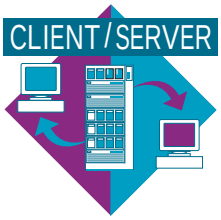
Shared Memory Programming on the RS/6000 SP



By David Klepacki and Xianneng Shen

This fourth article in a series about parallel programming on the RS/6000 SP focuses on the shared memory programming model using PAMS. The first article introduced the basic architecture of the RS/6000 SP and the three fundamental parallel programming models.[1] The second article focused on the message-passing model using MPI/LAPI,[2] and the third article focused on the data-parallel model using HPF.[3]

Shared memory programming is now possible on the RS/6000 SP™ with software called Parallel Application Management System (PAMS) from Myrias, Inc. PAMS enables programmers to develop load-balanced parallel applications from serial applications without data distribution, task management, or modification of source code.

The shared memory programming model can be subdivided into two categories: virtual shared object (VSO) and virtual shared memory (VSM). In the virtual shared object model, multiple processes globally share data objects. Regions of memory are reserved for shared access and, by necessity, involve the usual locking mechanisms to coordinate access among the processes.

This type of programming model has many implementations: Linda and its Tuplespace (Scientific Computing Associates), Treadmarks (Rice University), Global Arrays (Pacific-Northwest National Lab), and ABC++ shared regions (IBM Corporation), to name a few. To make it run in parallel, this model typically requires the addition of new function calls, data constructs, and in some cases, language extensions to an existing serial application.

The virtual shared memory model, on the other hand, assumes that the entire address space is shared. The way it shares memory pages is logically similar to that of the virtual memory manager (VMM) of the AIX® operating system.

When a page fault occurs in AIX, the memory address currently being referenced is not valid or available in main memory. It must be fetched from a paging file, usually located on a hard disk. Similarly, in a parallel computer using VSM, a page fault occurs when a referenced memory address is not valid or available in main memory. However, in this case, it might not be valid or available on the processing node, so it must be fetched from another processing node where it is available. (A processing node refers to one or more processors that share a single address space.)

With VSM, each processing node is a paging device for the parallel application and its associated virtual address space. Since the VSM model is a natural extension of the operating system's VMM, it does not

[1] Klepacki, David and Shen, Xianneng. "Parallel Programming Models on the IBM RS/6000 SP." *AIXpert*, September 1997.
[2] Shen, Xianneng and Klepacki, David. "Message-Passing on the RS/6000 SP." *AIXpert*, December 1997.
[3] Klepacki, David and Shen, Xianneng. "Data Parallel Programming with HPF on the RS/6000 SP." *AIXpert*, March 1998.

require additional constructs in an application. Often, serial applications can be made parallel on the RS/6000 SP without modifications to the source code, apart from comment-based directives to the VSM environment.

PAMS

PAMS is the only commercially supported VSM system in the marketplace today. This software implements a virtual shared memory environment for the RS/6000 SP. Using PAMS, the complex, distributed memory architecture of the RS/6000 SP looks to the programmer like one very large shared-memory multiprocessor system.

PAMS will run on any combination of RS/6000 SP node types—thin, wide, or high. Because of its design, PAMS can automatically balance the load of the application among the different performance capabilities of the nodes. Pages of memory naturally migrate to the processors that need them. Each processor in a parallel job can work at its own pace, making it unnecessary for programmers to perform any explicit data distribution or task management. In fact, the only programmer requirement is to identify the parallel opportunity in the application and to add the appropriate comment-based directives.

Memory Models

The three basic ways to share memory among processing nodes having different physical address spaces include the following:

- Uncached Store Coherent Memory (USC): Migrates a single memory region among the processors as needed
- Cached Store Coherent Memory (CSC): Replicates the memory regions as necessary, but with only one writable region
- Loop Coherent Memory (LC): Replicates the memory regions as necessary with all such regions being simultaneously writable

The USC model is simple and does not suffer any consistency problems that are associated with replicated data. However, this model does not always perform well. Its major drawback is the potential for "jitter." Jitter occurs when multiple processing nodes reference the same memory region so that this region ping-pongs among them. This happens because the USC memory model contains no other copies of the memory region. It is necessary to use semaphores or other locking mechanisms to control this condition.

> ***Using PAMS, the complex, distributed memory architecture of the RS/6000 SP looks like one very large shared-memory multiprocessor system.***

The CSC model reduces the jitter problem by having multiple read-only copies of a memory region available to all other processing nodes. Unlike the USC model where a memory region needs to be migrated for both read and write requests, the CSC model only needs to migrate memory regions for write requests.

