A Global View Programming Abstraction for Transitioning MPI Codes to PGAS Languages

Tiffany M. Mintz, Oscar Hernandez, and David E. Bernholdt

Oak Ridge National Laboratory
1 Bethel Valley Rd
Oak Ridge TN, USA
{mintztm,oscar,bernholdtde}@ornl.gov

Abstract. The multicore generation of scientific high performance computing has provided a platform for the realization of Exascale computing, and has also underscored the need for new paradigms in coding parallel applications. The current standard for writing parallel applications requires programmers to use languages designed for sequential execution. These languages have abstractions that only allow programmers to operate on the process centric local view of data. To provide suitable languages for parallel execution, many research efforts have designed languages based on the Partitioned Global Address Space (PGAS) programming model. Chapel is one of the more recent languages to be developed using this model. Chapel supports multithreaded execution with high-level abstractions for parallelism. With Chapel in mind, we have developed a set of directives that serve as intermediate expressions for transitioning scientific applications from languages designed for sequential execution to PGAS languages like Chapel that are being developed with parallelism in mind.

1 Introduction

The prevalence of multicore architectures for scientific computing has ushered in a new era in high performance computing. The multicore era has been marked by Peta-scale supercomputing machines with distributed shared memory architectures that exploit the advantages of both the data and message passing parallel paradigms[1]. The distributed shared memory architecture, known as the Non-Uniform Memory Access (NUMA) architecture [2, 3], is composed of a distributed yet globally accessible address space that allows all processors to have direct access to all memory. The address space is distributed such that each processor has a direct connection to a portion of memory, and is provided a mapping which allows direct access to memory connected to other processors. This global mapping enables a fast, direct reference of data stored in memory partitions connected to other processors (remote data), and even faster access to data in the processor's own memory partition (local data). Since the NUMA architecture is implemented on multicore devices [4–6], several processors are placed on a chip to form a single compute node with a direct connection to the

S. Poole, O. Hernandez, and P. Shamis (Eds.): OpenSHMEM 2014, LNCS 8356, pp. 120–133, 2014.

same memory partition. Each node is effectively a Symmetric Multiprocessor (SMP) with very fast, uniform access to memory from each processor.

While the multicore, NUMA architecture provides fast data movement and the potential for easy programmability, the current standard for programming scientific applications for parallel execution does not truly exploit these advantages. A sufficient programming model would need to provide mechanisms for managing data locality as well as take advantage of the global view of data provided by the architecture. Over the years, there has been much attention given to the need for programming models and languages that provide high level constructs that map well to scientific applications and provide opportunities for optimal use of the underlying architecture[7]. A programming model that has been the basis for much of the research and development of new languages is Partitioned Global Address Space (PGAS) [8–10]. There has been consistent research and development of PGAS languages from HPF in the early 90s to Chapel which debuted about a decade later with new features and functionality continually being added.

Although PGAS languages have yet to be fully adopted by the general HPC community, we are encouraged by the continued progress being made in the development of Chapel [11, 12] and the lessons learned from previous languages like HPF [13], X10[14] and ZPL[15]. So with a focus on aiding the adoption of PGAS languages by computational scientists in the HPC community, we have developed a directives based approach to expressing the global view of local data distributions and data movement in SPMD codes. This set of directives will serve as an intermediate step for incrementally transforming scientific codes from sequential, local view languages to parallel, global view languages.

The directives provide representations for high-level expressions of data distributions, parallel data movement, processor arrangements and processor groups. These assertions provide high-level constructs for describing the global nature of an application without programmers having to manage low-level details. The directives also correlate to high-level structures in Chapel, such as locales, parallel loops and domain maps so that replacing the directives with Chapel code is easy and straightforward. In addition to using the directives for describing the global state, a handle to the global domain is also created with every data distribution to allow parallel loops and interprocessor communication to be expressed from the global view using directives. For assertions of interprocessor data movement, the directives are translated to OpenSHMEM message passing operations, which provide consistent performance gains over MPI.

In this paper we continue our discussion of PGAS languages in Section 2. Section 3 gives more specific details about the directives and how they can be used to create explicit expressions of an application's global view. Section 4 provides a case study of the directives in stencil and matrix multiply codes. Section 5 concludes this paper with a summary of our approach.

