IBM's Magazine for AIX Developers December 1997

AIXpert

FOCUS ON PARALLEL PROGRAMMING

XL Fortran

INSIDE THIS ISSUE

- X11Release 6.1
- XL Fortran Compiler
- JavaBeans
- AIX READY

AIXpert BACK TO CONTENTS

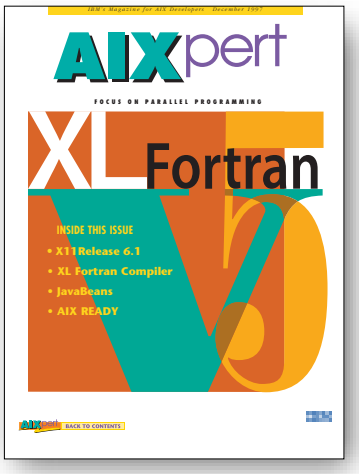
IBM

AIXpert

DIGITAL VERSION

IBM

TABLE OF CONTENTS



Click here to view a full-size version of this issue's cover.

Click here to go to a complete version of this issue for printing.

DECEMBER
1997

AIXpert BACK TO MAIN MENU

For Now, For Always… Fortran

COMMENTARY

With all the recent hullabaloo in the press (including *AIXpert*) about the significance of the new Java programming language, it's easy to lose sight of the contributions of the more established languages—both in sustaining existing applications and in exploiting new features in today's ever-more-powerful computers. In this issue we take advantage of the recent release of version 5 of IBM's XL Fortran compiler to remind you (and ourselves) that the Fortran language is a powerful programming tool. Read about the features of the newest version of this compiler, how to use it in a parallel processing environment, and how to program multithreaded Fortran applications in the two Fortran-related articles that we have in this issue. As a complement to Fortran in a parallel processing environment, be sure to read the second article in our series about parallel programming models on the RS/6000 SP™ system.

Of course, we can't wean ourselves entirely off Java™, so we've included an article that compares shell programming to Java Beans and introduces IBM's BeanExtender technology (available for evaluation download). In addition, the newly released AIX® 4.3 also provides some subject matter for this issue. Learn about the latest features in AIXwindows® 4.3 which incorporates X Window System® X11R6.1 in our article from the AIX developers responsible for the AIXwindows portion of AIX 4.3. And, to help boost the commercial success of your AIX-based software, be sure to take advantage of the new AIX READY™ compatibility mark described in the related article.

Capping off this issue we have a discussion of the Serial Storage Architecture used in attaching mass storage subsystems (and others) to the RS/6000™, as well as our AIX Questions feature from the technical support experts of the IBM Solution Developer Program.

George Noren

***George Noren**, IBM Corporation, Internal Zip 1034, 11400 Burnet Road, Austin, TX 78758. Internet: geo@austin.ibm.com. Since joining IBM in September 1979, Mr. Noren has written hardware and software manuals, including AIX and RS/6000 manuals, and was a member of the InfoExplorer™ design team. He has worked as a system administrator for several AIX systems and is a Certified AIX System Administrator. He is currently Editor in Chief of the World Wide Web site for IBM's Solution Developer Program (www.developer.ibm.com) in addition to his work with* AIXpert *Magazine. Mr. Noren studied engineering at Illinois Institute of Technology, holds a BA in English from the University of Minnesota and an MBA from St. Edwards University in Austin.*

George Noren

Get on the Mark with AIX READY

By Barbara Formichelli

AIX READY™ is a new feature available to RS/6000™ Partners in Development that provides an AIX READY compatibility mark to qualifying software developers for use in marketing their products. Displaying this mark will help them signify to customers that their products comply with the criteria required to be compatible with AIX, IBM's premier UNIX® operating system.

Binary compatibility has become an increasingly important issue for software developers, especially when considering the variety of hardware platforms, associated operating systems, and multiple releases requiring support. IBM is demonstrating its commitment to binary compatibility, and is providing support to help you meet the challenges of today's diverse and competitive business environment.

For this reason, we are pleased to introduce the AIX READY package, which is an advanced entitlement for IBM Solution Develop Program RS/6000 Partners in Development. This offering, which includes a compatibility mark, enables you to identify your products as being binary compatible with AIX. This mark will be available to you upon completion of a fast and simple certification process.

How can AIX READY make your software more valuable? By helping to broaden the appeal of products developed for AIX. It can also help reduce your time and effort by allowing you to use any AIX system as a reference platform for building and maintaining applications.

Program Benefits

Once you complete the certification process, you will be able to take advantage of a wide range of attractive benefits. It all begins with access to the AIX compatibility mark, which you may use on product packaging, in printed and electronic marketing materials, and in advertisements. An online design guide accompanies the mark and provides examples illustrating its use.

AIX READY also offers other benefits to its members:

- Technical support: Available through the IBM Solution Developer Technical Support Center if you experience any incompatibilities in an AIX environment
- "Test-drive" facilities: Allow you to test your products on, or port them to, a variety of hardware platforms at the IBM Solution Partnership Center in San Mateo, California
- Access to developer programs: Includes AIX-related marketing and/or technical assistance offered by original equipment manufacturers (OEMs) participating in the AIX Multiple Vendor Program

Further additions to this offering are planned for the coming year.

Barbara Formichelli

AIX READY and AIX MVP: A Winning Combination

The ease of self-certification for the AIX READY compatibility mark is made possible due to the stringent requirements of the AIX Multiple Vendor Program (MVP). The goal of MVP is to ensure conformance to a common Application Binary Interface (ABI) and Application Programming Interface (API) across a range of systems that use the AIX operating system. Under this program, all systems that run AIX and use the AIX trademark must pass a set of certification tests that demonstrate this compliance. AIX MVP is currently sponsored by IBM, Bull®, Hitachi/Hitachi Data Systems®, Motorola™, and Thomson/CETIA™.

What's Being Said About AIX READY

"Since the release of AIX Version 4, IBM has consistently demonstrated its commitment to binary compatibility. The success of the AIX Multiple Vendor Program has driven this message even further by helping to provide AIX binary compatibility not only across releases, but across a range of platforms from participating system vendors.

"The AIX READY mark is intended to signify to the marketplace that products labeled AIX READY will run wherever AIX runs. The ability to deliver one AIX across a variety of hardware platforms is the key to the AIX READY mark."

—Mike Borman, general manager, RS/6000 Division, IBM Corporation

---

"We look forward to the opportunities and benefits that the AIX READY compatibility mark will offer to the industry. In an effort to encourage widespread AIX product usage and the resulting benefits, we feel that this mark will definitely represent the value of binary compatibility which truly benefits all participants and should be universally recognized."

—Toshiakira Ikeda, general manager, Strategic Business Development Division, Information Systems Group, Hitachi Ltd.

"Bull today offers a complete line of AIX servers from small departmental to large enterprise clusters and is a charter participant in the AIX Multiple Vendor Program. AIX READY now enables ISVs, with no additional development effort, to offer their applications on all AIX platforms including Bull's Escala and Estrella products."

—Armand Malka, UNIX servers vice president, Bull Worldwide Information Systems

---

"Moving applications between platforms and into different environments can be very challenging. As a leading OEM supplier of open architecture PowerPC-based single-board computers and systems products, CETIA (a subsidiary of Thomson-CSF) relies on the AIX Multiple Vendor Program and the AIX READY mark to ensure that our customers' applications will run reliably in high-performance, complex embedded systems.

"These VME (Versa Module Europe)-based systems are being deployed in laboratory and harsh field environments such as airborne and shipboard defense applications where dependable software is very critical. AIX is a proven operating system that enjoys popularity worldwide. Coupled with the outstanding performance of the PowerPC microprocessor, AIX is unbeatable and that is why Thomson-CSF has selected this IBM offering as its standard UNIX solution."

—Bob Huettner, general manager, CETIA, Inc.

"Motorola Computer Group (MCG) provides AIX to technical OEM customers. With AIX, MCG provides unparalleled scalability from single-board computers to servers to complete fault-tolerant systems. Only AIX scales across this broad spectrum of hardware offerings. MCG's customers can run one operating system on our broad spectrum of hardware without changing versions of AIX or third-party software, which provides uncompromising scalability.

"AIX READY means that software running on IBM RS/6000 systems will run on MCG's single-board computers, servers, and fault-tolerant systems without any changes. The market for software companies grows to include other AIX systems without any additional effort."

—William Donlan, vice president and director, Technical Systems and Telecommunications Marketing, Motorola Computer Group

How to Enroll

If you are not yet a member of the RS/6000 Partners in Development, begin by accessing our Web site at *www.developer.ibm.com* and follow the instructions for registering online. Check the RS/6000 specialty option on the registration form and complete the non-confidential Business Development Plan. As soon as your application is approved, you'll have access to the many RS/6000 Partners in Development benefits available and can apply for advanced entitlements like AIX READY.

Once you are an RS/6000 member, you can access your RS/6000 Partners in Development Notebook at *www.developer.ibm. com/pid_rs6000*, select Marketing, then select AIX READY: How to get access. Here you will find the AIX READY Compatibility Checklist and License Agreement to have your product approved for AIX READY. The checklist certifies your compliance with compatibility guidelines for specific versions/releases of your software. The license agreement states that in addition to meeting this criteria, your software performs all functions specified in your product documentation and is running successfully in a customer environment on AIX or has been fully tested in an AIX environment to ensure its compatibility with AIX.

You must certify that your software meets the following requirements:

- It must only use interfaces (APIs, commands, and data structures) that are published in IBM documentation on AIX.
- It does not use any AIX features that are defined by IBM documentation on AIX as non-portable.
- It does not require any specific devices, device drivers, or other specific hardware.
- It uses shared libraries; therefore, it does not bind static.
- It is compiled in common mode; therefore, it is not for one specific processor mode.

If the software does not meet the designated criteria, you will be required to perform testing on representative platforms and submit a statement indicating the steps that have been taken to ensure compatibility. Assistance is available to guide you through this process.

Once your application has been approved, you will be able to use the AIX READY compatibility mark. The mark is available for downloading in a variety of formats—PDF, EPS, TIF, GIF, and others. Design guidelines and examples showing how to use the mark are also available.

For More Information

We encourage you to register now! Here's how you can learn more about these offerings:

- For information on AIX READY, visit the RS/6000 Partners in Development Notebook at *www.developer.ibm. com/pid_rs6000* and look under "Marketing."
- For information on the AIX Multiple Vendor Program, visit the AIX MVP home page at *www.developer.ibm. com/mvp*.
- Send E-mail to *ibmsdp@us.ibm.com*.
- Call 770-835-9902 worldwide or 800-627-8363 in the U.S. or Canada and ask for the RS/6000 Partners in Development Representative.

Barbara Formichelli, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Ms. Formichelli is responsible for management and administration of marketing and technical support activities associated with the AIX Multiple Vendor Program and AIX READY. She has a BS in English from State University of New York at New Paltz.*

Serial Storage Architecture

By Dave E. Hall

This article describes Serial Storage Architecture (SSA), a definition and general specification of a high-performance serial link for the attachment of input/output devices. An overview of the architecture is followed by a description of the IBM SSA products available for the RS/6000™ under AIX®, and some hints and tips on configuring SSA products with AIX.

Serial Storage Architecture (SSA) defines a high-performance serial link for the attachment of input/output devices. It has been optimized for storage applications such as hard disk drives, host adapter cards, and array controllers.

SSA has many advantages over existing parallel interfaces such as the Small Computer Systems Interface (SCSI-2). It uses compact cables and connectors, and it has better performance, connectivity, and reliability. SSA is an open standard ANSI® architecture. IBM first announced and started shipping products based on SSA for use with the RS/6000 in August 1995. It has subsequently announced support for SSA products on IBM Personal Computers and other manufacturers' systems, such as Hewlett-Packard® and Sun®.

By November 1997, IBM stated that it had shipped over two petabytes of SSA storage since its introduction, and that it was currently working on a future implementation of SSA-based products that would be twice the speed of existing ones. SSA products form a key part of the recently announced Seascape Architecture.

SSA Overview

The heart of any IBM SSA-based product is the SSA chip—the Serial Interface Chip (SIC). The SIC, implemented in IBM CMOS4-LP technology, provides two 20 MB/sec serial ports. A single SIC can be used to implement a dual-port SSA disk drive, host adapter, or bridge controller. For the products described in this article, each disk drive has one SIC and each adapter card has two.

The SSA design allows three basic configuration possibilities: string, loop, or switch. The most common one implemented in products is loop, because it has both higher bandwidth and reliability. Figures 1 and 2 show typical SCSI and SSA subsystems. Figure 3 shows a comparison of SSA and SCSI.

RS/6000 SSA Products Overview

Several products provide low-cost, high-performance disk storage for the RS/6000.

7131/7133 SSA Disk Subsystems

The 7131 is the entry-level SSA subsystem for the RS/6000 and RS/6000 SP™. It is sold as standard with a pair of high-performance Ultrastar disk drives; each pair consists of disks of 2.2 GB, 4.5 GB, or 9.1 GB capacity. Three additional hot swappable slots can contain a mix of any of these capacity drives for a maximum subsystem configuration of 45.5 GB. For storage requirements greater than this, multiple 7131s can be connected together.

