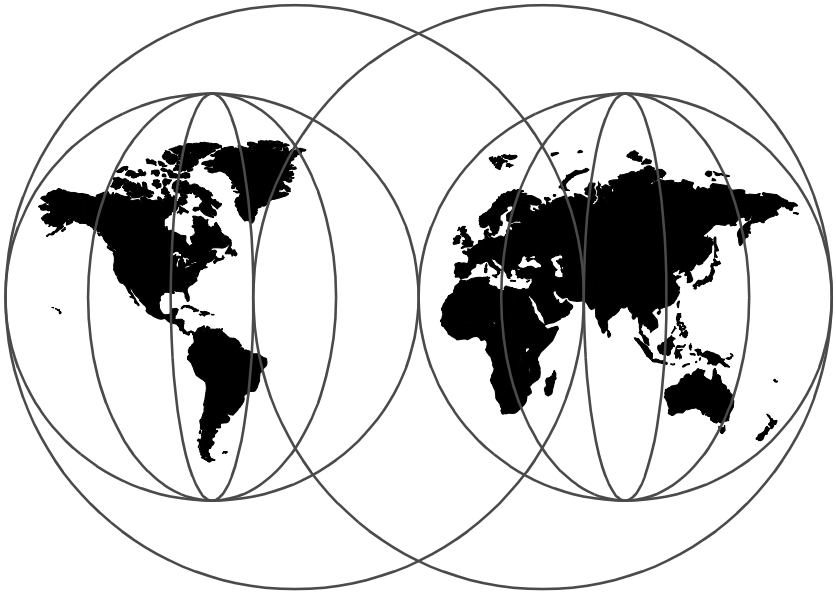
RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide

Stefan Andersson, Ron Bell, John Hague, Holger Holthoff
Peter Mayes, Jun Nakano, Danny Shieh, Jim Tuccillo



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-5155-00

SG24-5155-00

International Technical Support Organization

RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide

October 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix D, “Special Notices” on page 199.

First Edition (October 1998)

This edition applies to XL Fortran Version 5.1.1 (5765-C10 and 5765-C11) running under AIX Version 4.3 (5765-C34) on an RS/6000 43P 7043 Model 260 Workstation.

Note

This book is based on a pre-GA version of a product and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this redbook for more current information.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JN9B Building 045 Internal Zip 2834
11400 Burnet Road
Austin, Texas 78758-3493

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

This redbook provides information to help you understand and exploit the new generation of computer systems based on the RS/6000 POWER3 architecture. Specifically, this publication will address the following issues:

- POWER3 features and capabilities
- CPU and memory optimization techniques, especially for Fortran programming
- AIX XL Fortran Version 5.1.1 compiler capabilities and which options to use
- Parallel processing techniques and performance
- Available libraries and programming interfaces
- Performance examples on commonly used kernels and on several full applications

The anticipated audience for this redbook is as follows:

- Application developers
- End users who may be involved in making modifications to applications
- Technical managers responsible for equipment purchase decisions
- Managers responsible for project planning
- Researchers involved in numerical algorithm development
- End users with an interest in understanding the performance of their applications

While this publication is decidedly technical in nature, the fundamental concepts are presented from a user point of view and numerous examples are provided to reinforce these concepts. Furthermore, this publication is organized such that the information becomes more detailed as one progresses through the chapters. This organization will allow readers to stop, once they have achieved the level of understanding they desire, without having to search through the publication.

To some extent, this book should be regarded as a series of subtopics that can be read alone. Each chapter is relatively complete in itself, referring to other chapters where appropriate.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

Stefan Andersson is a Staff Engineer/Scientist at IBM Poughkeepsie. He has an MS in mathematics from the University of Heidelberg. He began his work with IBM at the IBM Scientific Center, Heidelberg in 1990. He has been involved in parallel computing on RS/6000 SP since 1992. Currently, he is a member of the technical benchmark team at IBM Poughkeepsie. His areas of expertise include performance tuning for the POWER architecture, distributed memory coding and tuning for the RS/6000 SP, and shared memory coding and tuning on IBM SMPs.

Ron Bell is an IBM IT Consultant in the UK. He has an MA in Physics and a DPhil in Nuclear Physics from the University of Oxford. He has 27 years of experience with IBM High Performance Computing. His areas of expertise include the Fortran language, performance tuning for POWER architecture, and MPI parallel coding and tuning for the RS/6000 SP. He has for many years collaborated with HKS Inc. to optimize their ABAQUS product for IBM platforms.