2 Implementations of the PGAS Model

Parallel programming models designed for partitioned global address space (PGAS) languages UPC [16], Global Arrays (GA) [17], Co-Array Fortran (CAF) [18] Fortran 2008, and Titanium [19], target large distributed memory systems at different levels of abstraction. The PGAS languages provide a means for expressing local and global views of data and do not expect the programmer to provide all the details of data exchange, thus improving productivity. To achieve high performance, these models may be adapted to operate on a "local", or fragmented, view of data, which entails major code reorganization. These languages are good for single-sided communication of small to medium size messages since they are optimized for low message latencies. They map well to data decomposition parallel schemes. However, their adoption has been limited as they have limited support for hybrid programming models and incremental parallelism.

The DARPA-funded "HPCS" programming languages Fortress [20], Chapel [21], and X10 [22] were designed to support highly productive programming for ultra scale HPC systems and merge the concepts of global views of data, tasks and locality. They provide a wealth of new ideas related to correctness, locality, efficiency and productivity. These languages offer different levels of expressivity and abstraction, giving them distinct flavors from the application developer's perspective. Yet they have much in common, including the assumption of a hierarchically parallel architecture and a global address space. They allow users to control the placement of work and data (tasks and data distributions), exploit ideas from object-oriented programming, and provide efficient synchronization via transactions. These new languages imply a high learning curve for the user and may not be intuitive enough for widespread adoption. Many proposed features have yet to be tested in real petascale-level applications. Nevertheless, much can be learned from these efforts and in the longer term, one or more of them may be adopted.

3 Enabling a Global Perspective

In order to help programmers transition their applications from source code written with a sequential language and parallelism enabled through the use of MPI, we have developed a set of directives that are representative of some of the principal concepts that are expressible using PGAS languages. A few of the key features of most PGAS languages is the expression of data from a global view, expressing data distribution patterns, and processor affinity for data movement. To achieve our goal of providing high-level programming abstractions that map to PGAS language constructs, we developed directives that provide high-level descriptions of data distributions, parallel computation and interprocessor data movement, as well as high level expressions for arranging and grouping processors. With these directives, the programmer is able to identify the regions of their application that map well to the high level constructs provided by PGAS languages, and incrementally transition their source code to these languages.

The remainder of this section, gives a description of each directive and how they may be used in scientific applications.

3.1 Data Distributions

One of the major differences between PGAS languages and sequential languages is the view of data. Sequential languages provide abstractions for a processor centric, local view of data; while PGAS languages primarily provide abstractions for a global view of data. Since the current standard for programming parallel applications uses sequential languages coupled with message passing library calls, programmers currently have to write their programs so that their data sets are pre-distributed, and all subsequent computation and communication is expressed relative to the distributed local data. In this programming model, there is no native abstraction for expressing the global view and using this expression to program applications.

To enable the expression of a global view of data in scientific applications, we use the `data_map` directive to describe the distribution of data across processors. This directive allows the definition of `BLOCK`, `CYCLIC` and `BLOCK_CYCLIC` distributions, and provides a handle to a global domain that can be used to access data and perform specific operations from the global view. The clauses associated with `data_map` are `local_data`, `global_domain`, `distribution` and `expand`. The notation used to describe local and global data is:

buf<*size1,...,sizeN*>*:[low1_idx..high1_idx,...,lowN_idx..highN_idx]*,

where *buf* is the name of a pointer or array, *N* is the number of dimensions in *buf*, *size1,...,sizeN* is a list of the size of allocated memory for each dimension (optional for `global_domain` clause), and *low1_idx..high1_idx,...,lowN_idx..highN_idx* is a list of the range of indices for each dimension. The `distribution` clause allows the programmer to specify a distribution that corresponds to each dimension, or specify the distributions in list form to explicitly indicate the distribution of each dimension (i.e. `distribution(NONE, BLOCK)` would indicate that the first dimension is not distributed and the second dimension has a BLOCK distribution). The `expand` clause is most applicable for data mappings that require a "scratch" space in the local data region. This region can be used to store regularly accessed data that is resident on another processor but should be mirrored on the current processor. Halo regions or ghost cells are common implementations of a scratch space in a data region. Section 4 provides examples of how `data_map` is used to describe distributions that are frequently used in scientific applications.

3.2 Processor Groups and Arrangements

Features such as processor groups and common processor arrangements provide a basis for which more complex expressions can be built. Our `group` and `arrange_processors` directives establish this foundation using succinct expressions that describe how the processors are assembled and ordered.

To create a processor group, a programmer would provide a group name and use triplet notation to indicate which processors would be in the group. A simple example of how the group directive could be used is provided in Fig. 1. The group that is created serves as a unique identifier for subsequent directive assertions. Once the group is created it can be mapped to an arrangement, data distribution, and some expression of computation or interprocessor data movement. If a group is not created or instanced by a directive, we assume the group to be all processors.