Dave E. Hall

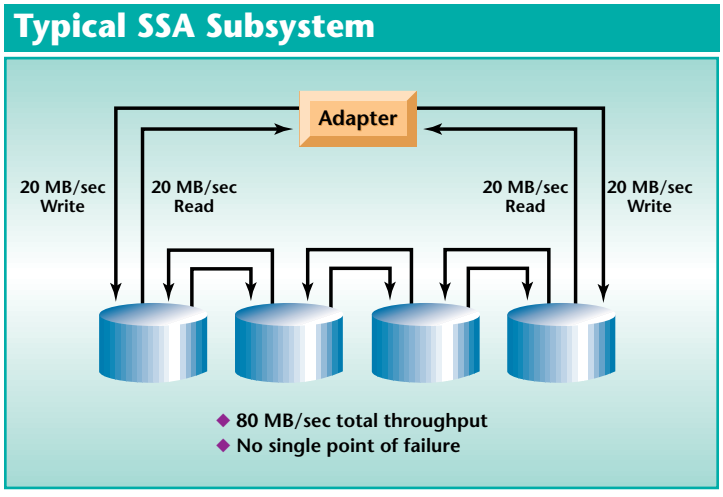


Figure 1. Typical SSA subsystem

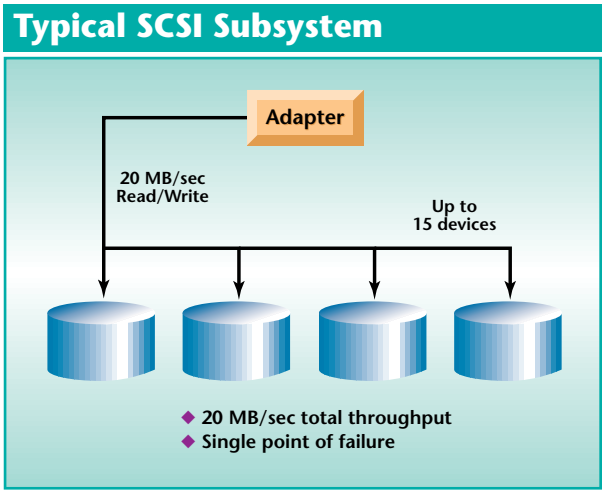


Figure 2. Typical SCSI subsystem

The 7133 is the highest performing disk subsystem for the RS/6000 and the RS/6000 SP. It is available as a rack mount or a desk-side tower unit. Its configuration is similar to the 7131, with Ultrastar drives that can be intermixed for flexibility in building storage environments from gigabytes to terabytes. Its serial loop architecture provides redundant paths to all disks. The 7133 supports mission-critical applications via High Availability Cluster Multiprocessing (HACMP).

Two adapters attach the 7131 and 7133 to multiple RS/6000s. The 4-Port Adapter connects up to two hosts while the Enhanced 4-Port Adapter can attach up to eight hosts. This allows multiple hosts to access the same data with high performance and availability features of SSA. Since SSA devices are configured in loops, they do not require bus arbitration. This enables multiple concurrent operations to occur in separate sections of the loop, resulting in higher overall throughput.

Two adapters provide RAID-5 capability for the SSA serial interface. The 4-Port RAID adapter supports Micro Channel® systems while the PCI SSA 4-Port RAID adapter supports selected PCI systems. Both adapters are ideal for video applications, data servers, and mission-critical storage needs.

IBM 7190 SCSI Host to SSA Loop Attachment

As the number of storage devices increases on a SCSI bus, performance decreases dramatically. The 7190 overcomes these SCSI limitations by allowing users to efficiently attach high-performance 7131 and 7133 SSA disk storage subsystems to systems that do not provide native SSA attachment capabilities. Up to 48 SSA disks can be

SSA	SCSI
4 signal wires—6 total	31 signal wires—68 total
Up to 80 MB/sec system throughput	Up to 20 MB/sec system throughput
No discrete terminators	Discrete terminators
127 devices per loop or string	15 devices per string
Hot plugging (simple, designed in)	Hot plugging (possible but difficult)
Alternate paths avoid a single point of failure	SCSI bus is a single point of failure

Figure 3. Comparison of SSA and SCSI

connected to each SCSI adapter. Because each 7190 and its complement of SSA disks looks to the system like a single SCSI device, the level of arbitration is dramatically reduced and typically leads to significant increases in performance over the traditional SCSI attachment method. The 7190 is supported on IBM and selected Sun and Hewlett-Packard computers.

The 7190 uses a Differential Fast/Wide SCSI-2 bus adapter to connect to the system. Because it relies on the host's standard SCSI driver and hardware for communications, there is no need for modification to system hardware, software, or applications.

All attached storage subsystems can be monitored online for the following:

- Display vital product data
- Show SSA loop topology
- Show SSA-to-SCSI ID/LUN mapping
- Set or change modes for drive identification and diagnostics
- Download microcode to the 7190 and SSA disk drives as required
- Provide detailed system-wide error log
- Alert system to redundant power/cooling errors, loop topology changes, or drive errors

SSA and AIX Devices

SSA subsystems and adapters introduce some new terms for AIX. Figure 4 shows the result of an AIX `lsdev` command issued against some SSA devices.

Note from the example in Figure 4 that pdisk0 is not necessarily associated with hdisk0; it is, in fact, associated with hdisk1. System administrators must determine the pdisk-to-hdisk mapping in order to perform certain functions, such as assigning disks to volume groups, cabling for performance, changing disk placement for performance, handling disk failures, and so on. Fortunately, with AIX 4.1.5 this

Command	State	Device
ssar	Defined	SSA Adapter Router
ssa0	Available 00-05	SSA Adapter
ssa1	Available 00-06	SSA Enhanced Adapter
ssa2	Available 00-07	SSA RAID Adapter
pdisk0	Available 00-05-P	1 GB SSA C Physical Disk Drive
pdisk1	Available 00-05-P	2 GB SSA C Physical Disk Drive
pdisk2	Available 00-06-P	2 GB SSA C Physical Disk Drive
pdisk3	Available 00-06-P	4 GB SSA C Physical Disk Drive
hdisk1	Available 00-05-L	SSA Logical Disk Drive
hdisk2	Available 00-05-L	SSA Logical Disk Drive
hdisk3	Available 00-06-L	SSA Logical Disk Drive
hdisk4	Available 00-06-L	SSA Logical Disk Drive

ssar—SSA router, which should always be in the defined state
pdisk—physical disk
hdisk—hard disk; one pdisk per hdisk for non-RAID, multiple pdisks per hdisk for RAID

Devices are configured in "walk the loop" order.
- Adapters from lowest numbered slot
- Pdisk in serial number order for each adapter
- Hdisks following the pdisks

Figure 4. Result of lsdev command on various SSA devices

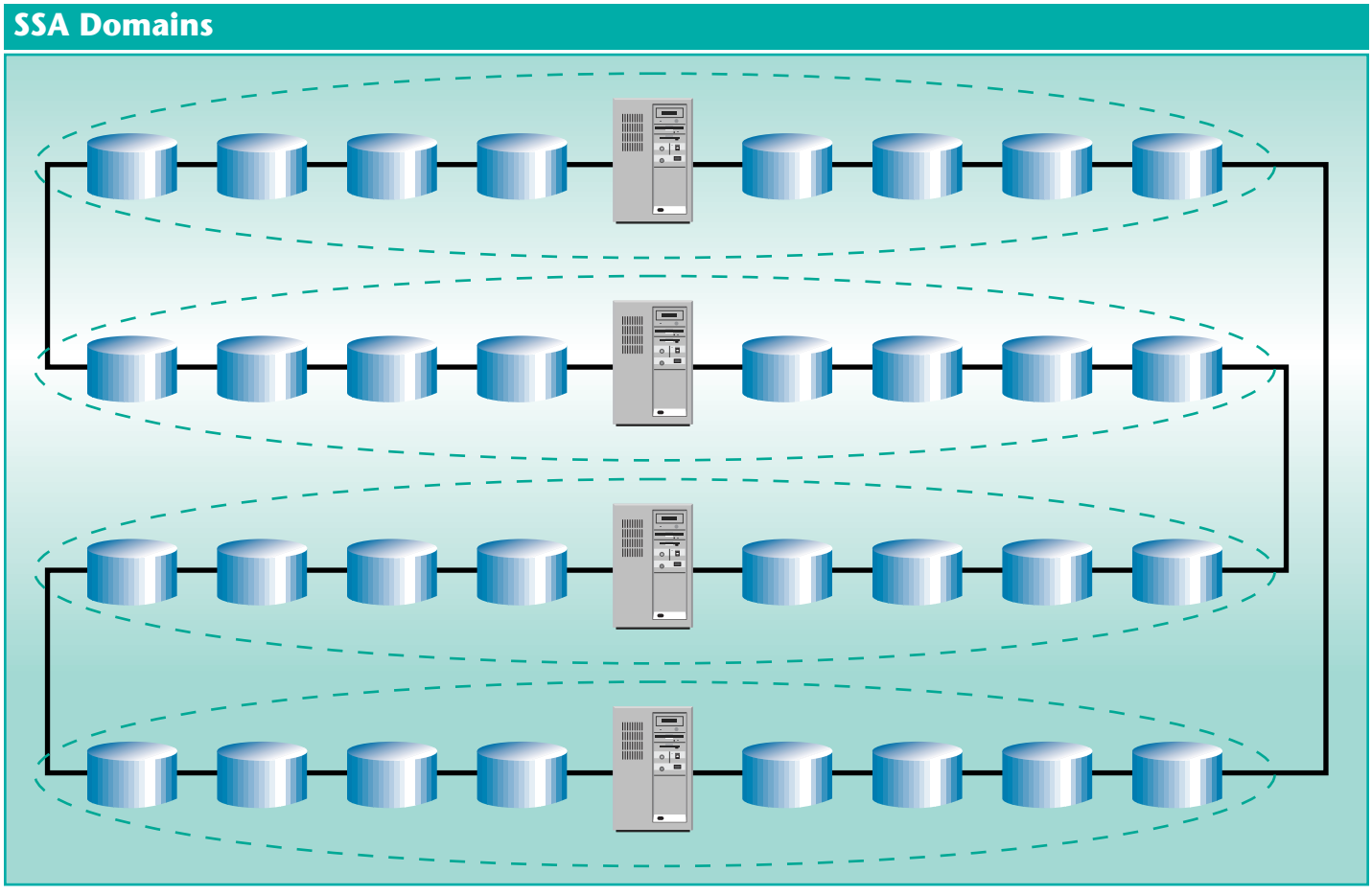


Figure 5. SSA domains

can be done via the System Management Interface Tool (SMIT).

The numbers associated with the adapters and disks are determined in the following way. When the system performs the "walk the loop" function at initialization, it configures the first SSA adapter that appears as ssa0, the next as ssa1, and so on. For each adapter, the SSA disks connected to it are configured in serial number order, as displayed on the front of each drive. This is not necessarily the order of the disks going around the loop.

SSA Domains

Figure 5 illustrates the concept of SSA domains; each domain is circled with a dotted line. Note that the adapter closest to a disk is in the same domain. The domain is defined from the adapter's point of view. If there are multiple adapters in a loop, it is best to put the disks that will be accessed by a particular adapter closest to it. Then, I/O from the disks to their adapter will not interfere with the I/O from other disks to their adapter.

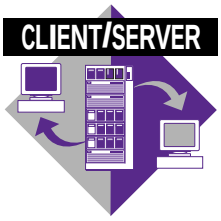
The adapter port domain means that for a particular loop of disks, those nearest to a port will be in that port's domain. For multiple computers in a loop, be sure that when you set up volume groups, the disks that comprise the group are closest to the system on which they are most likely to be used. This eliminates I/O interference between computers.

The loop arrangement enhances data availability by allowing one computer to access another computer's disks without any recabling. For example, a volume group can be varied off-line from one computer and

varied online to another. This enhances data availability if a computer fails or must be taken off-line for maintenance.

Dave E. Hall, *IBM Corporation, Langstone Road, Havant, Hampshire PO9 1SA, England. Mr. Hall works in the SSA Architecture and Development organization. He has a BSc in Mathematics from Brunel University.*

Message Passing on the RS/6000 SP



By Xianneng Shen and David Klepacki

This article is the second in a series about parallel programming models on the RS/6000 SP system. The focus is on the message-passing model and its features. In particular, unilateral (one-sided) communication is compared and contrasted to bilateral (two-sided) communication.

The first article[1] in this series briefly introduced the IBM RS/6000 SP architecture and its parallel programming models. Basic functions of the Message Passing Interface (MPI) standard were presented in another recent article[2]. Here, we discuss additional aspects of the message passing programming model as implemented on the RS/6000 SP.

Message-passing programming on the RS/6000 SP is performed with the IBM Parallel Environment (PE) product for AIX. PE allows users to develop, profile, debug, analyze, tune, trace, and execute parallel application programs.

PE supports two data communication protocols: the TCP/IP protocol and the User Space protocol. The TCP/IP protocol is the industry standard most commonly used in internetworking computers. The User Space Protocol (USP) is unique to the RS/6000 SP and provides a high-bandwidth and low-latency communication path to applications running over the high-performance SP switch network. Hence, USP is only meaningful if there is a high-performance SP switch available in the system (the SP switch is an optional hardware component of an RS/6000 SP).

USP allows applications to take full advantage of the RS/6000 SP switch communication network and offers the highest performance. However, this protocol cannot be used by more than one process per node (that is, per switch adapter) at a given time. On the other hand, the TCP/IP protocol has a different set of limitations. It can support multiple parallel applications running simultaneously on the same set of processing nodes, but at lower communication performance relative to using USP.

The MPI interface is a parallel programming library that specifies the functions and subroutine calls to be used from within the Fortran and C languages. Its specification is an international standard with a goal of not compromising efficiency, portability, and functionality for any given computer architecture or vendor. The IBM implementation of the MPI interface constitutes a library component of the PE product. It is important to remember that the protocols used for interprocessor communication are not part of the MPI specification. The IBM PE and its USP design are features that allow MPI-based applications to take full

[1]Klepacki, David and Shen, Xianneng. "Parallel Programming Models on the IBM RS/6000 SP." *AIXpert,* September 1997.
[2]Shen, Xianneng; Ho, Eddie; and Hammill, Mike. "Message Passing Interface for RS/6000 SP."*AIXpert,* March 1997.