John Hague is an IBM IT Consultant in the UK. He obtained a PhD in High Energy Physics at University College, London, and worked in this field at the Rutherford Lab in the UK and the Lawrence Livermore Lab in Berkeley until he joined IBM in 1970. John was assigned to the IBM ITSO in Poughkeepsie in 1985 to provide worldwide technical support for the IBM Vector Facility. Since then, he has worked exclusively in the scientific and technical area, and has considerable expertise in vectorizing, parallelizing, and tuning scientific programs, particularly in the Petroleum and Weather Forecasting areas.

Holger Holthoff is an IBM IT Consultant in Germany. He has been involved in parallel computing on RS/6000 SP since he joined the IBM Scientific Center, Heidelberg in 1994. Currently, he is a member of the RS/6000 Technical Support focusing on high-performance computing projects and CAE applications in manufacturing industries. He obtained the Dipl.-Ing. and Dr.-Ing. degree in mechanical engineering from University of Karlsruhe and Braunschweig, respectively. His areas of expertise include performance tuning for the POWER architecture and message passing programming for the RS/6000 SP.

Peter Mayes is a Senior IT Specialist in the UK. He has 15 years of experience in the field of high-performance computing. He holds the degrees of MA in Mathematics, MSc in Mathematical Modeling and Numerical

Analysis, and DPhil in Engineering Mathematics, all from the University of Oxford. His areas of expertise include Fortran programming, particularly for high-performance and parallel computers, and administration of RS/6000 SPs.

Jun Nakano is an IT Specialist of IBM Japan. From 1990 to 1994, he was with IBM Tokyo Research Laboratory and studied combinatorial optimization. Since 1995, he has been involved in RS/6000 and SP benchmarks. He holds MSc in physics from University of Tokyo. He is interested in algorithms, computer networks, and operating systems.

Danny Shieh is a Senior Engineer/Scientist of IBM Austin. He joined the IBM Palo Alto Scientific Center in 1969. From 1974 to 1976, he was assigned to the Large Scale Computing department in IBM San Jose Research. From 1985 to 1986, he was assigned to the IBM International Technical Support Center in Poughkeepsie, NY to support the IBM 3090 Vector Facility. He joined the IBM RS/6000 team in 1992. His current assignment is technical support of S&TC marketing for RS/6000 products. He received the MS and PhD degrees in Atmospheric Sciences in 1967 and 1969, respectively, from New York University.

Jim Tuccillo is an atmospheric scientist by training. He has attended Cornell University, Old Dominion University, and Johns Hopkins University. Jim has been involved in the development of Numerical Weather Prediction (NWP) Models on high-performance vector, parallel vector, and distributed memory systems since 1980. Jim has worked in the NWP development labs of the US Weather Service and NASA where he has been involved in the development of research and operational NWP codes for weather forecasting in the US. Jim currently works for IBM's Global Government Industry organization where he is involved in issues associated with NWP and high-performance computing on IBM's SP system. Jim has research interests in the areas of parallel algorithms and parallel programming paradigms for high-performance, numerically intensive computing.

This project was coordinated by:

Scott Vetter	IBM Austin

Thanks to the following people for their invaluable contributions to this project:

Alan Adamson	IBM Toronto
Yukiya Aoyama	IBM Japan
Arthur Ban	IBM Austin
Howard Brauer	IBM Austin

Luke Browning	IBM Austin
Frank Johnston	IBM Poughkeepsie
Matthias Laux	IBM Heidelberg
Lisa Martin	IBM Toronto
Joan McComb	IBM Poughkeepsie
Frank O'Connell	IBM Austin
Mark Papermaster	IBM Austin
Farid Parpia	IBM Poughkeepsie
Jim Shearer	IBM Watson Research
David Tuttle	IBM Austin
Steve White	IBM Austin

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in "ITSO Redbook Evaluation" on page 221 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

  For Internet users `http://www.redbooks.ibm.com`
  For IBM intranet users `http://w3.itso.ibm.com`

- Send us a note at the following address:

  `redbook@us.ibm.com`

Chapter 1. Introduction

This publication is designed to familiarize you with the IBM RS/6000 POWER3 architecture and provide you with the information necessary to exploit the new high-end technical workstations based on this architecture.

The two-way symmetric multiprocessing (SMP) workstation RS/6000 43P 7043 Model 260 will be the first POWER3 system to be available. Thus, most analysis presented in this publication refers to this system.