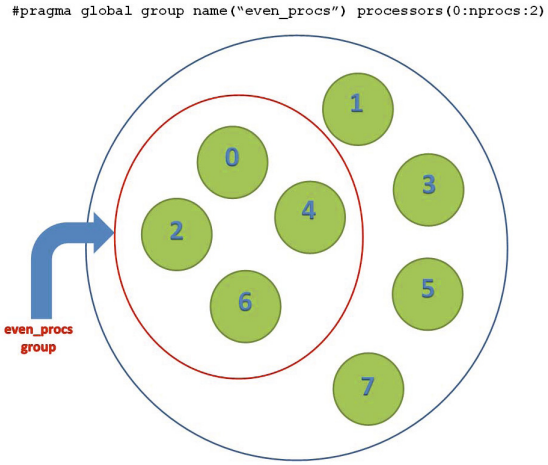


Fig. 1. Example using group directive to create the group 'even_proc'

Arranging processors has an equally simple expression which uses a combination of keywords and intuitively named clauses to describe the processors' disposition. This directive supports several arrangements, including `Master-Worker`, `Grid`, `Tree`, `Ring`, and `List`. Each of these arrangements have their own set of associated clauses for creating the corresponding formations. Table 1 gives a list of the arrangements and their clauses.

Each arrangement also has a unique set of relationships that can be assigned based on the values passed to the directive. See Table 1. These relationships help to describe data movement more concisely and with terminology that is familiar to the user. Using this directive also relieves the programmer from having to compute and manage, in some form, the process ids that correspond to these relationships.

3.3 Data Movement

Once the programmer has defined how the processors will be grouped and how they will be arranged within the group, expressing data movement using the

Table 1. Arrangements, associated clauses and the defined relationships within each formation

Arrangement	Clauses	Relationships in Arrangement
GRID <1D|2D|3D>	size_x_axis size_y_axis size_z_axis	North Neighbor South Neighbor East Neighbor West Neighbor
MASTER-WORKER	master_processor*	Master Worker Master_Worker
TREE	root* order*	Parent Sibling Children Root Leaves Depth (tree and positional)
RING	direction*	Previous Next
LIST	direction*	Previous Next Head Tail

*indicates optional clauses that have some default behavior defined when not in use

global domain is very straightforward. Configuring the data mapping relative to the processor arrangement enables a simplified expression of communication and computation from the global view. To express data movement across processors, programmers need only assert the `update` directive. The clauses associated with this directive are `update_domain`, `update_mirror`, and `on`. The `on` clause is used to specify the destination of the update.

When specifying a destination, the programmer can leverage the relationships among processors in the arrangement declared for the group. For example, if the processors are arranged in a list configuration, the programmer can simply indicate `HEAD`, `TAIL`, `NEXT`, or `PREVIOUS` as the destination for the update. To further support the PGAS programming model, this directive is translated to an appropriate communication pattern using the OpenSHMEM message passing library.

As previously stated, the expression of computation is also simplified when using the global view of data. Defining a global domain allows the expression of a parallel loop in the form of a `forall` directive. This directive has the clauses `index_var`, `domain` and `expression`. The `domain` clause is used to define the iteration space of the parallel forall loop. A user can specify the global domain created by the data distribution or express the domain as a range of indices. If an explicit range is specified, each processor's iteration space is determined by evenly distributing the indices in the range according to the number of the processors in the group. The `index_var` clause accepts a list if variables used to iterate over the domain. If the global domain handle is specified in the `domain` clause, then the variables' position in the list corresponds to that dimension in the global domain. If index ranges are specified in the `domain` clause, then the variables' position in `index_var` corresponds to the range in the same position of the `domain` clause.

The computation in the `expression` clause is then concurrently executed on each processor in the group.

4 Preliminary Experiments

As an initial step to demonstrating the simplicity of programming with our global view directives to incrementally transition applications to a PGAS programming paradigm, we have chosen two algorithms that are commonly implemented in scientific applications. The first algorithm is Jacobi's iterative method for solving a system of linear equations, and the second algorithm is a dense matrix-matrix multiplication. We compare C+MPI versions of these algorithms to selective portions of the algorithms programmed using the directives. These experiments were executed on a Cray XK7 system with 83 compute nodes. Each node has a 16-core AMD Opteron 6274 processor running at 2.2 GHz with 32 gigabytes of DDR3 memory, and Cray's high performance Gemini network.