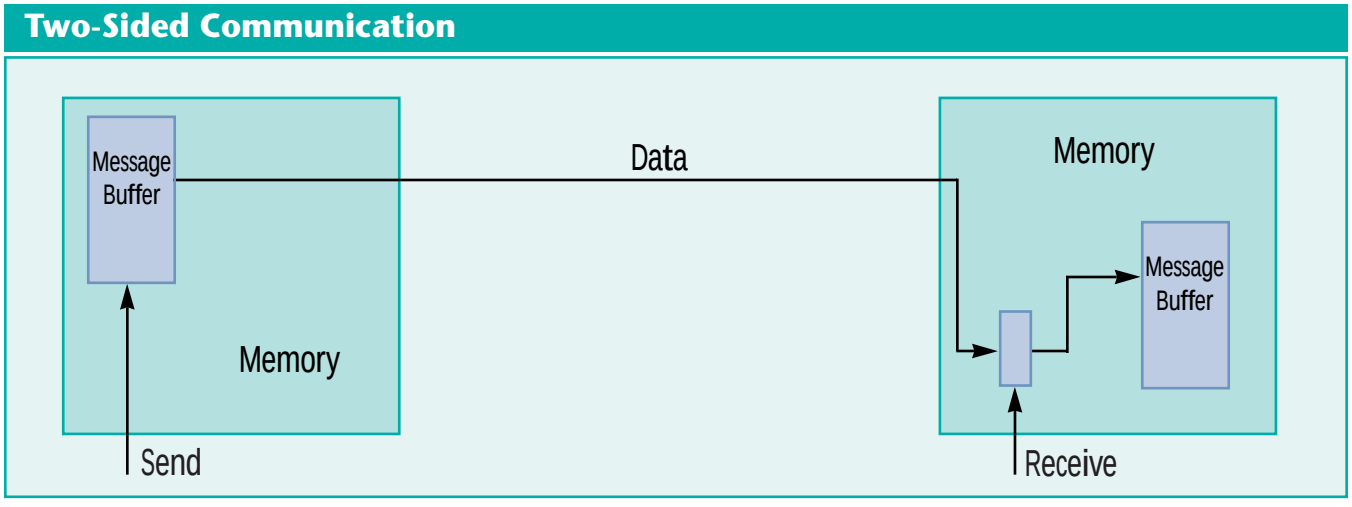


Figure 1. Send and receive

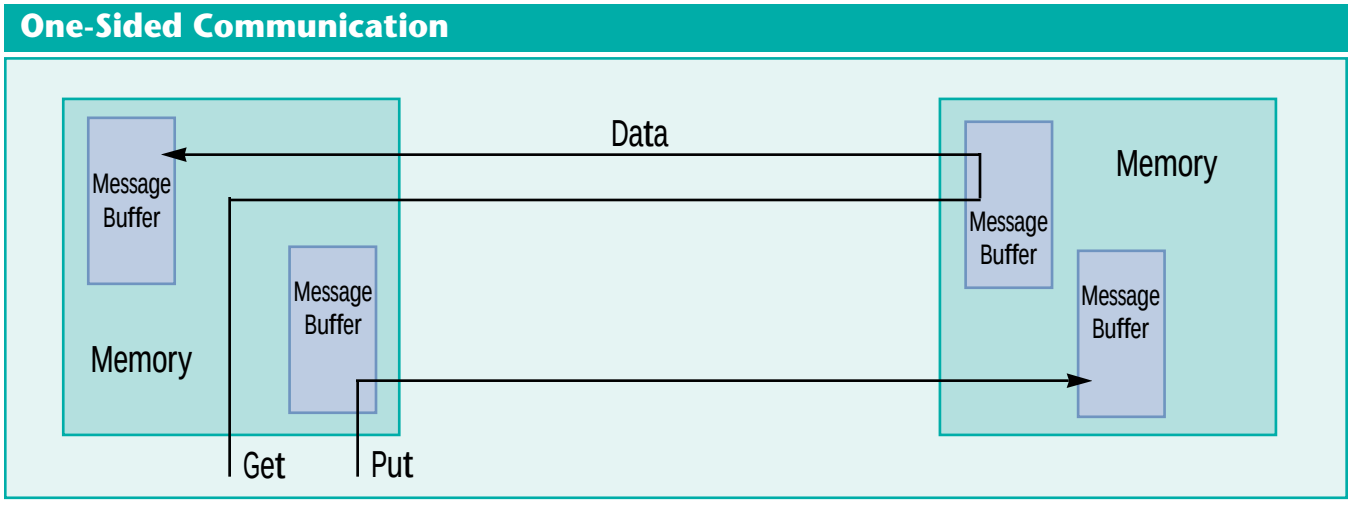


Figure 2. Put and get (one-sided communication)

advantage of the hardware capabilities of the RS/6000 SP.

MPI Enhancement

The message-passing programming model is simply described as a collection of processes that communicate among themselves using messages (that is, a sequence of bytes). The communication can be either unilateral (one-sided) or bilateral (two-sided). Bilateral communication consists of one process which initiates an operation (for example, `send`) and another process which completes the communication with a complementary operation (for example, `receive`). In fact, bilateral message passing is often referred to as a "send-receive" model (see Figure 1). In contrast, unilateral communication does not require a second process to take a complementary action. It is often referred to as a "put-get" model, whereby a `put` operation pushes messages and a `get` operation pulls messages to or from a processor's memory (see Figure 2).

In principle, a message-passing program requires nothing beyond the use of the basic `send-receive` or `put-get` functions. However, the MPI standard offers a rich set

of programming features to enhance the power and ease-of-use of message passing. In particular, a set of collective operations is defined which can be applied to user-defined groups of processes. These operations include various data movement operations (for example, broadcast, scatter, gather, all-to-all, all-to-one) as well as collective computation operations (that is, "global" sum, minimum, maximum, and so on). Additional features of MPI include virtual process topologies (such as graphs and Cartesian grids), asynchronous communication modes, and derived datatypes. Altogether, there are approximately 150 functions defined in the current MPI specification.

The MPI specification is now in its second release and is appropriately referred to as MPI-2. MPI-2 incorporates many new functions and enhancements based upon the experience gained from the first specification. The additional features can be classified into four areas: dynamic task management, unilateral communication, a parallel I/O interface, and miscellaneous functions.

Dynamic task management allows a parallel application to alter the number of concurrent processes during the course of its execution. In MPI-1, the number of concurrent processes is fixed at runtime. With MPI-2 on the RS/6000 SP, scalability can be controlled since the number of actual nodes can change to accommodate the change in requested processes. This feature allows independent MPI jobs to interact concurrently as well.

The unilateral communication features of the MPI-2 are very similar to the `put-get` semantics described above with the additional caveat that synchronization is explicitly required.

The parallel I/O interface specification, aptly called MPI-IO, introduces a portable method of efficiently working with various parallel file systems. It allows multiple processes to collectively operate on parallel files in a seamless fashion, and be able to take full advantage of MPI-related efficiencies (for example, MPI-derived datatypes).

Lastly, the miscellaneous category of the MPI-2 includes features such as non-blocking collective communication operations and the handling of external interfaces.

> ***The message-passing programming model is a collection of processes that communicate among themselves using messages, that is, a sequence of bytes.***

IBM Unilateral Communication

In addition to MPI, IBM offers a separate library for unilateral communication, called the Low-level Application Programming Interface (LAPI). LAPI is completely independent of MPI, but can coexist with MPI in the same application. Its purpose is to provide a high-performance alternative to the specification given by MPI-2. This performance improvement can be obtained by relaxing some of the MPI-2 requirements that fully support the MPI message-passing semantics. For portability purposes, the programmer can easily substitute the corresponding MPI-2 functions. LAPI is designed for use by libraries and power programmers for whom performance is more important than code portability.

The LAPI library provides `PUT` and `GET` functions and a general active message function that allows programmers to supply extensions by means of additions to the message notification handlers. LAPI has the following advantages over similar programming interfaces (like MPI):

- `Performance`: LAPI is designed to especially provide low latency on short messages, low interrupt latency, and high bandwidth.
- `Flexibility`: LAPI provides a more primitive interface (than either MPI or TCP/IP) to the SP switch that coexists with the standard communications protocols.

- Extendibility: LAPI supports programmer-defined handlers that are invoked when a message arrives. Programmers can customize LAPI to their specific environments.

Some other general characteristics of LAPI include the following:

- Reliability (LAPI provides guaranteed delivery of messages. Errors not directly related to the application are not propagated back to the application.)
- Flow control
- Support for large messages
- Non-blocking calls
- Interrupt and polling modes
- Efficient exploitation of switch function
- By default, ordering of messages is not guaranteed.

LAPI functions (see Figure 3) can be divided into three categories:

1. A basic active message infrastructure that allows programmers to install a set of handlers that is invoked and executed in the address space of a target process on behalf of the process originating the active message. This interface has enhanced capabilities for user-written handlers that allow decoupling of the tasks for data transfer, incorporating incoming data into ongoing computation and synchronization. This generic interface allows programmers to customize the LAPI function to their unique environments.

2. A set of defined functions that is complete enough to satisfy the requirements of most programmers. These defined functions make LAPI more usable and, at the same time, lend themselves to efficient implementation because their syntax and semantics are well-known.

Operations	Functions
Setup functions	`LAPI_Init and LAPI_Term`
Active Message function	`LAPI_Amsend`
Data Transfer functions	`LAPI_Put, LAPI_Get`
Synchronizing functions	`LAPI_Rmw`
Signaling functions	`LAPI_Setcntr, LAPI_Waitcntr, LAPI_Getcntr`
Ordering functions	`LAPI_Fence, LAPI_Gfence`
Address Exchange function	`LAPI_Address_init`
Environment functions	`LAPI_Qenv, LAPI_Senv`

Figure 3. LAPI functions

3. A set of control functions for the initialization and eventual orderly shutdown of LAPI and to query the state of the LAPI subsystem.

Conclusion

A few words should be said about thread-based library implementations. With the advent of Symmetric Multiprocessor (SMP) nodes on the RS/6000 SP (that is, high nodes), it becomes necessary to support threaded application code with libraries that are thread safe. This means that one thread of execution will not interfere with the operation of another thread. For instance, when dynamically allocating storage with `malloc()`, a non-thread-safe library could not guarantee that two threads would not end up allocating the same storage locations and writing over one another; a thread-safe library prevents this from happening.

Both LAPI and the IBM implementation of MPI provide thread-safe libraries. However, a thread-safe library does not guarantee a thread-safe application, and the programmer must still apply thread-safe techniques when developing parallel programs.

Acknowledgments

The authors would like to thank the following developers who took the time to explain the richness of their work: Paul DiNicola, Rama Govindaraju, Chulho Kim, Gautam

Shah, Jamshed Mirza, Carl Bender, Kevin Gildea, and Richard Treumann.

Xianneng Shen, *IBM Corporation, 522 South Road, Poughkeepsie, NY 12601. E-mail: xshen@us.ibm.com. Dr. Shen is a senior marketing support representative 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 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.*

Shell Programming—JavaBeans Style

By Jim Knutson

Java is being used with increasing frequency on UNIX systems for smart Web forms, backend servlets, GUI frontends to applications, and Web-enabling enterprise applications. This article compares and contrasts Sun's JavaBeans component model with shell programming to show how easy it can be to build up Java applets and applications with very little knowledge of programming.

Shell programming is a rudimentary form of component programming. Many of the features available in today's component models have been in use since the early '70s when UNIX® first appeared. This article will cover concepts and terminology related to component programming with the JavaBeans component model and show their similarity to shell programming. The natural progression from shell to component programming is learning how components can be snapped together easily.

Component Programming in Shell Code

Shell programming makes it possible to compose multiple commands (components) together to perform a sequence of actions. The JavaBeans component model does the same.

Suppose we wanted to offer a menu-driven interface to allow users to talk to each other. A first-pass shell program might look similar to the following:

```
#!/bin/sh
who | sed 's/ .*//'
echo "Enter user: "
read user
talk $user
```

Note that the `who`, `sed`, and `talk` commands are being glued together using a scripting language (the scripting language of the Bourne shell). The JavaBeans component model supports two methods of connecting bean components: scripting and event connections. JavaBeans does not include scripting language support, but several bean-builder tools support the use of scripting to compose beans together. Most of these bean-builder tools currently support only a single scripting language (usually Java™), but IBM's BeanExtender technology offers support for multiple scripting languages, including Java and NetRexx, with the capability of adding other scripting languages.

In the shell script, the components perform these functions: data access, data filtering, and presentation. Though not required, beans typically separate those functions into separate components, which reduces the runtime requirements when presentation is not needed. It also allows for different styles of presentation components to plug in and display the information. Similar to UNIX commands and filters, bean components work best when designed to do one thing and do it well.

Jim Knutson

A third consideration is how component execution can be altered. Shell commands take arguments to affect their execution. JavaBeans offers a similar function through properties. However, properties offer a much richer function for controlling components than simple arguments. Any component interested in determining when the value of a property in another component changes can listen for the change. It can veto that change as well.

In shell programming two of the components are exchanging data using the standard I/O paradigm of a single data producer and single consumer. JavaBeans has similar concepts for I/O, but the typical way to exchange data is through Events. When a bean component has altered its state or wishes to make data available to another component, it generates an event with the data in the event or available for access from a property (which could also generate an event when its value changes). This event can be distributed to an arbitrary number of consumers.

In shell programming, it is difficult to capture the state of a component so that it can be restarted from that state at a later time. In JavaBeans, it is not only easy to do, but quite natural. Anyone who has dealt with the state management of the UNIX accounting system will likely see the advantages.

Finally, the way input is obtained from the user differs. Nothing in the JavaBean component model predefines the method of obtaining input. Suffice it to say that a component could perform the function of gathering input.

