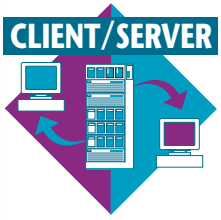
Data Parallel Programming with HPF on the RS/6000 SP



By David Klepacki and Xianneng Shen

This article is the third in a series about parallel programming models on the RS/6000 SP. The focus is on the features of the data parallel model, which is most often employed using the High Performance Fortran (HPF) Specification.

The first article[1] in this series briefly introduced the IBM RS/6000 SP™ architecture and its parallel programming models. Previous articles[2,3] discussed the message-passing model and its standard interface (MPI). This article describes an alternative to message passing: the data parallel model. The industry-standard specifications for this type of model are known as High Performance Fortran (HPF) and the more recent Data Parallel C (DPC). This article discusses the HPF implementation of the data parallel model.

The data parallel model is not new. It was predominantly used in the late 1980s during the era of parallel computers with the single instruction-multiple data (SIMD) architecture. SIMD characterizes how computations were carried out on this type of machine. In essence, each parallel processor would execute the same instruction simultaneously, but on different data. For example, a large two-dimensional matrix could be broken down into a "checkerboard" of smaller submatrices. Different processors could then operate upon these submatrices in parallel. This data parallel model could easily and naturally accommodate many engineering and scientific applications.

The scientific/engineering community quickly realized the need for a standard language/interface specification, especially for Fortran. The result was the HPF 1.0 specification, which appeared in 1992.[4] During that same period, rapid changes occurred in technology, which resulted in the demise of the SIMD architected computers in favor of more cost-effective and general-purpose computers with multiple instructions-multiple data (MIMD) architecture, such as the IBM RS/6000 SP.

MIMD enables multiple processors to operate on different instructions and different data at the same time. However, as described in the first article of this series,[1] the programming model is independent of the underlying hardware architecture.

[1] Klepacki, David and Shen, Xianneng. "Parallel Programming Models on the IBM RS/6000 SP." *AIXpert* magazine, September 1997.
[2] Shen, Xianneng; Ho, Eddie; and Hammill, Mike. "Message Passing Interface for RS/6000 SP." *AIXpert* magazine, March 1997.
[3] Shen, Xianneng and Klepacki, David. "Message Passing on the RS/6000 SP." *AIXpert* magazine, December 1997.
[4] High Performance Fortran Forum, "High Performance Fortran Language Specification, Version 1.0", *Technical Report CRPC-TR92225*, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992.

The data parallel model with HPF is as applicable on today's MIMD machines as it was on the older SIMD machines. In fact, three HPF compilers are available today for use on the IBM RS/6000 SP: xlHPF (IBM), pgHPF (Portland Group Inc.), and xHPF (Applied Parallel Research Inc.). The HPF Specification is now in its second revision (HPF 2.0, 1996).

High Performance Fortran

HPF is a set of compiler directives that facilitate data parallel programming. Although an in-depth discussion of HPF is beyond the scope of this article, its fundamental components consist of the following:

- Data partitioning directives, such as `TEMPLATE`, `ALIGN`, `PROCESSOR`, and `DISTRIBUTE`
- Data parallel execution features, such as the `FORALL` statement
- Extended intrinsic functions; for example, `number_of_processors()`
- Extrinsic procedures; Sequence and Storage Association

HPF allows the programmer a high-level expression of parallelism. In the message-passing model, the array elements of distributed data derive new offsets relative to the local name spaces of each process. HPF is based upon a global name space, where the variables and arrays are manipulated from a global perspective (as in a serial application).

HPF programmers do not have to deal with the lower level nuances, such as buffering and synchronization, as in the message-passing programming model. HPF programmers need only be concerned with the distribution of data among the processors. The HPF environment performs the necessary low-level operations in order for this to occur. Therefore, the programmer can concentrate on the physical/numerical solution aspect of the application at hand.

On the RS/6000 SP, the HPF environment, not the user, transparently manages the physical message passing between the processors. Therefore, message passing can be viewed as the "assembly language" of parallel programming. The data parallel model provides a high-level language interface to parallel programming. However, this typically does not come without a cost. The ease-of-use associated with HPF usually results in sacrificing some degree of performance, not unlike the classical difference between high-level language programming and assembly language programming.

Fundamentals of HPF

Data parallel programming generally consists of three basic steps:

1. Create logical templates for the data and decide how the data should be partitioned. This should exploit the natural parallelism inherent in the application, not the computer.

2. Define virtual topologies for the processors (for example, a two-dimensional mesh or a three-dimensional cube). Typically, these topologies mimic the physical geometries and exploit the natural symmetries of the application itself.

3. Map the templates onto the virtual processor topologies. The HPF environment will then provide the necessary translation of operations onto the physical processors during compilation and runtime.

Let us consider how these steps are implemented in HPF using a simple example of rank sorting a vector of numbers. In Fortran 77, the serial program to perform this function is shown in Figure 1.

```
subroutine permute (key, rank, temp, n)
integer n, key(n), rank(n), temp(n)
do i = 1, n
    temp(i) = key(rank(i))
enddo
do i = 1, n
    key(i) = temp(i)
enddo
```

Figure 1. Rank sorting of a vector of numbers

Three vectors of length n are passed into this subroutine. The key vector is reordered according to the values in the rank vector. Because of the indirect addressing involved, it is necessary to introduce a temporary vector to hold the values of the keys as they are being reordered. This keeps key values that have not yet been reordered from being overwritten. Lastly, the key vector is updated from the temporary vector.