4.1 2D Jacobi Iterative Solver

The 2D Jacobi iterative solver implements a common data distribution pattern where the problem space is mapped onto a two dimensional grid and partitioned across processors typically in block fashion. Points in the grid are updated iteratively, but an update may require data in neighboring cells that reside in a partition stored on another processor. This requires frequent remote data accesses to processors that "own" the neighboring data. So parallel implementations not only represent the program space as a 2D grid, but the processor formation is also conceptualized as a 2D grid. While this is a very common distribution and data access pattern, there are no native programming abstractions in sequential languages that embody the concept. PGAS languages like Chapel have native representations of this data distribution, but because of the global view of data, have no need for explicit point-to-point communication when accessing remote data.

Using our directives, we were able to explicitly express this distribution of data while preserving the global view in the program for very simple, unobtrusive assertions of communication. First we show, in Fig. 2, the difference in how the 2D Grid arrangement is constructed in the two versions of the algorithm. As you can see, we have been able to greatly reduce lines of code and programming effort for constructing a 2D Grid. Next, Fig. 3 shows how the `data_map` directive was used to express the block data distribution and create a handle to the global domain. In this assertion, we specify `mat` as the local partition of data with `plines` rows and `pcols` columns. The "scratch" space which will be used to store remote data is defined as an expansion of one row and one column (in every direction) of the local partition `mat`, and this data is to be mirrored in the distribution. The global domain is defined as a 2D matrix with `plines*proc_y` rows and `pcols*proc_x` columns, and is accessible through the handle `Global_Matrix`. `Global_Matrix` is then used to assert an update of

Directive Assertion of 2D Grid Processor Arrangement

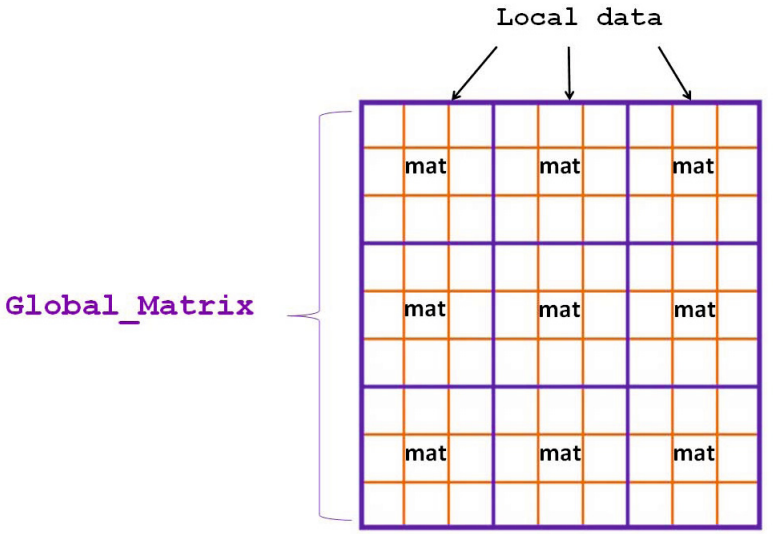
```
#pragma global arrange_processors arrangement(GRID:2D) \
     size_x_axis(proc_x)  size_y_axis(proc_y)
```

C code to mimic 2D Processor Grid

```
x = myid % proc_x;
y = myid / proc_x;

if( y == 0 )
  north = BORDER;
else
  north = x + (y - 1) * proc_x;
if( y == proc_y - 1 )
  south = BORDER;
else
  south = x + (y + 1) * proc_x;
if( x == 0 )
  west = BORDER;
else
  west = myid - 1;
if( x == proc_x - 1 )
  east = BORDER;
else
  east = myid + 1;
```

Fig. 2. Comparison of C and directive versions of 2D Grid setup




```
#pragma global data_map local_data(mat<XPLINES,XPCOLS>:[1..plines, 1..pcols]) \
  global_domain(Global_Matrix:[1..(plines)*proc_y, 1..(pcols)*proc_x]) \
  distribution(BLOCK) expand(mat:[1,1]:MIRRORED)
```

Fig. 3. Depiction of the block distribution created with the directives in the Jacobi algorithm

Directive Assertion for Updating Neighbor Data

```
#pragma global update update_mirror(Global_Matrix) on(ALL_NEIGHBORS)
```