The Beans Component Alternative

How would we use beans to compose something that performed a function similar to the above? Assume a bean, called `WhosOn`, knows how to obtain a list of users on a system, and another bean, called `TalkToMe`, knows how to talk to a user. Using the BeanExtender technology from IBM, we would find those two components and drop them onto the visual assembly surface. We also need a way to present the list of users on the system; to do that, we use a presentation component related to `WhosOn` called `WhosOnViewer`.

The `WhosOn` component is designed so that it constantly monitors the system and generates a `WhosOnEvent` every time someone logs on or logs off the system. The `WhosOnEvent` has an array of users currently on the system as part of its data. The `WhosOn` component has a corresponding presentation component called `WhosOnViewer`, which consumes `WhosOnEvents`. It is a simple matter of point and click using BeanExtender's event connection view to connect the source of an event to any number of event sinks. See Figure 1.

The close relationship of events, event producers, and event consumers brings type safety into connecting components together. You can be relatively assured that a consumer of an event will know exactly what to do with the event data given to it.

The component, as written so far, will give us a real-time view of the users who are logged on. Now we need the ability to choose the users to talk to, then talk to them. Fortunately, the `WhosOnViewer` component generates a `UserSelectedEvent` whenever an item in its view is selected. This time we use NetRexx scripting to take a `UserSelectedEvent`, get the value of the event's user property, and set a property on the `TalkToMe` component before invoking a method to open the connection to the other user. See Figure 2.

This brings us to our next topic: What happens if the person refuses to talk to us?

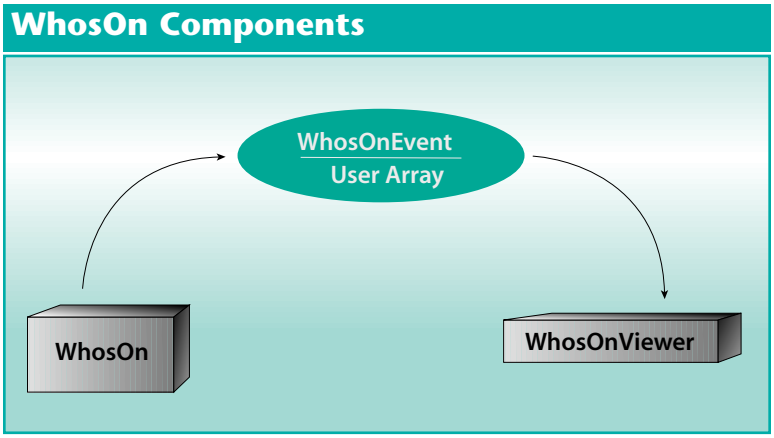


Figure 1. WhosOn components

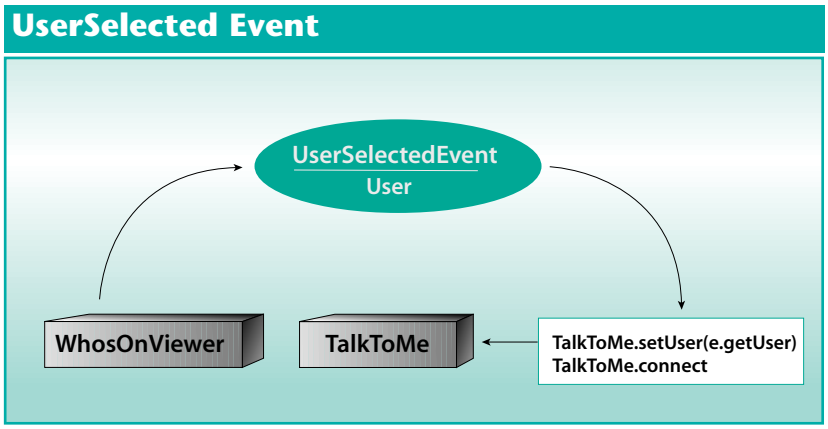


Figure 2. UserSelected Event

In shell programming, components relay failure information through return codes and signals. Java-based components communicate failure information through exceptions, which is similar to signal handling in shell programming.

When using pre-built components, you will most likely encounter this situation when writing script code using a class that throws exceptions. As a component programmer, you must decide whether to handle the exception directly in your script code using a try-catch block or pass it on to someone else. Usually you will need to handle it in your script code.

Some Java language features affect bean component programmers; for example, Java components may be used concurrently. Just as shell programming supports the notion of concurrent programming through the use of background processes, Java supports multiple threads of execution. This is easy to do, and Java programmers should expect their components to run multithreaded. Careful use of synchronization can avoid problems associated with running in a multithreaded environment.

How do environment variables in shell programming translate into beans component programming? Environment variables are often used as a persistent method for defining arguments to shell commands. Java supports a similar notion using "system" properties, which can be made persistent across invocations of Java applets and applications. Examples of these can be found in the Java Development Kit (JDK) library directory.

Bean Husbandry 101

If bean component programming is so easy, particularly with bean-builder tools such as IBM BeanExtender, why hasn't it become even more popular? The simplest answer is time.

The JavaBean specification, which describes how to write beans so that they can be composed, was not available until early this year. In addition, many Java developers do not understand what they need to do to turn their classes and applets into beans that can be composed with other beans.

Lastly, tools support for beans has been, until recently, nearly non-existent. We shall examine one more example to show how easy it is to make bean components, particularly with bean-aware builder technology such as IBM BeanExtender.

A common occurrence in shell programming is a loop that performs a repeated periodic function. The shell code typically looks like the following:

```
while true; do
  <function to perform>
  sleep <some interval>
done
```

Although this works fairly well, it cannot guarantee that the function is performed at specific intervals. It only

guarantees that an interval amount of time has occurred between the time when the last round of function ended and a new round starts. It is easy to build a bean component to provide similar functionality by using IBM BeanExtender.

Figure 3. Defining a new property

Figure 4. Adding a new event

Our new `Ticker` component will generate an event at specific intervals defined by a property that can be set. It runs as a separate thread; therefore, it is derived from the `java.lang.Thread class`. With BeanExtender, we can set the parent class of the component we are building by simply typing in the name of the parent class.

Next, we need to define a settable property to govern the interval between events. The Publish view of BeanExtender lets us define new properties easily by defining a name and type for the property. In this case we will define a property called *delay* with a type of long to represent milliseconds between event generation, shown in Figure 3. We also decide to be good bean citizens and make the property bound and constrained so that other components can be notified of an event change and optionally veto the change.

Next, we decide the delay needs to default to one second, so we select the constructor method, Ticker, in the Publish view for editing, and add the line `delay = 1000`, to the constructor.

To generate an event, we must first define an event to generate. The Publish view lets us create events as easily as we create properties. We generate a new event called `TickerEvent`. We decide that we do not need to supply any data items with the event and the method, which all sinks of the `TickerEvent` will need to implement to handle the event, will be called `tickerTocked`, shown in Figure 4. Defining the event allowed us to do the following:

- Generate the `TickerEvent` class
- Generate the `TickerEventListener` interface, which is implemented by components that want to be `TickerEvent` sinks
- Add a `fireTickerTocked` method to our Ticker bean component to handle the distribution of the event to everyone wanting to hear about it

The final task to complete our component is to write the code that actually generates the event. To do this, we publish a new public `void run()` method with the code shown in Figure 5.

This method overrides the run method of the `Thread` parent class and defines the function for the thread to perform. In this case, wait until the appropriate time to generate a new event and then fire it.

```
while(true) {
  try {  sleep(delay);  }
  catch (java.lang.InterruptedException e) { }
  fireTickerTocked(new TickerEvent(this));
}
```

Figure 5. Code to generate event

Conclusion

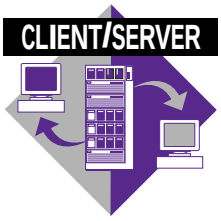
As you can see, with the appropriate tools and a little knowledge, it is easy to create a bean component within a short time. In this example, we created a new fully functional bean component supporting properties and events with six lines of code and a couple minutes of work. This new component can then be used to compose with other components to provide higher level function. One example of this is to take a few progress bar beans arranged vertically, along with some script code and a Ticker bean, to implement a visual representation of an egg timer. Another possible use for the Ticker bean would be to drive a ticker tape component.

BeanExtender is a technology preview available from IBM providing support for beans programming and deployment. It continues to mature and pieces of the technology are expected to appear in several IBM products.

Evaluation copies of BeanExtender can be found on the IBM VisualAge™ WebRunner home page at: *http://www.taligent.com/Products/webrunner/webhome.html*.
Additional information regarding BeanExtender can be found on the IBM BeanExtender Technology Web page at: *http://www.software.ibm.com/ad/javabeans/beanextender/*.

Jim Knutson, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Knutson is the technical lead and a development manager of the BeanExtender development group. He has worked at several companies, including the Microelectronics and Computer Technology Corporation (MCC) before joining IBM in 1993. He has a BA in Computer Science and 20 years of computer industry experience.*

XL Fortran Compiler for IBM SMP Systems



By Dattatraya H. Kulkarni, Sudarsan Tandri, Lisa Martin, Nawal Copty, Raul Silvera, Xin-Min Tian, Xing Xue, and Julian Wang

This article describes salient features of the XL Fortran V5 compiler, which leverages parallelism in Symmetric Multiprocessor (SMP) systems for improving the performance of scientific applications. The article also includes guidelines for performance debugging and tuning for effective parallelization.

The XL Fortran V5 compiler accepts full Fortran 90 as well as several new language features that support parallelization and performance tuning on Symmetric Multiprocessors (SMP). Nested loops, the major source of parallelism in scientific applications, account for a substantial portion of execution time. The XL Fortran compiler can be used to automatically identify the parallel loops. In addition, programmers can guide the compiler by providing directives to indicate available parallelism. In order to enable portability and code migration, the XL Fortran compiler includes user directives from the OpenMP specification. OpenMP is supported, or is planned to be supported, by many vendors, including SGI, DEC®, KAI, and IBM.

The compiler uses a fork-join computation model, implemented using POSIX™ threads, for executing parallel loops. The fork-join model enables the implementation of several scheduling strategies to improve locality and load balance, and an effective parallel execution that is tolerant to load variance in multi-user environments.

In this article, we outline the architecture of the XL Fortran V5 compiler and briefly describe the functionality of the components. Second, we will describe automatic parallelization of nested loops in the compiler. We will illustrate examples of programmer support in the XL Fortran V5 compiler to express parallelism. We describe the methodology used to implement the fork-join computation model for executing parallel loops on the SMP processors.

The XL Fortran compiler performs a seamless set of optimizations for competitive performance on both uniprocessor and multiprocessor systems. We believe that these optimizations are useful for the programmer to hand optimize the code as well. We also summarize the performance of the compiler on standard serial and parallel benchmarks.

SMP Architecture for Parallel Computation

A symmetric multiprocessor system has multiple processors that access a global shared memory. It typically has 2 to 16 processors, which share a single address space (shown in Figure 1). More importantly, each processor in an SMP requires the same number of cycles to access a data item from the global shared memory in the absence of contention.

SMP systems are attractive platforms for parallel computation because they provide the shared memory computation model that is easier to program. Compared with a message passing computation model, they provide a simpler path to migrate serial applications onto parallel platforms. They can also be used for other general-purpose

computational needs, such as enterprise computing.

Because scientific applications tend to be computation and resource intensive, there is a tremendous advantage in porting these applications to parallel hardware. A majority of these applications were originally written for uniprocessor systems, which means they cannot exploit the power of SMPs in their current form. Fortunately, nested loops account for a substantial portion of execution time, and they can potentially be executed in parallel across multiple processors of an SMP. Ideally, a compiler should identify all the parallel loops and optimize the application for good overall performance.

Parallelism and the XL Fortran Compiler

The XL Fortran V5 compiler helps exploit the parallelism in IBM SMP hardware to improve the performance of scientific applications. It can automatically identify parallel loops and generate code to execute them across the processors of the SMP. However, scientific applications can have nested loops that cannot be automatically parallelized by the current compiler technology.

The XL Fortran compiler provides directives to help users guide the compiler in parallelization. The compiler also provides directives to identify critical sections of code and indicate policies for balancing the workload across SMP processors. Thus, the XL Fortran V5 compiler serves two purposes:

- It automatically parallelizes nested loops for faster porting of serial applications onto SMP systems.
- It has user directives that guide parallelization for careful performance tuning of the applications. These user directives conform to the directives provided by most vendors.

The XL Fortran V5 compiler also supports explicit multithreaded programming with a Fortran 90 interface to the AIX `pthread` library. The compiler supports native Fortran constructs, such as the `CHARACTER` data type and array language constructs, which can be used with the `pthread` interface.

Another new feature introduced in XL Fortran V5 is asynchronous I/O, which is needed for speed and efficiency in large scientific applications. The performance benefits result from the ability to overlap the input and output of large chunks of data with the execution of other statements.

Symmetric Multiprocessor Architecture

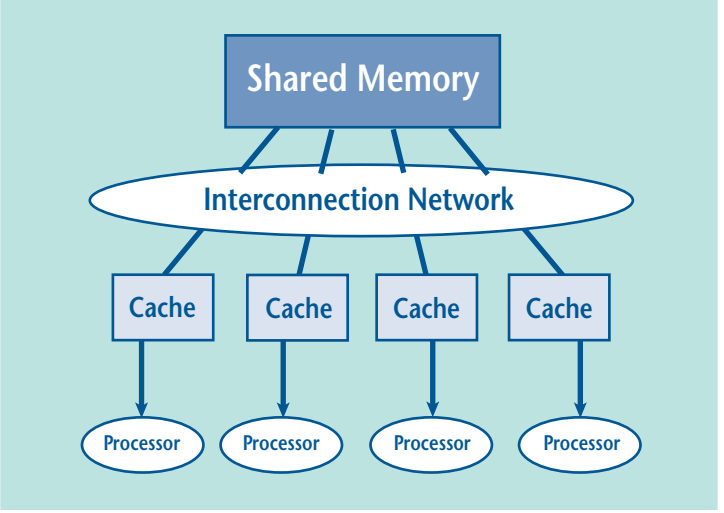


Figure 1. Symmetric multiprocessor architecture

The Compiler Architecture

The architecture of the XL Fortran V5 compiler is as shown in Figure 2. The internals of the XL Fortran compiler have four main component groups. The first group is the Fortran 90 frontend, which takes as input the user's source program, which may be annotated with directives for SMP parallelization. The frontend produces a high-level intermediate representation of the code for use by subsequent phases. In addition, the frontend ensures that the source conforms syntactically and semantically to the Fortran language definition and emits error messages for invalid source statements.