Memory replication introduces another problem: the potential for data inconsistency. For example, updates to shared memory regions may not become visible to some copies of the memory regions in the other processing nodes when they are needed. If this occurs, the application may end up using "stale" data in the processing nodes that have the replicated memory regions.

Another form of this problem can occur, known as a *race condition*, resulting in data inconsistency among the nodes. This happens when a memory region is updated before its previous update is replicated to the same memory locations in the other processing nodes. Synchronization and memory-locking techniques must be used to circumvent these types of problems.

The USC and CSC models both suffer from *false sharing*. This occurs when multiple processing nodes concurrently request to update memory locations that do not overlap, but reside on the same memory

page. The actual memory locations being modified are different, but since they are part of the same fundamental shared memory unit (page), they are treated as if they were the same memory locations being modified; hence, the term "false" sharing. This creates a performance penalty in these models when logically there should be none. False sharing incurs unnecessary jitter in these models.

The LC model can avoid some of these problems, because different processing nodes can concurrently write to every copy of a memory region. All updated memory regions are then merged in hierarchical fashion into a single, valid memory region when needed. Memory pages do not migrate in this model; therefore, no jitter or false sharing occurs, only replication and merging. Data inconsistency and race conditions can still occur in the LC model, but only through programmer error, such as declaring a loop to be parallel when it is not.

Lastly, the LC model does allow multiple processing nodes to update the same memory locations concurrently. This is valid only when the user identifies these memory locations as being part of a global reduction operation (see Dependency Violations below).

PAMS supports all three types of memory models but prefers the LC model, which in most cases, is sufficient for many applications. In addition, the memory models can coexist in the same application. Some applications may require fine tuning of the memory management by incorporating either the USC or CSC model, but this requires additional support functions and often results in the need to modify source code. On the other hand, the LC model is implemented entirely with comment-based directives.

PAMS Development

PAMS is a complete environment that consists of precompilers, runtime libraries, and an extensive set of GUI-based tools (such as a debugger and various monitors) that support the major programming languages: C, C++, and Fortran. It is important to note that PAMS uses IBM AIX native compilers, since this retains the highest optimization of code on the RS/6000 processors. The programming interface is quite small, but deceptively powerful (see Figure 1).

Basic Directives

Although PAMS can create parallel programs, it is necessary to identify parallel capability in the source code. The LC model has two forms: blocks of code that can be performed in parallel and loops with independent iterations. Figure 2 shows an example of two parallel blocks of code using Fortran 77 syntax. Each block can be executed on a different processing node.

Blocks of code that can be executed independently are identified to PAMS with the `cpams parbegin` directive at the beginning of the blocks and the `cpams endpar` directive at the end of the blocks. Within this series of blocks, the `cpams parallel` directive identifies each independent block.

Basic Directives	/* pams pardo */	cpams pardo
	/* pams parbegin */	cpams parbegin
	/* pams parallel */	cpams parallel
	/* pams endpar */	cpams endpar
Extended Features	/* pams mergeby ... */	cpams mergeby ...
	/* pams prefetch ... */	cpams prefetch ...
Advanced Features	Store coherent memory support	
	Local system calls	
	GUI debug, monitoring, performance toolset	

Figure 1. PAMS directives and features

```
cpams parbegin
c The following block of code is executed on one processing node
cpams parallel
       statement or subroutine 1
       statement or subroutine 2
       ...
c The following block of code is executed on another processing node
cpams parallel
       statement or subroutine 100
       statement or subroutine 200
       ...
cpams endpar
```

Figure 2. Two parallel blocks of code using Fortran 77 syntax

Then, each block of code can execute on a different processing node (see Figure 2).

Figure 3 shows an example of parallel loops using Fortran 77 syntax. The loops contain independent iterations that can be executed on different processing nodes. It also supports nested parallel loops.

A loop with independent iterations can be executed in parallel by identifying it with the `cpams pardo` directive. The PAMS environment uses a sophisticated algorithm to distribute the iterations among the processing nodes in a load-balanced fashion, especially important when employing processing nodes of varying power. Nested loops can be parallelized as shown in Figure 3. In addition, I/O statements (read/write) can be included in parallel loops (as well as parallel blocks) to achieve a parallel I/O capability.