C+MPI Code to Exchange Data Across Neighboring Processors

```
if( west != BORDER ){
  for( i = 0; i < lines; i++ )
    *(awest + i) = MAT(i, 1);
  MPI_Send(awest,lines,MPI_DOUBLE,west,MSGTYPE,MPI_COMM_WORLD);
}

if( east != BORDER ){
  for( i = 0; i < lines; i++ )
    *(aeast + i) = MAT(i, cols - 2);
  MPI_Send(aeast,lines,MPI_DOUBLE,east,MSGTYPE,MPI_COMM_WORLD);
}

if( north != BORDER )
  MPI_Send(mat+cols,cols,MPI_DOUBLE,north,MSGTYPE,MPI_COMM_WORLD);

if( south != BORDER )
  MPI_Send(mat+(lines-2)*cols,cols,MPI_DOUBLE,south,MSGTYPE,MPI_COMM_WORLD);


if( west != BORDER ){
  MPI_Recv(awest,lines,MPI_DOUBLE,west,MSGTYPE,MPI_COMM_WORLD);
    for( i = 0; i < lines; i++ )
      MAT(i, 0) = *(awest + i);
}

if( east != BORDER ){
  MPI_Recv(aeast,lines,MPI_DOUBLE,east,MSGTYPE,MPI_COMM_WORLD);
      for( i = 0; i < lines; i++ )
        MAT(i, cols - 1) = *(aeast + i);
}

if( north != BORDER )
  MPI_Recv(mat,cols,MPI_DOUBLE,north,MSGTYPE,MPI_COMM_WORLD);

if( south != BORDER )
  MPI_Recv(mat+(lines-1)*cols,cols,MPI_DOUBLE,south,MSGTYPE,MPI_COMM_WORLD);
```

Fig. 4. Comparison of MPI code and directive assertion for communicating with neighboring processors

the remote data using the keyword `ALL_NEIGHBORS` to indicate the destination. Figure 4 shows a comparison of the C+MPI and directive version of this communication. As in Fig. 2 we are again able to greatly reduce the lines of code and programming effort for expressing this communication.

As for performance, the overhead for creating and managing a 2D grid and the additional data structures needed for the neighbor communication in the C+MPI code is approximately 3.7x greater than the overhead to create and manage the 2D grid and block distribution with the directives. This significant difference in overhead performance is primarily due to the need for additional data structures to send and receive non-contiguous data in the matrix columns when using MPI point-to-point communication. Because OpenSHMEM provides strided message passing operations for point-to-point communication, the directive translation does not require additional structures for transferring data between east and west neighbors. So, the overhead performance cost for constructing the topology and distribution for a Jacobi algorithm using these global view directives is $O(1)$ since the number of computations and memory accesses needed to assign neighbors and compute and maintain global offsets and indices in the underlying translation of the directives is constant on each process even as the number of processes increase.

Moreover, translating the communication between neighboring processes using OpenSHMEM put operations provided additional performance improvements

over the original C+MPI code which implements MPI_Send and MPI_Recv operations. The OpenSHMEM translation of the directives provided a 2x average speedup over the MPI implementation of the neighbor communication. The performance results for the overhead and communication are plotted in Fig. 5 and Fig. 6, respectively.