The language supported by the frontend includes full Fortran 90, several industry extensions, as well as selected Fortran 95 features. FORTRAN 77 is a subset of the Fortran

90 language definition, so the XL Fortran compiler supports full FORTRAN 77 as well.

The second group contains the scalarizer for converting the array language statements of Fortran 90 into FORTRAN 77 loops, as well as locality optimizer, serial and SMP optimizer, parallelizer, and finally, the outliner.

The third group contains components that support the actions of the components in the second group. The components in this group analyze the data flow and dependence in the program, and implement loop and data transformations required by the components in the second group.

The final group consists of the optimizing backend, which produces object code from the internal representation of the parallelized input program, and the SMP runtime.

Scalarizer

The scalarizer transforms the high-level intermediate representation of Fortran 90 array language statements into scalar `DO` loops. Through the use of data dependency information, the scalarizer eliminates temporaries, where possible, to generate efficient code for array language statements. The scalarizer also generates inline code for most Fortran 90 array intrinsics. Inlining of Fortran 90 intrinsics eliminates explicit calls, extra array temporaries, and array copying, which helps Fortran 90 programs to achieve performance comparable to FORTRAN 77 programs.

Invoking the scalarizer before parallelization is an important optimization. This is because the compiler can exploit parallelism in nested loops generated by the scalarizer as well as nested loops coded explicitly by the user.

Automatic Parallelization of Nested Loops

The locality optimizer, other optimizers for serial and parallel execution, and the parallelizer are within the high-order transformation framework of the compiler. The optimizations are robust in that they systematically combine transformations for improved serial execution with transformations for exploiting maximum parallelism.

Fortran V5 Compiler Architecture

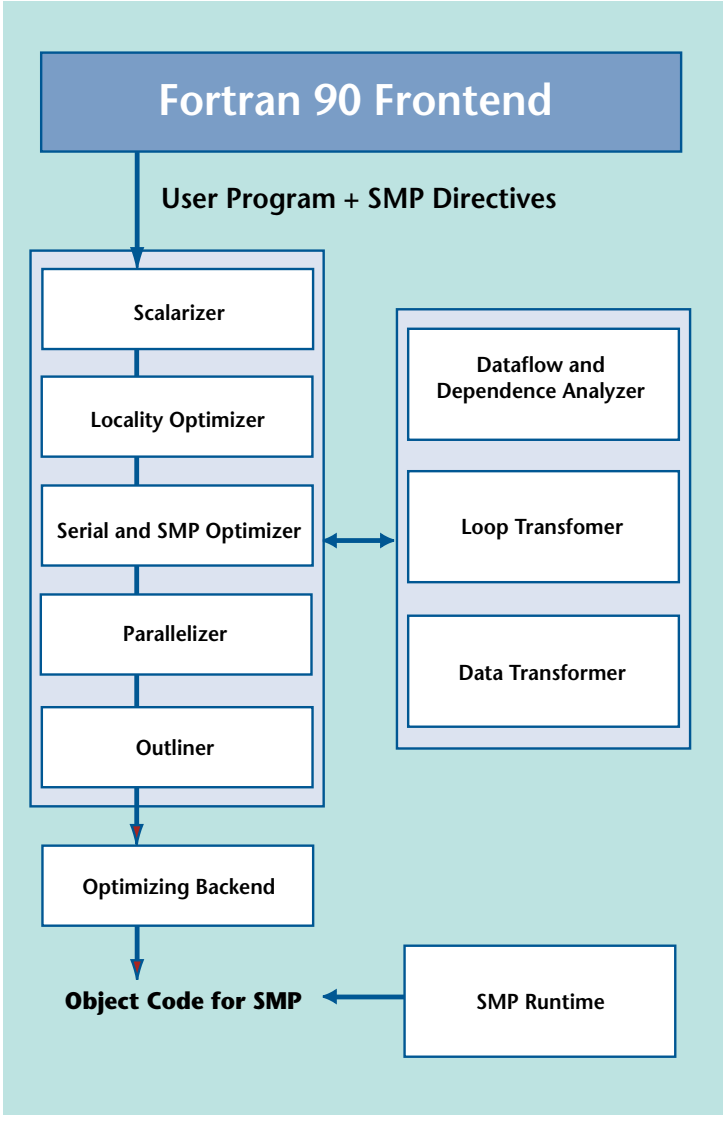


Figure 2. XL Fortran V5 compiler architecture

The locality and other optimizers use loop reordering and array padding transformations in order to improve the overall program performance. For example, nested loops are reordered so that the arrays are accessed with the smallest stride first to improve cache locality. Other optimizations performed within the compiler include loop tiling, loop unrolling, replacement of array references by references to temporary scalars, and elimination of conditionals. Most of these optimizations use cost models

that are parameterized by the hardware characteristics of the target system.

The parallelizer uses loop reordering transformations to automatically parallelize loops at the outermost levels. This minimizes parallelization overheads and ensures larger computation granularity on each of the processors of the SMP system.

Outlining

Outlining is a program transformation technique that is the converse of subroutine inlining. Whereas inlining a subroutine inserts the code in the subroutine at the call site, outlining converts a region of code into a subroutine and replaces the region of code by a call to the subroutine.

The outliner in the XL Fortran compiler converts parallel loops identified either by the compiler or specified by the user into subroutines. These subroutines implement a fork-join model for executing parallel loops with the help of calls to the SMP runtime library.

Outlining is a key ingredient in allowing scheduling schemes that make the performance of applications tolerant to load variations in multi-user SMP environments.

Augmenter

The main tasks of the augmenter include:

- Implementing XL Fortran's parameter passing conventions
- Creating runtime descriptors and implementing operations that require knowledge of their contents, including pointer assignment, `ALLOCATE`, and `DEALLOCATE`
- Inlining and generating calls to a variety of XL Fortran intrinsic functions

Automatic Parallelization

Nested loops in Fortran programs have long been recognized as an important source of parallelism. A majority of these loops have control structures and data access patterns that have made formal frameworks for automatic transformation and parallelization feasible. Over the past several years, the technology for automatically parallelizing nested loops has matured. The XL Fortran V5 compiler incorporates a substantial amount of this technology for parallelizing Fortran 90 programs on IBM SMP systems.

The loop and data transformer, locality optimizer, serial and SMP optimizer, and parallelizer interact with each other extensively. These components are collectively called the transformer in the XL Fortran V5 compiler. Input to the transformer is an internal representation of the scalarized Fortran 90 program. The transformer optimizes the input program with a perfectly nested loop at a time, with the help of the data flow and dependence analyzer. Optimizations in the transformer have five main purposes:

- Identify parallel loops automatically
- Derive transformations that improve cache locality
- Perform optimizations that target certain serial and SMP-specific optimizations, such as replacing invariant array references by scalars and recognizing scalar and array reductions
- Check viability and legality of parallelism due to assertion directives on the loop such as the `INDEPENDENT` directive (see "Assertion Directives" below)
- Retain parallelism on the loop due to prescriptive directives, such as `PARALLEL DO`

The basis for the loop transformations is the *unimodular transformation* framework that enables loop interchange, skew, reversal, and permutations to be applied uniformly. As a first step, the transformer distributes the loops to maximize opportunities for parallelization. The loop fusion that follows locality optimization and parallelization fuses the loops together, in case it is beneficial and legal to do so. Loop fusion increases the granularity of computation in each iteration

and may improve cache performance and instruction-level parallelism.

Locality optimization is central to the transformer, where nested loops are transformed so that arrays are accessed with the smallest stride first. A loop nest may be tiled, if necessary, for improving cache performance. Locality optimization also identifies a group of innermost loops, called the *locality group*, where all the array accesses are contained in the cache. The later optimizations and parallelization attempt to preserve the locality group for continued locality of access.

The transformer uses unimodular transformations to expose available parallelism in outermost loops. While doing this, loops belonging to locality groups are parallelized only if the transformer cannot parallelize other loops.

An important aspect of the parallelization technique in the transformer is the use of information about the loop provided by the programmer. For a loop with the `INDEPENDENT` directive, the loop is parallelized unless the dependence analysis finds precise information that the loop iterations are dependent.

For a loop with the `CNCALL` directive, the loop is parallelized unless there is a dependence due to references other than the arguments to the subroutine and the call itself. For a loop with the `PERMUTATION` directive, the dependence analyzer assumes that there is no dependence due to array references indexed by the permutation variable. In some instances, two array references indexed by a permutation variable may refer to the same memory location—the loop is not parallelized in such cases.

The transformer assumes that all loops with the `PARALLEL DO` directive are indeed parallel. In fact, the transformer may prevent certain optimizations and reordering transformations to preserve the semantics of the `PARALLEL DO` directive.

The transformer also unrolls both inner and outer loops for improved instruction schedules. In order to eliminate repeated index computations, scalars replace invariant accesses to arrays.

Support for Programmer-Driven Parallelization

XL Fortran V5 supports programmer-driven parallelization of applications. This support is useful in tuning the application performance with the automatic parallelization in the compiler. The support enables programmers to express both data and task parallelism. They can improve data parallelism by providing hints to the compiler on parallelism in nested loops. They can implement task parallelism by creating independent threads of control, which may have data or task parallelism.

User Directives for Data Parallelism

XL Fortran V5 provides two classes of SMP directives: *assertion directives* and *prescriptive directives*. The assertion directives are *hints* to the compiler, whereas the prescriptive directives are *directions* to the compiler.

Assertion directives provide the compiler with additional information about loop characteristics. The automatic parallelizer can use this information to determine whether a given loop should be parallelized. Prescriptive directives can help in forcing the compiler to parallelize or to mark sections of code as critical. Thus, prescriptive directives provide users with more control over which sections of code are to be parallelized or which shared resources are to be protected by simultaneous access. The assertion and prescriptive directives supported in the XL Fortran V5 compiler are shown in Figure 3.

SMP directives are Fortran comment lines that begin with a special character sequence called a *trigger*. The compiler

Programmer Directives in XL Fortran

Assertion Directives	INDEPENDENT CNCALL PERMUTATION ASSERT
Prescriptive Directives	PARALLEL DO PARALLEL SECTIONS CRITICAL

Figure 3. Programmer directives in XL Fortran

```
      x = 0
!SMP$ INDEPENDENT, REDUCTION(X), NEW(I)    ! The iterations of the loop
      do i = 1,m                           ! are independent, where x is
        x = x + i**2                       ! a reduction variable.
      end do                               ! i is private to each iteration.
```

Figure 4. INDEPENDENT directive

```
!SMP$ ASSERT (ITERCNT(100), NODEPS)  ! The loop has approx. 100 iterations
      do i = 1,n,step                ! and there are no dependencies between
                                     ! iterations.
        a(i) = a(i-m) * parm
      end do
```

Figure 5. ASSERT directive

```
!SMP$ CNCALL
      do i = 1,n
        call no_side_fx(a(i),b)
      enddo
```

Figure 6. Concurrently called procedures

recognizes several default triggers when SMP compilation is requested (for example, `SMP$` and `$OMP`).

Comment lines beginning with such triggers are processed as SMP directives. By default (when SMP compilation is not requested), the triggers are not recognized and the lines with directives are treated as comments. This is important because it allows users to add directives to a program and compile it for either serial or parallel execution.

Assertion Directives

The following describes several assertion directives.

INDEPENDENT: This directive is adopted from High Performance Fortran. The `INDEPENDENT` directive asserts that the iterations of the `DO` loop that follows can be executed in parallel. It supports two clauses:

- `NEW` clause with a list of variables specifies that new objects should be created for variables in the list
- `REDUCTION` clause that lists variables appearing in reduction operations

Figure 4 shows an `INDEPENDENT` directive.

ASSERT: The `ASSERT` directive is an extension of the `-qassert` compiler option. It can indicate to the compiler that no data dependencies exist between iterations of a loop. It can also approximate the iteration count for a loop with runtime bounds, which can help the compiler determine whether parallelizing a given loop would be beneficial. Figure 5 shows an example of the `ASSERT` directive.

CNCALL: This directive asserts that procedures invoked within the loop can be called concurrently by separate iterations of the loop. Figure 6 shows an example.

```
!SMP$ PERMUTATION(index)
      do i = 1,n
         a(index(i)) = a(index(i)) + b      ! There are no data dependencies
                                            ! between iterations of this loop
                                            ! if index has no repeated values.
      enddo
```

Figure 7. PERMUTATION directive

```
!SMP$ PARALLEL DO PRIVATE(i), SHARED(a), REDUCTION(s), &
!SMP$              IF(n > 100), SCHEDULE(GUIDED)

      do i = 1,n                              ! This loop will be executed in
        a(i) = i*2                            ! parallel if there are more than
        s = s + a(i)                          ! 100 iterations.
      enddo
```

Figure 8. PARALLEL DO prescriptive directive

PERMUTATION: Data dependence analysis is difficult and often not possible when array subscript expressions are complex. Applications with irregular computations typically have array subscript expressions that involve references to other arrays. The PERMUTATION directive helps the compiler analyze nested loops that have such indirect array references by specifying arrays whose elements have unique values. It is difficult for the compiler to determine that a loop is independent if array element references within it use other arrays in subscript expressions. The compiler uses this directive to identify parallelism within loops, which have complex subscript expressions referencing the arrays specified in the PERMUTATION directive. Figure 7 shows an example.