Extended Features

The extended features of PAMS consist of the `mergeby` and `prefetch` directives (see Figure 1). The `prefetch` directive is a fine-tuning optimization that tells the PAMS environment to begin acquiring the necessary memory locations prior to their actual reference. The `mergeby` directive defines the values for storage locations that are concurrently updated by different processing nodes. This is commonly known as a *global reduction operation*.

Dependency Violations

Three types of data dependency can inhibit a loop from being executed in parallel. The first is a *reduction violation*, which appears in the following form:

```
do i = 1, n
  sum = sum + ...
enddo
```

This global sum reduction operation example is typical of iterative solution methods. This type of loop cannot be parallelized (yet) since the iterations depend on each other. The `sum` memory location is repeatedly overwritten, based upon its previous value.

The USC and CSC memory models require a semaphore lock for updating `sum` with partial sums from each processor. However, the `mergeby` directive of the LC

```
cpams pardo
       do x = 1, n
          statement or subroutine 1
          statement or subroutine 2
          ...
cpams pardo
          do y = 1, m
             statement or subroutine 100
             statement or subroutine 200
             ...
           enddo
       enddo
```

Figure 3. Parallel loops using Fortran 77 syntax

memory model makes global reduction possible without any serialization. This directive can declare `sum` to be a merge variable, so that its memory location can be independently updated with different partial sums on each replicated memory copy of the processing nodes. At the end of the loop, the replicated memory regions are merged into a single valid image; that is, all of the partial sums are added during the merging process to produce the final result in the `sum` location. Any binary commutable operation including `sum`, `maximum`, and `minimum` can be used to achieve this.

The second type of data dependency called a *store-store* violation appears as follows:

```
do i = 1, n
  ...
  x = f(i) ...
enddo
y = x + ...
```

The scalar variable `x` is not a temporary loop variable, but one that is used after the last loop iteration. This loop must be modified in order to be parallelized because the memory location `x` must be updated in sequence to retain its last value. Declaring such a loop to be parallel is a common oversight, so PAMS can perform dynamic dependency analysis (for Fortran) on all loops to detect this situation at runtime. If it is detected PAMS logs an error, but the program will continue to execute the loop serially rather than abort or produce an incorrect result.

The last dependency violation type is the well-known *store-fetch* violation:

```
do i = 1, n
  f(i) = f(i - 1) + ...
enddo
```

This type of loop cannot be parallelized since each iteration is dependent upon its previous iteration. Unfortunately, the only choice is to find an alternative algorithm that does not employ store-fetch violations in the loops.

PAMS Example

Let us consider the simple example taken from a previous article in this series[4] concerning rank sorting a vector of numbers. Figure 4 shows the PAMS program in Fortran 77 that performs this function.

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

Figure 4. Sorting a vector of numbers

The addition of two comment-based directives parallelizes this routine; therefore, this code still runs unchanged on a stand-alone workstation. Compare this to the High Performance Fortran (HPF) implementation[5] and notice that data distribution is not necessary when using virtual shared memory. The data movement among the processors is performed transparently during runtime by the PAMS environment. *Note:* If you have not implemented this routine using message-passing, now is a good time to do so in order to understand some differences among the three programming models.

Summary

Virtual shared memory provides parallel programmers with an environment that is conceptually similar to traditional uniprocessor development—it has a single, logical address space. Virtual shared memory can overcome the physical limitations associated with physically shared memory machines. Most physically shared memory machines cannot sustain more than 32 processors.

Virtual shared memory is intended for systems having many processors. PAMS, which is relatively new on the RS/6000 SP,

[4] Klepacki, David and Shen, Xianneng. "Data Parallel Programming with HPF on the RS/6000 SP." *AIXpert*, March 1998.
[5] Ibid.

has already demonstrated scalability up to 64 nodes. PAMS enables many applications that run on traditional symmetric multiprocessing (SMP) systems to be easily migrated to the RS/6000 SP. This exciting technology has great promise to simplify the programming process for machines that are ever increasing in size and complexity.

Other software vendors, such as Applied Parallel Research, Inc. (APR) are building tools to further enhance the usability of PAMS. APR now produces a tool that analyzes FORTRAN code for parallel opportunities and automatically inserts the appropriate PAMS directives into the source code.

Acknowledgements

The authors would like to thank Wayne Karpoff and Brian Lake of Myrias, Inc. for the many lucid discussions about virtual shared memory and PAMS, and for pioneering the next generation of parallel software development.

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.

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.