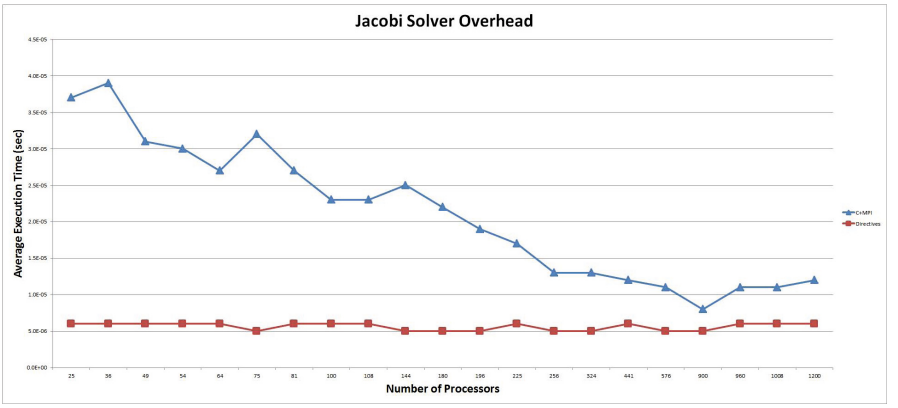


Fig. 5. Graph of overhead in C+MPI and directive version of the Jacobi solver

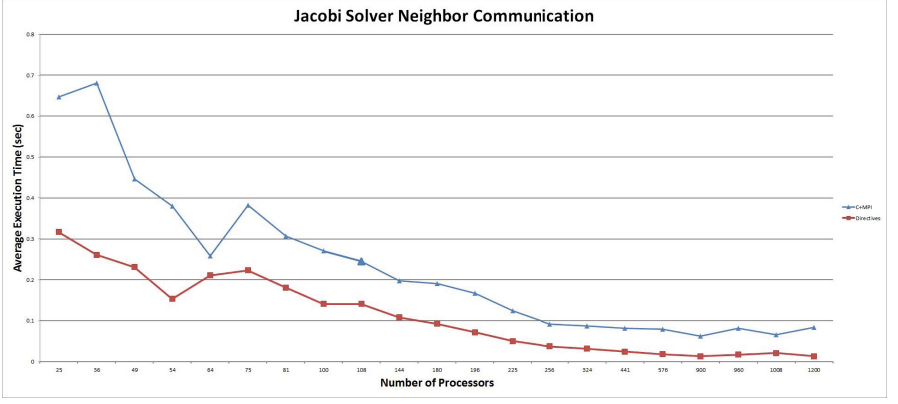


Fig. 6. Graph of neighbor communication time in C+MPI and directive version translated to OpenSHMEM of the Jacobi solver

4.2 Matrix-Matrix Multiply

The matrix-matrix multiply algorithm has a similar data distribution as the Jacobi algorithm. This distribution is also a block distribution, but only the first dimension of the 2D space is distributed. The processor arrangement for this algorithm is a `MASTER-WORKER` formation where the master also shares the computational workload. Because of its computational characteristics, we were able

Global View Matrix-Matrix Multiply with Directives

```
#pragma global arrange_processors arrangement(MASTER_WORKER, MASTER_SHARES_WORK)

#pragma global data_map local_data(a<nrows,NCA>:[0..nrows-1,0..NCA-1])  \
     global_domain(global_a:[0..NRA-1,0..NCA-1]) distribution(BLOCK, NONE)

#pragma global data_map local_data(c<nrows, NCB>:[0..nrows-1,0..NCB-1])  \
     global_domain(global_c:[0..NRA-1,0..NCB-1]) distribution(BLOCK, NONE)

#pragma global  forall  index_var(i, j)  domain(global_a)  \
     expression(global_a[i][j] = i+j)

#pragma global forall  index_var(i, j)  domain(global_c)  \
     expression(global_c[i][j] = 0.0)

for(k=0; k<NCB; k++)
  #pragma global forall index_var( i, j) domain( global_a)  \
       expression(global_c[i][k] = global_c[i][k] + global_a[i][j] * b[j][k])

#pragma global update update_domain(global_c) on(MASTER)
```

C + MPI Matrix-Matrix Multiply

```
if (taskid == MASTER) {
   for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
         a[i][j]= i+j;

   offset = (extra > 0) ? averow+1 : averow;
   for (dest=1; dest<numworkers; dest++) {
      rows = (dest <= extra) ? averow+1 : averow;
      MPI_Send(&offset, 1, MPI_INT, dest, FROM_MASTER, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, dest, FROM_MASTER, MPI_COMM_WORLD);
      MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD);
      offset = offset + rows;
   }
   rows = (extra > 0) ? averow+1 : averow;

   for (k=0; k<NCB; k++)
      for (i=0; i<rows; i++) {
         c[i][k] = 0.0;
         for (j=0; j<NCA; j++)
            c[i][k] = c[i][k] + a[i][j] * b[j][k];
      }
   if(extra > 0) {
     list_offsets[0] = 0;
     row_cnts[0] = (averow+1)*NCB;
     for (dest=1; dest<numworkers; dest++) {
        rows = (dest <= extra) ? averow+1 : averow;
        row_cnts[dest] = rows*NCB;
        list_offsets[dest] = list_offsets[dest-1]+row_cnts[dest-1];
     }
     MPI_Gatherv(MPI_IN_PLACE,0,MPI_DATATYPE_NULL,&c,row_cnts,list_offsets,MPI_DOUBLE,MASTER,MPI_COMM_WORLD);
   }else
     MPI_Gather(MPI_IN_PLACE,0,MPI_DATATYPE_NULL,&c,rows*NCB,MPI_DOUBLE,MASTER,MPI_COMM_WORLD);
}

if (taskid > MASTER) {
   MPI_Recv(&offset, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD, &status);
   MPI_Recv(&rows, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD, &status);
   MPI_Recv(&a, rows*NCA, MPI_DOUBLE, FROM_MASTER, mtype, MPI_COMM_WORLD, &status);

   for (k=0; k<NCB; k++)
      for (i=0; i<rows; i++) {
         c[i][k] = 0.0;
         for (j=0; j<NCA; j++)
            c[i][k] = c[i][k] + a[i][j] * b[j][k];
      }
   if(extra > 0)
     MPI_Gatherv(&c,rows*NCB,MPI_DOUBLE,&c,row_cnts,list_offsets,MPI_DOUBLE,MASTER,MPI_COMM_WORLD);
   else
     MPI_Gather(&c,rows*NCB,MPI_DOUBLE,&c,rows*NCB,MPI_DOUBLE,MASTER,MPI_COMM_WORLD);
}
```