The primary prescriptive directives are as follows: PARALLEL DO, CRITICAL, and PARALLEL SECTIONS.

PARALLEL DO: This directive can direct the compiler to parallelize a loop. It allows the programmer to indicate which variables should be private to the independent iterations of the loop and which variables should be shared. The directive also provides a REDUCTION clause, which is similar to the REDUCTION clause of the INDEPENDENT directive. The IF clause specifies a runtime condition under which a loop should be parallelized. The SCHEDULE clause indicates which chunking algorithm should be used to partition the loop. Figure 8 shows an example.

CRITICAL/END CRITICAL: CRITICAL sections are blocks of code that should be executed by a single thread at a time. These directives should protect access to shared resources such as shared variables. Each CRITICAL section is associated with a named lock variable; however, critical sections that do not specify explicit lock names share the same global default lock.

If a lock name is specified for more than one critical section, the compiler will allow only one thread to execute any of these sections at a time. See Figure 9 for an example.

PARALLEL SECTIONS/END PARALLEL SECTIONS: PARALLEL SECTIONS allow users to mark multiple independent sections of code for parallel execution. The PARALLEL SECTIONS construct supports clauses similar to PARALLEL DO; variables may be specified as PRIVATE, SHARED, and REDUCTION; the IF clause may be used to specify conditional parallelization. Note that although this construct is a convenient way to parallelize

```
!SMP$  PARALLEL DO  PRIVATE(I)
       do i = 1,n
          .
          .
          .
!SMP$  CRITICAL (lock_my_lib)              ! Calls to non-thread-safe
         call my_lib_routine(a,b)          ! library routines should
!SMP$  END CRITICAL (lock_my_lib)          ! be serialized
          .
          .
          .
       enddo
```

Figure 9. CRITICAL SECTION prescriptive directive

```
!SMP$  PARALLEL SECTIONS
!SMP$  SECTION
           call compute_sum(a)
!SMP$  SECTION
           call compute_min_element(a)
!SMP$  END PARALLEL SECTIONS
```

Figure 10. PARALLEL SECTIONS directive

independent blocks of code, it does not scale well since the number of parallel work items (that is, the number of `SECTION` clauses) is constant. See Figure 10.

Thread-safe Parallel I/O

The XL Fortran thread-safe I/O library, `libxlf90_r.a` , provides support for parallel execution of Fortran 90 I/O statements. It is essential to use this library in Fortran applications that contain I/O statements in a parallelized loop or in applications that create multiple threads and execute I/O statements from within different threads at the same time. In other words, such applications must be linked with this library to get the expected results.

Synchronization of I/O Operations

During parallel execution, multiple threads can perform I/O operations on the same file at the same time. If the threads are not synchronized, then the results of these I/O operations could be shuffled or merged so that the applications produce incorrect results or even crash. The XL Fortran thread-safe I/O library synchronizes I/O operations for parallel applications to ensure the integrity and correctness of each individual I/O operation. The synchronization, performed within the I/O library, is transparent to application programs. The synchronization for external files is performed on a per unit basis. When a thread is performing an I/O operation on one unit, other threads attempting to perform I/O operations on the same unit will have to wait until the first thread finishes its operation. For the purposes of I/O synchronization, all internal files are treated as though they were one single logical unit.

The XL Fortran thread-safe I/O library sets its internal locks to synchronize access to logical units. This should have no functional impact on the I/O operations performed by a Fortran program. Also, it will not impose additional restrictions to the operability of Fortran I/O statements other than what they already have, except for the use of I/O statements in a signal handler that is entered because of an asynchronous-generated signal.

Non-determinacy in Parallel I/O

The order in which parallel threads perform I/O operations is not predictable. The XL Fortran thread-safe I/O library does not control the ordering.

Parallel I/O can only be used in cases where each thread performs I/O on a predetermined record in *direct* access files, where the result of an application does not depend on the order in which records are written out or read in, or where each thread performs I/O on a different file. In these cases, results of the I/O operations are independent of the order in which threads execute.

For multiple threads to write or read the same *sequential* access file, however, the order of records written out or read in depends on the order in which the threads execute the I/O statement on them. Since this order is not predictable, the result of an application could be incorrect if it supposes records are sequentially related and cannot be written out or read in any order. For example, if the following loop:

```
do i = 1, 500
print *, i
enddo
```

is parallelized, the numbers printed out will no longer be in the sequential order from 1 to 500 as the result of a serial execution.

Outlining: Fork-Join Computation Model

Two basic approaches have been used for executing the work in different iterations of a parallel loop: the fork-join and the Single Program Multiple Data (SPMD) models. In the fork-join model of computation, a single thread—the main thread—sets up and coordinates parallel work, while all threads participate in performing the work. On the other hand, the SPMD computation model requires the compiler to set up the work for all the threads. In the fork-join model, threads are typically created at the start of work and destroyed once the work is completed. To optimize the thread creation overhead, threads are created once and they wait for work to be set up. The parallel work is divided among processors according to a loop scheduling strategy. Thread synchronization of the threads at the end of the parallel loop is implicitly accomplished by the main thread, which monitors the completion of work.

There are two reasons for this outlining. First, it creates a context for parallel execution. Secondly, it simplifies storage management, since each thread executing the subroutine will have a separate copy of the local variables automatically allocated on its stack, while non-local variables will be shared among the threads. The outlined subroutines are executed by all the participating threads with the help of a runtime library.

Figure 11 illustrates outlining of an example parallel loop, which transforms the parallel loops into a subroutine.

Optimizing Applications for SMPs

The key to optimizing applications on SMPs is improving parallelism and cache locality by either compiler optimizations or careful coding by the programmer. In this section, we first outline selected optimizations performed by the XL Fortran V5 compiler.

> ***The key to optimizing applications on SMPs is improving parallelism and cache locality by either compiler optimizations or careful coding by the programmer.***

Programmers can use these optimizations as templates to hand-tune applications on SMPs. In the second subsection, we provide broad guidelines for performance debugging and performance tuning of applications while using the XL Fortran V5 compiler.

Compiler Optimizations

The XL Fortran V5 compiler performs several optimizations to enhance both serial and parallel application performance. These optimizations are aimed at improving locality of data access and parallelism in nested loops, improving locality and balance of workload among parallel threads, and reducing access to data shared among the parallel threads.

Before Outlining

```
program user_prog
integer a(100),   i

parallel do i = 1, 100
       a(i) = i
end do

print * ,   a
end
```

After Outlining

```
program user_prog
integer    a (100) , i
integer _parDoBounds (3), _parDoChunkCtl(4) , _parDoStep

_parDoBounds    * . . .
_parDoChunkCtl = . . .
_parDoStop        = . . .
call _xlsmpParDoSetup (_parDoBounds, _parDoChunkCtl,
                                          _outlined_sub_1, 1. _parDoStep)
print *,  a


contains

subroutine _outlined_sub_1 (_libControl, _parDoStep)
integer _libControl, _parDoStep
integer i_1

do while (_xlsmpParDoChunk (_libControl, _parDoFrom, _parDoTo)
      do i_1 = _parDoFrom, _parDoTo, _parDoStep
           a(i_1) = i_1
      end do
end do

return
end _outlined_sub_1

end
```

Figure 11. Parallel Loop Outline

The compiler transforms nested loops in an effort to parallelize loops at the outermost loop levels. This improves the granularity of computation performed by each thread and also reduces the overhead of executing parallel iterations. While parallelizing the loops, the compiler avoids the parallelization of nested loops that must be serial for better cache locality. The runtime system has scheduling schemes that preserve locality in the parallel loops and distribute computational load evenly among the parallel threads.

Idiom recognition has been vital in exploiting the performance of multiprocessor systems. The XL Fortran V5 compiler uses scalar and array reductions and privatization as an integral part of the loop transformation process. Whenever possible, the compiler attempts to replace array references by scalars to reduce index computation overhead as well as implement certain array reductions efficiently. The compiler minimizes access to shared data by identifying variables that can be private to parallel

Benchmark #	Serial Time (s)	Parallel Time (s)
101.tomcatv	1101	727
102.swim	1719	570
103.su2cor	716	420
104.hydro2d	1442	504
107.mgrid	907	329
110.applu	955	713

Figure 12. The preliminary performance of SPECFP95 benchmarks with the XL Fortran V5 compiler on a 4-processor J40 SMP system

threads. The compiler also has techniques to identify repeated calls to intrinsics and replace them by efficient routines that apply intrinsics to a vector of array elements.

Performance Debugging and Tuning

The XL Fortran V5 compiler can generate a report on the parallelization and locality enhancements performed by the compiler by invoking the `-qreport=smplist` command-line option. The generated report is a pseudo-Fortran listing of the transformed input program. In this report, parallel loops are explicitly identified with a `PARALLEL DO`.

The report contains the compiler's reasons for not parallelizing certain loops. These reasons are useful as pointers for identifying the directives that programmers can insert to help parallelize the loops. For example, the compiler cannot parallelize a loop with a subroutine call because the compiler cannot determine the side-effects of the call. In such cases, the programmer may be able to analyze and determine that the calls to the subroutine indeed do not have any side-effects that affect the execution order of the iterations. Therefore, the programmer may insert the `CNCALL` directive to help the compiler in parallelizing the loop. However, an imprecise dependence in the loop can still prevent the compiler from parallelizing the loop. If this occurs, the programmer can analyze the loop to determine if the loop iterations are independent and insert the `INDEPENDENT` directive. Note that the `INDEPENDENT` directive is only an assertion, so the programmer may need to use the `PARALLEL DO` directive to force the compiler to parallelize the loop.

The programmer's next task is to improve the performance of parallel loops. The efficiency of a parallel loop is good when the benefits of dividing the work among the processors is much higher than the overhead of parallel execution. In particular, inner loop parallelism incurs much overhead because of barrier synchronization at the end of every parallel iteration. For this reason, parallelism in the outer loop is preferable to parallelism in the inner loops. In general, it is essential to prevent the parallelization of loops with very little computation and enable the parallelization of loops with significant computation.

Good performance results from a balance between parallelization and locality optimization. The XL Fortran V5 compiler performs several optimizations to improve locality of data access. While specifying the `PARALLEL DO` directive for a loop, it is necessary to ensure that the parallel threads access data mostly from the local cache. Sometimes, it is useful to replace the `PARALLEL DO` directive with the `INDEPENDENT` directive so that the compiler has freedom to make the trade-off between locality and parallelism in the loop.

Locks protect the access to shared resources in `CRITICAL SECTIONS`. The overhead due to these locks can be high, especially when they occur inside a nested loop.

In such cases, it may be worthwhile to investigate the need for the `CRITICAL SECTION` or to serialize the loop nest.

Performance Results

Figure 12 shows the preliminary performance of SPECFP95 benchmarks using the XL Fortran V5 compiler. The serial time corresponds to the best serial execution time without any parallel overheads. The parallel time corresponds to the execution time on an IBM SMP with four PowerPC 604™ processors.

Conclusion

We described the XL Fortran V5 compiler, which helps the programmer exploit the parallelism in IBM SMP hardware. Five features of the compiler are noteworthy:

- The compiler accepts the full Fortran 90 language as well as selected Fortran 95 features.
- It implements both advanced serial optimizations and automatic parallelization of nested loops seamlessly.
- It supports industry-accepted user directives, which help the programmer in tuning parallel program performance.
- It supports thread-safe parallel I/O.
- It provides support for task parallelism through a Fortran 90 interface to the AIX `pthreads` library.

The XL Fortran V5 compiler provides an effective means to port serial scientific applications onto SMPs and to tune parallel applications on SMPs.

Dattatraya H. Kulkarni, *IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Kulkarni is a member of the compiler team at the SWS Toronto Laboratory focusing on high-level optimizations in the XL Fortran compiler. His areas of interest include programming languages, optimizing compilers, and environments. He holds a PhD in Computer Science from the University of Toronto.*

Sudarsan Tandri, *IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Sudarsan is a member of the compiler development team at the SWS Toronto Laboratory. His main focus has been robust runtime support for parallelization on SMP systems. His areas of interest include compiler and emerging applications on multiprocessor systems. He holds a PhD in Computer Science from the University of Toronto.*

Lisa Martin, *IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Ms. Martin is a member of the compiler development team at the SWS Toronto Laboratory. She is the architect of the XL Fortran compiler. She holds a BMath degree from the University of Waterloo.*

Nawal Copty, *IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Ms. Copty is a member of the compiler development team at the SWS Toronto Laboratory. She is currently working on the outlining support for the XL Fortran compiler. Her interests include parallel languages, algorithms, and compilers, and issues in parallel and distributed computing. She has a PhD in Computer Science from Syracuse University in New York.*

Raul Silvera, *IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Silvera is a member of the compiler development team at the SWS Toronto Laboratory. He has an MS in Computer Science from McGill University in Montreal.*

Xin-Min Tian, *IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Tian is a member of the compiler development team at the SWS Toronto Laboratory. He has a PhD from the Tsinghua University in Beijing.*

Xing Xue, *IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Xue is a member of the compiler development team at the SWS Toronto Laboratory. He specializes in the I/O runtime environment and the augmentor phase of the compiler. He has a PhD in Computer Science from Nanjing University in China.*

Julian Wang, *IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Wang is a member of the compiler development team at the SWS Toronto Laboratory. His areas of interest include compiler development and the operating system. He has an MS in Computer Science from the University of Toronto.*

AIX Questions

The AIX Solution Provider Technical Support Group in Austin, Texas, supports software vendors who are developing or porting applications to AIX. This article is a compilation of questions that are frequently asked by vendors. The name of the responding Technical Support Group staff member appears after each response.

How can I pass a macro to a Fortran program?