1.1 RS/6000 Processor Evolution

In this section, the stages of processor development are discussed, starting with the POWER1 architecture through to the latest POWER3. Various references for additional reading are included.

1.1.1 POWER1

The first RS/6000 products were announced by IBM in February of 1990, and were based on a multiple chip implementation of the POWER architecture, described in *IBM RISC System/6000 Technology*, SA23-2619. This technology is now commonly referred to as POWER1, in the light of more recent developments. The models introduced included an 8 KB instruction cache (I-cache) and either a 32 KB or 64 KB data cache (D-cache). They had a single floating-point unit capable of issuing one compound floating-point multiply-add (FMA) operation each cycle, with a latency of only two cycles. Therefore, the peak MFLOPS rate was equal to twice the MHz rate. For example, the Model 530 was a desk-side workstation operating at 25 MHz, with a peak performance of 50 MFLOPS. Commonly occurring numerical kernels were able to achieve performance levels very close to this theoretical peak.

In January of 1992, the Model 220 was announced, based on a single chip implementation of the POWER architecture, usually referred to as RISC Single Chip (RSC). It was designed as a low-cost, entry-level desktop workstation, and contained a single 8 KB combined instruction and data cache.

The last POWER1 machine, announced in September of 1993, was the rack-mounted Model 990. It ran at 71.5 MHz and had a 32 KB I-cache and a 256 KB D-cache.

1.1.2 POWER2

Announced in September 1993, the Model 590 was the first RS/6000 based on the POWER2 architecture, described in *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, SA23-2737. The most significant improvement introduced with the POWER2 architecture for scientific and technical applications is that the floating-point unit (FPU) contains two 64-bit execution units, so that two floating-point multiply-add instructions may be executed each cycle. A second fixed-point execution unit is also provided. In addition, several new hardware instructions were introduced with POWER2:

- Quad-word storage instructions. The quad-word load instruction moves two adjacent double-precision values into two adjacent floating-point registers.
- Hardware square root instruction.
- Floating-point to integer conversion instructions.

Although the Model 590 ran with only a marginally faster clock than the POWER1-based Model 580, the architectural improvements listed above, combined with a larger 256KB D-cache size, enabled it to achieve far greater levels of performance, as shown in Table 1.

Table 1. Performance of POWER1 versus POWER2

	Model 580	**Model 590**
Architecture	POWER1	POWER2
MHz	62.5	66
D-cache	64KB	256KB
Peak MFLOPS	125	264
LINPACK DP MFLOPS	38	130
LINPACK % of peak	30%	49%
LINPACK TPP MFLOPS	104	237

In October 1996, IBM announced the RS/6000 Model 595. This was the first machine to be based on the P2SC (POWER2 Super Chip) processor. As its name suggests, this is a single chip implementation of the POWER2 architecture, enabling the clock speed to be increased further. The Model 595 runs at 135MHz, and the fastest P2SC processors, found in the Model 397 workstation and RS/6000 SP Thin4 nodes, run at 160 MHz, with a theoretical peak speed of 640 MFLOPS.

1.1.3 PowerPC

The RS/6000 Model 250 workstation, the first to be based on the PowerPC 601 processor running at 66 MHz, was introduced in September, 1993. The 601 was the first processor arising out of the partnership between IBM, Motorola, and Apple. The PowerPC architecture includes most of the POWER instructions. However, some instructions that were executed infrequently in practice were excluded from the architecture, and some new instructions and features were added, such as support for symmetric multiprocessor (SMP) systems. In fact, the 601 did not implement the full PowerPC instruction set, and was a *bridge* from POWER to the full PowerPC architecture implemented in more recent processors, such as the 603, 604, and 604e. Currently, the fastest PowerPC-based machines from IBM for technical purposes, the four-way SMP system RS/6000 7025 Model F50 and the uni-processor system RS/6000 43P 7043 Model 150, use the 604e processor running at 332 MHz and 375 MHz, respectively.

1.1.4 POWER3

The new POWER3 processor, described in detail in Chapter 2, “The POWER3 Processor” on page 7, essentially brings together the POWER2 architecture, as currently implemented in the P2SC processor, with the PowerPC architecture. It combines the excellent floating-point performance delivered by P2SC’s two floating-point execution units, while being a 64-bit, SMP-enabled processor ultimately capable of running at much higher clock speeds than current P2SC processors.