Fig. 7. C+MPI and global view directives versions of Matrix-Matrix Multiplication algorithm

to almost completely program this algorithm using our global view directives. Figure 7 shows a comparison of the C+MPI version and the version using our global view directives. The most obvious difference is the substantial reduction in the lines of code. Another significant difference is the use of the data's global view to express loop computation. We were able to execute each loop using the `forall` directive.

Since in the C+MPI code the master processor is responsible for computing then distributing initial data, there is a considerable difference in the overhead.

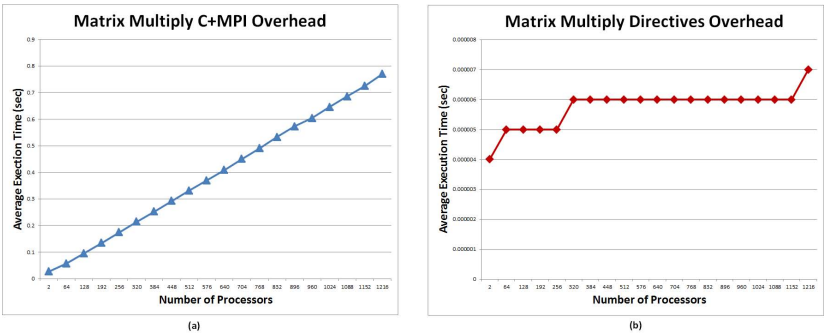


Fig. 8. Graph of matrix-matrix multiply overhead for (a)C+MPI version and (b)Directive version

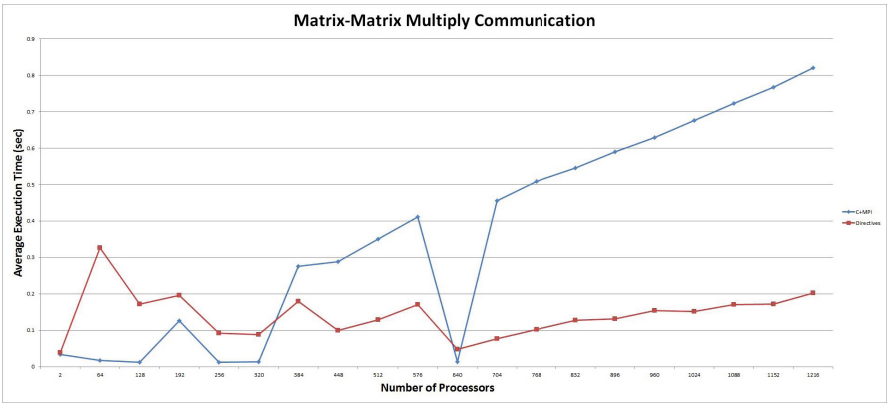


Fig. 9. Graph of average execution time for communication needed to transfer the matrix-matrix multiply solution to the master process

The directive code uses the `forall` directive to initialize data which does not require any message passing communication. Figure 8 provides the graphs plotting the overhead.

As for the communication needed to update the matrix-matrix multiply solution on the master processor, the update directive is translated to OpenSHMEM put operations on all the processors except the master with a barrier synchronization. Even with a notoriously costly collective synchronization, the OpenSHMEM translation of the directive provides a 2.5x average speedup over MPI. Figure 9 shows the average execution time for the matrix-matrix multiply communication.