You can pass a macro to a Fortran program by calling the C preprocessor, which in turn can pass this information to a Fortran program for compilation. The code example below contains the flags needed to preprocess macros. When `xlf` is invoked on a Fortran file with an `.F` extension, the preprocessor creates a preprocessed file that is prefixed with an `F` and the file extension is changed to a lowercase f (for example, `<filename>.F` becomes `F<filename>.f`).

The syntax of the compile is as follows:

```
xlf -d -WF,-D_AIX <filename>.F
```

where

`-d` leaves preprocessed source files produced by `cpp`, instead of deleting them. The `-WF` passes the listed options to a component that is executed during compilation. The component is `p`, `F`, `c`, `d`, `I`, `a`, or `I` corresponding to an optimizing preprocessor, the C preprocessor, the compiler, the `-S` disassembler, the Interprocedural Analysis (IPA) tool, the assembler, and the linker, respectively.

In the string following the `-W` option, use a comma as the separator and do not include any spaces. The `-D<name>` is equivalent to the `#define name`.

Note: Anything following `-WF` will be passed automatically to the `cpp`.

The following is the original Fortran program (that is, `<filename>.F`):

```
#ifdef _AIX
write (*,*) "_AIX is defined"
#endif
```

The preprocessed Fortran program (that is, `F<filename>.f`) is the following:

```
write (*,*) "_AIX is defined"
END
```

The output of the above program follows:

```
_AIX is defined
```

—Jeff Simon

Can an application written for parallel processing be run on a PowerPC?

Jeff Simon

Parallel programming requires the presence of certain Application Programming Interfaces (APIs) to enable clustering SP support. A Parallel Operating Environment (POE) for managing the development and execution of parallel applications is required to run any application that executes parallel programs on the AIX platform. A PowerPC

that is not configured to handle parallel processing will not be able to handle an application written for parallel processing. A parallel processing environment contains one or more of the following:

- IBM's Parallel Environment (PE)
- IBM's Message Passing Library (MPL)
- Message Passing Interface (MPI)
- IBM's Parallel Virtual Machine (PVM)

The flexibility of the SP system allows customers to run both parallel and non-parallel applications, but systems developed for parallel processing require that the environment contain the proper APIs and libraries.

—Jeff Simon

How can I find the hardware Ethernet address of an Ethernet adapter?

You can query your system for this information as follows:

```
lscfg -l <ethernet_device> -v | grep
Address
```

that is,

```
lscfg -l ent0 -v | grep Address
```

—Jeff Simon

Where can I find hints and tips about AIX/6000 Service?

AIX Service continues to add AIX/6000® hints and tips to the automated faxes available from the IBM Fax Information Service. This service is available free of charge to customers and field personnel 24 hours each day, seven days each week by calling the numbers listed below and following the instructions.

- Within the U.S., call 1-800-IBM-4FAX from a touch-tone phone.
- Outside the U.S., call 415-855-4FAX from your fax machine.

Figure 1 shows a few of the many faxes about AIX that are available from IBM Fax.

Wade Carlin

IBM Fax Information Service

Fax #	Description
1829	About AIX Service Hints and Tips from 4FAX
1293	Backup/Install—AIX 4.1 Installation Tips
6962	AIX Version 4.1 Quick Installation Guide
6964	AIX Version 4.2 Quick Installation Guide
3685	AIX 4.2 Installation Tips

Figure 1. AIX faxes

—Wade Carlin

What does mirroring the root volume really mean on AIX?

A simple mirroring allows users to create a copy of the default logical volumes if a disk experiences a failure during runtime.

The default root volume groups in AIX are as follows: hd1, hd2, hd3, hd4, hd5, hd6, hd8, and hd9var. All default root volume groups must be resident on the same disk. Their equivalent mirrors—those which are being mirrored—must reside on another disk. The logical volumes and the mirrored copies cannot span multiple disks.

Figure 2 shows the steps to take to extend the default root volume group from hdisk0 to hdisk1 while running as root.

```
extendvg rootvg hdisk1    (This extends root volume group to hdisk1)

chvg -Qn rootvg           (This disables QUORUM)

mklvcopy hd1 2 hdisk1     (Mirrors   /home file system)
mklvcopy hd2 2 hdisk1     (Mirrors   /usr file system)
mklvcopy hd3 2 hdisk1     (Mirrors   /tmp file system)
mklvcopy hd4 2 hdisk1     (Mirrors   / (root) file system)
mklvcopy hd5 2 hdisk1     (Mirrors   blv, boot logical volume)
mklvcopy hd6 2 hdisk1     (Mirrors   paging space)
mklvcopy hd8 2 hdisk1     (Mirrors   file system log)
mklvcopy hd9var 2 hdisk1  (/var file system)

syncvg -v rootvg

bosboot -a

bootlist -m normal hdisk0 hdisk1
```

Figure 2. Extending the default root volume group

Then you must shut down your system for the mirroring to take effect.

—*Wade Carlin*

How can I query the port status?

If you are writing a client/server application and errors occur when the client attempts to access the server port, you should isolate whether the application is failing and verify that the AIX (server) port is functioning properly.

There are several ways to monitor the accessibility and status of your port. Monitoring the port status can be accomplished at a high level by using the following command:

```
netstat -a | grep 50 | grep LISTEN
```

This checks to see if the port is in the "listen" mode (you can also check for other things. The `rpcinfo -p` command will report the status of the server. For example, this command reports whether the server is ready and waiting, or not available (see InfoExplorer™ for further details).

On a lower level coding approach, AIX uses the `ioctl` subroutines `getsockopt` and `setsockopt`, both documented in InfoExplorer. With these routines you can query information about existing sockets or assign new information to unused sockets.

Another lower level approach is to look at the `/usr/include/sys/socket.h`. Here you will note that the `sa_family` will contain the client address that has connected to the port, shown in Figure 3. Prior to connecting the port, the port is in the listen mode.

The process is as follows: After a connection-oriented server executes a `listen` system call in its server or test code, an

```
/*
* Structure used by kernel to store most
* addresses.
*/
struct sockaddr {
 u_char sa_len;      /* total length */
 u_char sa_family;   /* address family */
 char sa_data[14];   /* actually longer; address value */
};
```

Figure 3. Lower level coding approach

```
#include
#include
int accept (int sockfd, struct sockaddr *peer, int *addrlen)
```

Figure 4. Accept system call

actual connection from some client process is waited for by having the server execute the `accept` system call.

The `accept` system call takes the first connection request on the queue and creates another socket with the same properties as `sockfd`. The peer and `addrlen` arg's (see Figure 4) return the address of the connected peer process (the client). The new socket descriptor returned by `accept` refers to a complete association:

```
{protocol, local-addr, local-process,
foreign-addr, foreign-process}
```

where the foreign address (the client who made the connection) has been established. This allows the server to identify that a connection has been made and to log the data appropriately.

The AIX port layout may also be useful. On the Internet, 1 to 1023 are reserve ports and 1024 to 5000 are ports automatically assigned by the system. To request a port number, use one between 512 and 1023 allocated by `rresvport (int *aport)`. See Figure 5 for the port layout.

AIX Ports

Range	Usage
1-255	Well-known, privileged Internet services
256-511	Considered reserved, but not currently used by any standard Internet applications and not allocated by `rreservport()`
512-1023	Ports allocated by `rreservport()` (privileged)
1024-5000	Ports automatically assigned by system
5001-16383	Reserved for servers (not necessarily privileged)
16384-65535	Free for application use

Figure 5. AIX port layout

—David McCloud

Does AIX have a NFS Version 3 offering?

The Network File System (NFS) has been updated in AIX 4.2.1 to include support for the latest NFS protocol update—NFS Version 3. AIX NFS continues to provide distributed file system access for AIX as in the past. The AIX 4.2.1 implementation provides an NFS 2.0 client and server; therefore, it is backward compatible with the existing installed base of NFS clients and servers. Some NFS features include the following:

- The NFS client can request an asynchronous write and commit sequence for writing file data. This feature enables faster file writes to the NFS server. With the current NFS 2.0 protocol, the NFS server must write file data to disk before responding to the NFS client. If the NFS 3.0 asynchronous write request is used, this is not required.
- NFS 2.0 limited the size of `READ` and `WRITE` requests to 8 KB. The NFS 3.0 protocol relaxes the transfer size for `READ`s and `WRITE`s. The AIX implementation, like most in the industry, offers a 32 KB `READ` and `WRITE` size for both client and server. For example, with NFS 2.0 the reading of a 128 KB file would require the NFS client to send 16 individual remote procedure calls to the NFS server. With NFS 3.0, the same file could be read with four remote procedure calls.

- The NFS 3.0 protocol can now access files greater than 2 GB in size. The AIX NFS client and server, therefore, provides access to files greater than 2 GB.

Together with NFS 3.0 support, AIX 4.2.1 supports other functional changes in the NFS client and server.

NFS over TCP. The NFS client and server can use the TCP network transport for communication. Prior to AIX 4.2.1, NFS was limited to the User Datagram Protocol (UDP) transport for remote procedure calls. The default transport for AIX 4.2.1 NFS is UDP. For those environments that could benefit from TCP transport usage, the transport can be selected when the file system is mounted on the NFS client. Some examples of environments that may benefit from NFS over TCP usage would be networks with several intermediate gateways or routers, wide area networks, or environments with heavily loaded NFS servers. In all cases, NFS over TCP should provide a well-balanced network load.

Multithreaded NFS server. The NFS client and server daemons were implemented in AIX 4.2.1 using AIX's multithreading support. The NFS server daemon, `nfsd`, has been a multiprocess implementation in the past. With the new multithreaded NFS server, load balancing the server becomes much easier. NFS server threads are created and destroyed on demand as the incoming NFS client requests increase and decrease. The NFS client also takes advantage of the multithreaded approach to provide a well-balanced resource approach to reading and writing files.

Improved NFS file locking implementation. In the previous AIX releases, the NFS daemon requested by services network file locking has been a separate, user-level process. In AIX 4.2.1, the NFS file locking requests are serviced very similar to the classic NFS requests. The `rpc.lockd` daemon, a multithreaded kernel-level implementation, allows better throughput and response time. The AIX 4.2.1 NFS implementation, in certain circumstances, will provide better NFS performance than previous releases. Improvements may be observed in those environments where throughput is not limited because of CPU, network bandwidth, or disk bandwidth.

- The NFS 3.0 implementation can provide better throughput for `READ` because of the larger transfer sizes. Up to 20% more throughput can be obtained using a 32 KB `READ` size over NFS 3.0 compared to the 8 KB `READ` size over NFS 2.0.
- Using the asynchronous write capability of NFS 3.0, along with the larger `WRITE` transfer sizes, can result in an increase of two to three times the current sequential write throughput measured with NFS 2.0.
- The NFS 2.0 server implementation can provide up to 10% improvement for overall throughput or NFS operations per second.

—David McCloud

PS Command Values

Command	Description
CPU	The sum of the CPU usage by each of the threads
PRI	The highest priority of all the threads; for example, if a process has three threads with priorities 20, 30, and 40, the highest priority, 20, is displayed
WCHQN	1. Blank: None of the threads are waiting on a channel or none of the threads are waiting for an event 2. An address: (for example, 19e174c). Only one thread is waiting on a channel 3. An asterisk (*): More than one thread is waiting on a channel

Figure 6. The ps command values

How should the C, PRI, WCHAN on a ps command be interpreted for a multithreaded process?

For a command `ps -elUx | pg`, the values for `C` (CPU usage), `PRI` (priority), and `WCHAN` (wait channel) for a multithreaded process can be interpreted as shown in Figure 6.

—Asma Saudagar

*Compiled by **Jeff Simon**, IBM Corporation, 11400 Burnet Road, Austin, TX 78758.*

Multithreaded Programming in XL Fortran Version 5

By Julian Wang

To facilitate explicit parallelization of Fortran programs, XL Fortran Version 5 provides a Fortran 90 module to interface to the `pthread` *library. The module provides procedures that handle the details of Fortran-C interlanguage calls, making it easy for Fortran codes to exploit and use threads.*

Parallel programming is essential to exploit the power of multiprocessor systems. XL Fortran Version 5 supports two fundamental approaches for parallel programming. First, the programmer can insert appropriate directives into the source code to guide the compiler in parallelizing nested loops. Second, the programmer can use the thread library to explicitly parallelize the computations. The first approach corresponds to data parallel programming, while the second approach corresponds to task parallel programming models.

The choice between the two approaches largely depends on the nature of the application, more specifically, the computation model used in the underlying algorithm. It may be beneficial to employ task-level parallelism to map large chunks of the application and use data parallelism to map iterations of nested loops within the chunks. Thus, the two approaches are not mutually exclusive, and in fact, XL Fortran V5 allows the programmer to mix the two approaches in the same application.

The XL Fortran V5 compiler is effective in parallelizing applications by automatically detecting data parallelism. However, it may be necessary to explicitly parallelize programs using task parallelism for improved performance. Although it is tedious to inform the compiler about task-level parallelism, often it is the only approach to exploit parallelism in certain applications.

The facilities in the XL Fortran V5 compiler for explicitly specifying parallelization provide the programmer finer control over overlapping operations so as to obtain the best performance possible. The `pthread` library on AIX provides necessary and portable facilities to parallelize a program. However, for native Fortran 90 programmers, a Fortran 90 interface to the `pthread` library is highly desirable for ease of programming. This article summarizes the `pthread` library facilities in general, and introduces a Fortran 90 interface module, `f_pthread`, provided with the XL Fortran V5 compiler.

AIX pthread Library

The `pthread` library on AIX is based on the emerging POSIX™ 1003.1c industry standard. It provides an object-oriented interface to the application. The programmer manipulates opaque objects through object-type related operations. The library defines both the object types and the allowed operations. The `pthread` library has three pairs of object types:

- Threads and thread attributes objects
- Mutexes and mutex attributes objects
- Condition variables and condition attributes objects

Julian Wang

Creating an object requires the creation of an attributes object. The attributes object specifies certain aspects of the characteristics of the created object. See *AIX Version 4.1 General Programming Concepts: Writing and Debugging Programs* for more information on the characteristics that can be specified by the programmer for each type of object.