Parallelization with HPF

We want to parallelize this subroutine using HPF. The first step is to identify the logical structures to be partitioned, then determine exactly how they should be partitioned. In this example, all structures are in the same form: a vector of length n. So, our logical structure of partitioning can be represented as t(n). HPF has three methods of partitioning data:

- BLOCK
- CYCLIC
- BLOCK-CYCLIC

The BLOCK scheme partitions a vector into contiguous sub-blocks, each of length n/p, where p represents the number of processors available for this computation. The CYCLIC scheme assigns each consecutive element of the vector to a different processor, "round-robin" style (for example, as in dealing cards). The BLOCK-CYCLIC scheme is a combination of the previous two schemes. It assigns sub-blocks that are much smaller than the n/p size in the BLOCK case to processors round-robin style. We chose the BLOCK scheme for our example.

The second step is to define a virtual processor topology. This step is optional on the RS/6000 SP, and the default topology is a set of processors that are completely connected. However, for illustration purposes, we chose a linear array of processors p to mimic the vector nature of our application.

Lastly, we map our variables onto our virtual processor grid using the distribute directive. The final parallel subroutine is shown in Figure 2.

The template directive defines the fundamental logical structure of a partitioned

```
 subroutine permute (key, rank, temp, n)
 integer n, key(n), rank(n), temp(n)
!hpf$ template t(n)
!hpf$ align with t :: key, rank, temp
!hpf$ processors p(number_of_processors())
!hpf$ distribute t(BLOCK) onto p
 do i = 1, n
    temp(i) = key(rank(i))
 enddo
 do i = 1, n
    key(i) = temp(i)
 enddo
```

Figure 2. Final parallel program

object. Since it does not allocate any storage, there is no need to declare its type in the program. The align directive is optional here, but we use it to illustrate good programming practice. Typically, the arrays and vectors of more complex programs will require alignment due to variations in their declared sizes.

The processor topology is defined with the processor directive. In this example, it uses the HPF Extended Intrinsic function, number_of_processors(), which returns the number of physical processors employed at runtime. Using this function, the programmer does not need to hard-code the program for any specific number of processors. Rather, this program will function on any number of processors without the need to recompile each time a different number of them are used.

This application has been parallelized with the addition of four comment-based directives; therefore, this code still runs unchanged on a stand-alone workstation. It is simple to change the distribution of the data among the processors and/or the processor topology. For example, cyclic data distribution among the processors requires only that the BLOCK parameter be changed to CYCLIC. As another example, introducing a three-dimensional grid of 27 processors can be obtained with the specification of q(3,3,3).

More than one processor topology can be defined concurrently. More complex applications may require one topology during one part of the computation and another topology during another part of

the computation. The reader should compare and contrast these abilities with the message-passing model. In general, these data parallel constructs are not as easily handled with the message-passing model.

FORALL Construct

The `FORALL` statement is another powerful feature of HPF. It provides a convenient syntax for simultaneous assignments to array elements. Figure 3 shows some examples.

Note that the parallelism is implied. The arrays can be physically partitioned via the `distribute` directive. All of the necessary interprocessor communication is transparently performed by the HDF environment.

Summary

Data parallel programming is a viable and powerful model when used appropriately. In particular, it should be used whenever regular structures of data are manipulated in a static and predetermined way. Otherwise, load-balancing issues arise, which would be better handled by one of the other two programming models.

To really appreciate the power of HPF, you should attempt to parallelize the simple rank-sort programming example in this article using the message-passing model (for example, `MPI`). This example will be reviewed again in the next article of this series, which will discuss the Virtual Shared Memory model.

Acknowledgments

The authors would like to thank the following people for invaluable discussions on HPF: Christopher Kerr (IBM Corporation), John Levesque (Applied Parallel Research, Inc.), and Doug Miles (Portland Group, Inc.).


```
! A = Transpose of B
FORALL (i=1:m, j=1:n) A(i,j) = B(j,i)

! Assignment to diagonal elements
FORALL (i=1:n) A(i,i) = B(i)

! A of the second assignment uses values
! of the first assignment
FORALL (i=2:n-1, j=2:n-1)
  A(i,j) = A(i,j-1) + A(i-1,j)
  B(i,j) = A(i,j)
END FORALL
```

Figure 3. FORALL statement examples

***Xianneng Shen**, IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. E-mail: xshen@us.ibm.com. Dr. Shen is a programming consultant in the RS/6000 Executive Briefing Center. He has a BS and an MS in Electrical Engineering from the University of Electronic Science and Technology of China, an MS in Computer Engineering from Syracuse University, and a PhD in Electrical Engineering from Syracuse University.*

***David Klepacki**, IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. Dr. Klepacki has been working in IBM's POWERparallel Systems Group since its beginning in 1991 as a computational physicist and scientific applications specialist with emphasis on performance benchmarking. Today, in addition to his technical endeavors, he also manages the parallel software tools segment for the technical marketing branch of the RS/6000 Division. Dr. Klepacki's current interests include performance programming, scalable parallel algorithms, scalable I/O, and portable high-performance computing tools. He holds a PhD in Theoretical Nuclear Physics from Purdue University as well as an MS in Electrical Engineering from Syracuse University.*