5 Conclusion

Supercomputing architectures are steadily pushing the performance envelope in order to reach the next levels of computing capabilities. While our computer

and computational scientists have been able to steadily evolve their applications to run on these advanced architectures, more and more effort is being spent transforming source code. This is a definite signal to the HPC community for a new programming paradigm that provides high- level abstractions for parallel programming and enables good performance. We believe the PGAS model has the potential to be or greatly influence a new paradigm. PGAS languages like Chapel are continually making progress toward providing a rich set of features for parallel programming and good run-time performance. We believe incrementally transitioning scientific applications to PGAS languages will facilitate their adoption. Our global view directives are a fitting approach to this incremental transition. By providing directive assertions for data distributions, processor groups and arrangements, and global data movement, we enable global expressions that are analogous to the expressions found in Chapel codes in applications with an otherwise local, processor centric view of data.

Acknowledgment. This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy, including the use of resources of the Oak Ridge Leadership Computing Facility. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. This manuscript has been authored by a contractor of the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

References

1. Top 500 supercomputers (2013), `http://www.top500.org/`
2. Bolosky, W., Fitzgerald, R., Scott, M.: Simple but effective techniques for numa memory management. In: Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, SOSP 1989, pp. 19–31. ACM, New York (1989)
3. Black, D., Gupta, A., Weber, W.D.: Competitive management of distributed shared memory. In: COMPCON Spring 1989. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers, pp. 184–190 (1989)
4. Blagodurov, S., Zhuravlev, S., Fedorova, A., Kamali, A.: A case for numa-aware contention management on multicore systems. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, pp. 557–558. ACM, New York (2010)
5. Rabenseifner, R., Hager, G., Jost, G.: Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 427–436 (2009)
6. Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., Chapman, B.: High performance computing using mpi and openmp on multi-core parallel systems. Parallel Comput. 37(9), 562–575 (2011)
7. Kasim, H., March, V., Zhang, R., See, S.: Survey on parallel programming model. In: Cao, J., Li, M., Wu, M.-Y., Chen, J. (eds.) NPC 2008. LNCS, vol. 5245, pp. 266–275. Springer, Heidelberg (2008)

8. Yelick, K., Bonachea, D., Chen, W.Y., Colella, P., Datta, K., Duell, J., Graham, S.L., Hargrove, P., Hilfinger, P., Husbands, P., Iancu, C., Kamil, A., Nishtala, R., Su, J., Welcome, M., Wen, T.: Productivity and performance using partitioned global address space languages. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, PASCO 2007, pp. 24–32. ACM, New York (2007)
9. Bonachea, D., Hargrove, P., Welcome, M., Yelick, K.: Porting gasnet to portals: Partitioned global address space (pgas) language support for the cray xt. Cray User Group, CUG 2009 (2009)
10. Barrett, R.F., Alam, S.R., d Almeida, V.F., Bernholdt, D.E., Elwasif, W.R., Kuehn, J.A., Poole, S.W., Shet, A.G.: Exploring hpcs languages in scientific computing. Journal of Physics: Conference Series 125(1), 012034 (2008)
11. Dun, N., Taura, K.: An empirical performance study of chapel programming language. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp. 497–506. IEEE Computer Society, Los Alamitos (2012)
12. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21(3), 291–312 (2007)
13. Kennedy, K., Koelbel, C., Zima, H.: The rise and fall of high performance fortran: an historical object lesson. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III, pp. 7–1–7–22. ACM, New York (2007)
14. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 519–538. ACM, New York (2005)
15. Chamberlain, B.L., Choi, S.E., Deitz, S.J., Snyder, L.: The high-level parallel language zpl improves productivity and performance. In: Proceedings of the First Workshop on Productivity and Performance in High-End Computing (PPHEC 2004), pp. 66–75. Citeseer (2004)
16. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Technical report, Center for Computing Sciences (May 1999)
17. Nieplocha, J., Krishnan, M., Tipparaju, V., Palmer, B.: Global Arrays User Manual
18. Numrich, R.W., Reid, J.K.: Co-Array Fortran for parallel programming. ACM Fortran Forum 17(2), 1–31 (1998)
19. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high performance Java dialect. Concurrency: Practice and Experience 10, 825–836 (1998)
20. Allen, E., Chase, D., Luchangco, V., Maessen, J.W., Ryu, S., Steele Jr., G., Tobin-Hochstadt, S.: The Fortress language specification, version 0.785 (2005)
21. Cray Inc.: Chapel specification 0.4 (2005), `http://chapel.cs.washington.edu/specification.pdf`
22. Charles, P., Donawa, C., Ebicioğlu, K., Grothoff, C., Kielstra, A., Saraswat, V., Sarkar, V., Praun, C.V.: X10: An object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, ACM SIGPLAN, pp. 519–538 (2005)