Typically, an attributes object is created with attributes having default values. Individual attributes in the attributes object can then be modified using operations provided by the library. Once the objects are created, they are not affected by destroying or modifying the attributes object used to create them.

Thread Creation and Termination

A *thread* is an independent execution sequence in a process. The `pthread` interface allows more than one thread to be active at a time. Executing multiple threads concurrently can possibly reduce the time to complete an application on a multiprocessor system. Multiple threads in a process share the system resources possessed by the process, including the memory.

A thread is created by a call to `pthread_create`, where one of the arguments is the routine that becomes the entry point for the created thread. The thread is terminated automatically when the thread returns from this routine. In addition, a thread can explicitly terminate itself by calling the `pthread_exit` procedure. This allows resources associated with the thread to be available in a timely manner.

A thread can also be terminated by any other thread in the process. Because all threads share the same data space, it is important to perform cleanup operations at the termination time. For this purpose, the `pthread` library provides a procedure to register cleanup handlers for a thread.

It should be noted that the initial thread is created by the system when the process starts. If this initial thread terminates by returning from the main program, then the process itself is terminated, including all its threads. If the initial thread terminates by calling `pthread_exit`, it will not affect other threads in the process.

Thread Synchronization

When multiple threads are active simultaneously, the ability to synchronize their activities is necessary. However, data inconsistencies may result in these interactions because of race conditions. A race condition occurs when two or more threads need to perform operations on the same set of data items, but the results of computations depend on the order in which these operations are performed.

Another form of interaction among threads is through explicit communication about the events that occur. This requires threads to be able to wait for other threads to complete an activity, including a thread's termination. When the condition occurs, then the waiting threads should be informed of the condition.

The `pthread` library provides a primitive synchronization mechanism for each of the purposes above, mutexes and condition variables, respectively. There are procedures to create, manipulate, and destroy mutex or condition variable objects. Mutex objects can be created by calling `pthread_mutex_init`. The mutex objects can then be locked by calling `pthread_mutex_lock` or `pthread_mutex_trylock` to prevent data inconsistencies from happening. The mutex locks can be released by calling `pthread_mutex_unlock`. Finally, mutexes can be deleted by the procedure `pthread_mutex_destroy`.

Threads can call `pthread_cond_wait` or `pthread_cond_timedwait` to wait on a condition variable after it is created by `pthread_cond_init`. When the condition is satisfied, a thread can use `pthread_cond_signal` or `pthread_cond_broadcast` to inform the waiting thread(s) of the event. After using a condition variable, its associated resources can be reclaimed by calling `pthread_cond_destroy`.

Thread Scheduling

Threads require resources such as processor time, memory, and file descriptors in order to execute properly. The manner in which the threads are scheduled to use those

resources can have a profound effect on the performance of programs. The pthread library provides several facilities to handle and control the scheduling of threads, including:

- Setting scheduling attributes when creating a thread
- Dynamically changing the scheduling attributes of a created thread
- Defining the effect of a mutex on the thread's scheduling when creating a mutex
- Dynamically changing the scheduling of a thread during synchronization operations

The last two types of controls are known as *synchronization scheduling*. Synchronization scheduling is not yet available on AIX, so it will not be discussed further.

Three scheduling parameters are associated with each thread: contention scope, scheduling policy, and scheduling priority. Usually, the library provides default values for these parameters, which are sufficient for most cases. However, the scheduling parameters can also be set either by using the thread attributes object before the thread's creation, or by dynamically changing attribute values during the thread's execution. The procedure pthread_attr_getschedparam can be called to check the current setting of scheduling attributes in a thread attributes object; whereas pthread_attr_setschedparam can be used to change those attributes' values. After a thread is started, pthread_getschedparam can be used to retrieve the current scheduling attributes of the thread, and pthread_setschedparam can be called to change them.

Advanced Features

The thread library provides some advanced features to be used by more experienced programmers. These include the one-time initialization facility, thread-specific data support, and thread stack management.

Some program libraries are designed for dynamic initialization; that is, the library will be initialized the first time a procedure in the library is called rather than requiring the caller to explicitly call an initialization procedure before any others. In multithreaded programs, however, this requires extra care since more than one thread can simultaneously call into the library. If not protected properly, the library may be initialized more than once. To make this task easier, the thread library provides a procedure called pthread_once. It will do nothing if other threads have called it already. This will make porting such a library to a multithreaded environment straightforward.

Threads require resources such as processor time, memory, and file descriptors in order to execute properly.

Many applications require that certain data be maintained on a per-thread basis across procedure calls. The thread-specific data interface is provided to meet these demands. Essentially, thread-specific data can be viewed as a two-dimensional array, with keys serving as the row index and thread IDs as the column index. Before manipulating thread-specific data, threads must create (or know) the thread-specific data keys designating the data by calling pthread_key_create. It should be noted that the keys are common to all threads in a process, only the data associated with them are private to each thread. Thread-specific data can be manipulated by the pthread_setspecific and pthread_getspecific procedures, with a key as one of the arguments.

Thread stacks are usually automatically and transparently managed by the system. In general, these stacks are sufficient for most applications. However, they may be too restrictive for certain applications. Using advanced thread attributes, it is possible for the user to control the size of the stack (for example, pthread_attr_setstacksize).

The f_pthread Fortran 90 Module

As it is, the pthread library on AIX is easy to use with the C programming language (or its derivatives). But it is not that straightforward to use within Fortran applications. At least three obvious issues need to be addressed to make it friendly for Fortran programmers:

- Provide derived data types for each object type mentioned above
- Observe possible differences in argument-passing conventions between Fortran and C
- Support native Fortran 90 data types and data structures (for example, the CHARACTER type and the array section construct)

A Fortran 90 module can be used to help resolve these issues. A module provides a means for packaging data types and procedures that operate on those types. This is why the Fortran 90 module f_pthread is provided in the XL Fortran Version 5.1 product. Before discussing the details of how f_pthread addresses the three issues, some general features of the module should be noted.

There is a one-to-one correspondence between the entities provided by the f_pthread module and those in the pthread library. Entities are named by prefixing f_ to the corresponding pthread name (except for constants). For example, there is a Fortran 90-derived type f_pthread_attr_t to be used in Fortran programs as pthread_attr_t is used in C programs. f_pthread_create can be called to create a thread using the module instead of pthread_create from the library. The module procedure will return whatever value is returned from the library routine. The error code constants are also provided in this module. EINVAL from this module has the same value as is defined in the system header file. These symbolic names will make Fortran programs more portable.

Data Types

The object-oriented interface provided by the pthread library is intended to make programs that use this interface portable across different platforms. It can also shield the application programmers from the exact definitions of the object types on each platform. As a matter of fact, the definitions of the object types are highly likely to be different on different systems, and may change from release to release on a particular one. This poses an obstacle to native Fortran programmers trying to do multithreaded programming without basic support, such as an f_pthread module, from the language product. The C struct definitions must be mapped very carefully into Fortran 90-derived types, observing padding and alignment rules. This is non-portable and tedious, to say the least.

With the f_pthread module, it is easy to use pthread object types in Fortran, and the portability across AIX® levels is guaranteed by the product. The example in Figure 1 shows how to declare a thread attributes object, a thread object, and a mutex object.

```
use f_pthread
type(f_pthread_attr_t)  obj_attr
type(f_pthread_t)       obj_thread
type(f_pthread_mutex_t) obj_mutex
```

Figure 1. Declaring thread attributes, thread, and mutex objects

Argument Passing

The differences of argument-passing conventions among programming languages makes interlanguage programming error-prone. It is possible to directly program in Fortran to the pthread library interface, but it is important to observe the argument-passing mechanisms from Fortran to C and from C back to Fortran. For example, in XL Fortran 5.1, the default is call-by-reference under most situations, whereas the default in C is call-by-value.

Although there are built-in functions that can be called to enforce calling convention consistency when language boundaries are crossed, programmers are

```
use f_pthread
type(f_pthread_attr_t)  obj_attr
integer old_state, new_state
integer any_err

! Initialize a thread attributes object
any_err = f_pthread_attr_init(obj_attr)

! Retrieve the default state
any_err = f_pthread_attr_getdetachstate(obj_attr, old_state)

! Set a new state: can be done either simply,
! any_err = f_pthread_attr_setdetachstate(obj_attr,  &
!       PTHREAD_CREATE_UNDETACHED)
! or,
new_state = PTHREAD_CREATE_UNDETACHED
any_err = f_pthread_attr_setdetachstate(obj_attr, new_state)
```

Figure 2. The detach state attribute in a thread attributes object

responsible for determining when they should be used. By using a module such as `f_pthread`, Fortran programmers are relieved of this problem since the module procedures provided worry about the interlanguage calling issues. Furthermore, the compiler is able to detect when incorrect arguments are used.

The code segment in Figure 2 shows how to manipulate the detach state attribute in a thread attributes object (error checking is not shown).

Native Fortran Constructs

Certain data types or data constructs are specific to Fortran and cannot be easily mapped to constructs in other languages. For example, the Fortran `CHARACTER` type has a length attribute, and Fortran 90 array sections have a shape attribute. The representation of these attributes is highly implementation dependent. It is a daunting job even for experienced programmers to figure out the exact details in a particular

```
use f_pthread
interface
    subroutine char_sub(charg)
        character(*) charg
    end subroutine
end interface
type(f_pthread_attr_t) attr
type(f_pthread_t) thread1
character(20), save:: c_arg
integer any_err

any_err = f_pthread_attr_init(attr)
any_err = f_pthread_create(thread1, attr, FLAG_CHARACTER, &
          char_sub, c_arg)
```

Figure 3. Subroutine requiring a CHARACTER argument

```
module global
    use f_pthread

    ! This program can be used to compute:
    !           c = a X b, where
    ! c is a matrix of dimension (x_size,z_size)
    ! a is a matrix of dimension (x_size,y_size)
    ! b is a matrix of dimension (y_size,z_size).
    ! The computation will be distributed among nthreads to be
    ! completed.
    integer nthreads, x_size, y_size, z_size
    parameter (nthreads=4, x_size=16, y_size=16, z_size=16)

    ! Set up a barrier for synchronization
    integer finish/0/
    type(f_pthread_mutex_t) ba_mutex/PTHREAD_MUTEX_INITIALIZER/
    type(f_pthread_cond_t)  ba_cond/PTHREAD_COND_INITIALIZER/
    type(f_pthread_t) threads(nthreads)

    ! Matrix mult_a and mult_c are arranged to take into account the
    ! fact that Fortran arrays are column-major. Spatial locality
    ! has substantial impact on the performance in this application.
    real(8) mult_a(y_size, x_size), mult_b(y_size, z_size)
    real(8) mult_c(z_size, x_size)
end module global

program example
    use global

    ! An explicit interface is required for an assumed-shape array
    ! argument.
    interface
        subroutine mult(rows)
            real(8) rows(:, :)
        end subroutine
    end interface
    integer i, ret

    ! Input mult_a and mult_b: not shown here.

    threads(1) = f_pthread_self()
    do i=2, nthreads
        ! We can do without a thread attributes object; the system
        ! default will be used. The workload distribution is cyclic.
        ret = f_pthread_create(threads(i), flag=FLAG_ASSUMED_SHAPE, &
                               ent=mult, &
                               arg=mult_a(1:y_size, i:x_size:nthreads))
    end do

    call mult(mult_a(1:y_size, 1:x_size:nthreads))

    ! Output mult_c: not shown here.
end program example

subroutine mult(rows)
    use global
    real(8) rows(:,:), part_r
    integer i, j, k, ret, which_thr, stride
    type(f_pthread_t) myself
```

Figure 4. Multithreaded Fortran 90 program (continued on following page)

```
    ! Find out who I am
    myself = f_pthread_self()
    do i = 1, nthreads
        if (f_pthread_equal(myself, threads(i))) then
            which_thr = i
        end if
    end do

    ! Do my part of work
    stride = 0
    do i = 1, ubound(rows, 2)
        do j = 1, z_size
            part_r = 0.0
            do k = 1, y_size
                part_r = part_r + rows(k, i) * mult_b(k, j)
            end do
            mult_c(j, which_thr+stride) = part_r
        end do
        stride = stride + nthreads
    end do

    ! Synchronize: main thread will return, but others will exit.
    ret = f_pthread_mutex_lock(ba_mutex)
    finish = finish + 1
    if (which_thr .eq. 1) then
        if (finish .ne. nthreads) then
            ret = f_pthread_cond_wait(ba_cond, ba_mutex)
        end if
    else
        if (finish .eq. nthreads) then
            ret = f_pthread_cond_signal(ba_cond)
        end if
    end if
    ret = f_pthread_mutex_unlock(ba_mutex)
end subroutine mult
```

Figure 4. Multithreaded Fortran 90 program (continued from previous page)

compiler. It is even more difficult to use them in calls to an interlanguage library such as the `pthread` library. Support is built into the `f_pthread` module, which allows a Fortran program to make use of these Fortran-specific features in calls to the `pthread` library.

Figure 3 shows how to create a thread, whose entry is a subroutine that requires a `CHARACTER` argument.

Application Examples

Figure 4 shows a multithreaded Fortran 90 program for matrix multiplication using the `f_pthread` module. The algorithm was chosen to show how native Fortran data constructs are supported and is not intended to be the best algorithm in terms of performance. For simplicity, all error checking is ignored. The comments in the source code in Figure 4 are helpful in understanding the algorithm.

***Julian Wang**, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Wang is a staff software developer, working on the Fortran compiler and runtime. He has a BS in Computer Science from the University of Science and Technology of China, an MS in Computer Engineering from Academia Sinica, and an MS in Computer Science from the University of Toronto.*