1.2 SMP-Based System Views

Since the POWER3 architecture provides SMP support, POWER3-based systems will feature multiple CPUs with a uniform access shared memory and shared I/O resources. This section outlines the different ways in which these multiple CPUs can be exploited, either by running multiple job streams to achieve greater overall system throughput, or by using a shared or distributed memory programming model to reduce the time to solve an individual problem.

1.2.1 Job Level Parallelism with Single CPU Jobs

For work loads consisting of many independent jobs each using a single CPU, the multiple CPUs of a POWER3 based system will provide greater throughput performance than a uni-processor system. For example, POWER3 based systems with two CPUs may provide twice the nominal performance on a work load when compared with a comparable

uni-processor system. Each POWER3 CPU will also provide an improvement in performance over existing CPUs.

1.2.2 Automatic Parallelization (Fortran)

The XL Fortran compiler (Version 5.1.1 or later) provides support for automatic parallelism of programs to provide increased performance so as to reduce the elapsed time of a program. Essentially, the code is analyzed for independent pieces of work that can be dispatched, in parallel, to the multiple CPUs of a POWER3 based system. This SMP capability is also available on machines using PowerPC processors, such as the Model F50. The ability of the compiler to detect opportunities for parallelism can vary and is dependent on the intrinsic properties of the problem being solved and the source code implementation. The nominal performance improvement over using a single CPU is generally limited to the number of CPUs on the POWER3 based system. Typically, new programs can be written in a manner that allows for a high-level of compiler-detected parallelism. Existing programs can often be modified to allow for significant levels of parallel efficiency. The automatic parallelization capabilities of XL Fortran can often be assisted through the insertion of compiler directives, as discussed in the next section.

1.2.3 Compiler Directives

Compiler directives are often used in conjunction with the automatic parallelization capability of the XL Fortran compiler to assist in situations where the dependency analyzer is unable to detect independent pieces of work. Compiler directives appear as Fortran comments so that code portability is preserved. OpenMP is an evolving industry standard that will provide for code portability across shared-memory parallel systems.

1.2.4 Message Passing Interface

The Message Passing Interface (MPI) is the industry standard for parallel programming on distributed memory systems, such as the IBM RS/6000 Scalable Parallel (SP) system. Programs that have been parallelized using the Message Passing Interface are highly portable between different platforms. In general, MPI programs also perform excellently on SMP systems. MPI is supported on clustered RS/6000 uni-processor machines as well as on SMP systems.

With this paradigm, the programmer has explicitly decomposed the problem to run as separate processes that communicate and synchronize through the MPI library. The separate processes of an MPI program are transparently mapped against the multiple CPUs of a POWER3 based system.

For IBM RS/6000 SP systems, an alternative approach exploiting SMP nodes is to assign a separate MPI process to each CPU of each node. With this approach, MPI message passing will take place at both the intra-node and inter-node level, and threads are not required to address the multiple CPUs of each node.

1.2.5 Using POSIX Threads

The *thread* programming interface is the native interface of parallel programming on SMP systems, but also used for performance improvements on uni-processor systems. On RS/6000, POSIX threads support is provided through both a C and Fortran application program interface (API) and allows for the exploitation of the multiple CPUs of a POWER3 based system. Since POSIX threads is an industry standard, programs written using this library are generally portable to other SMP platforms. At the time of publication, the Fortran binding for pthreads is not part of the POSIX pthreads standard, therefore, Fortran pthreads implementations may be AIX specific.

1.2.6 Combined MPI/Threads Paradigm

For IBM RS/6000 SP systems with SMP nodes, a combined MPI and threads programming paradigm is also supported. With this approach, a single MPI processes is assigned to each SMP node, and multiple threads are executed on each node. The threads will be used to execute the computational kernels so as to exploit the multiple CPUs on the node, and MPI communication will take place between the nodes. Threads can be either explicitly created through the POSIX Threads library or can be implicitly created with the automatic parallelism features of the XL Fortran compiler (with or without compiler directives), as discussed in 1.2.2, “Automatic Parallelization (Fortran)” on page 4.

Chapter 2. The POWER3 Processor

The POWER3 microprocessor introduces a new generation of 64-bit processors especially designed for high performance and visual computing applications. POWER3 processors are the replacement for the POWER2 and POWER2 Super Chips (P2SC) in high-end RS/6000 workstations and technical servers.

