 0.0 ? 1.0 / time : -1.0);
87   GUPs *= 1e-9*NUPDATE;
88
89 #pragma xmp reduction(+:GUPs)
90
91   if(rank == 0) {
92     printf("Executed on %d node(s)\n", size);
93     printf("Time used: %.6f seconds\n", time);
94     printf("%.9f Billion(10^9) updates per second [GUP/s]\n", GUPs);
95   }
96
97   return 0;
98 }
```