2.1 Processor Overview

The POWER3 implementation of the PowerPC architecture provides significant enhancements compared to the POWER2 architecture. The SMP-capable POWER3 design allows for concurrent operation of fixed-point instructions, load/store instructions, branch instructions, and floating-point instructions. The POWER3 is designed for ultimate frequencies of up to 600 MHz when fabricated with advanced semiconductor technologies such as copper metallurgy and silicon-on-insulator (SOI). In contrast, the P2SC design has reached its peak operating frequency at 160MHz. The first POWER3 based system, RS/6000 43P 7043 Model 260, runs at 200 MHz.

Capable of executing up to four floating-point operations per cycle (two multiply-add instructions), the POWER3 maintains the emphasis on floating-point performance and memory bandwidth that has become the hallmark of POWER2 based RS/6000 systems. Integer performance has been significantly enhanced over the P2SC with the addition of dedicated integer and load/store execution units, thus improving its SPECint95 performance relative to the 160 MHz P2SC by about 50 percent at 200 MHz. This gives the POWER3 far more balanced performance, which is especially notable in graphics intensive applications.

The POWER3 is a 64-bit PowerPC implementation with a 32-byte backside L2 cache interface (private L2 cache bus), and a 16-byte PowerPC 6XX bus, as shown in Figure 1. The POWER3 has a peak execution rate of eight instructions per cycle (compared to six for the P2SC) and a sustained performance of four instructions per cycle.

Significant investments in the chip’s data flow, instruction routing, and operand buffering have been made in order to sustain a high computational and corresponding data rate. The POWER3’s level-one (L1) data cache is an efficient interleaved cache capable of two loads, one store, and one cache line reload per cycle. Although half the size of the P2SC's cache, the L1 is effectively supplemented by a dedicated second level (L2) cache, which may be from 1 MB to 16 MB in size. Data and instruction prefetching mechanisms

improve the memory access performance by hiding memory latency. Also, the large 128 byte line size takes advantage of the locality of reference (spacial reuse) characteristic of large engineering and scientific data reference patterns

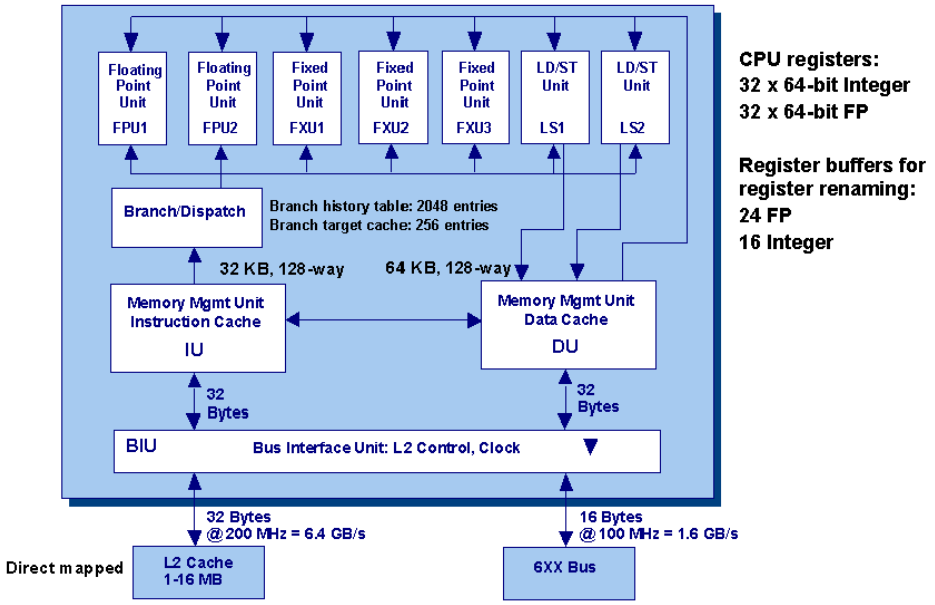


Figure 1. POWER3 Processing Units (Model 260)

2.2 POWER3 Execution Core

Unlike some competitive chips, which need several pipeline stages before instructions enter the first execution stage, POWER3 keeps this front end of the pipeline short, using only three stages. POWER3 needs only one cycle to access the instruction cache, one cycle to decode and dispatch the instructions to different execution units, and one more cycle to access the operands. POWER3's relatively short pipeline keeps its mispredicted branch penalty to only three cycles, up to 24 cycles shorter than its competitors.

Up to eight instructions (two floating-point, two load/store, two single-cycle integer, a multi-cycle integer, and a branch) can be in execution in each cycle. Ready instructions are issued out of order from the issue queues, allowing instructions of different types, as well as of the same type, to execute out of order. The load/store and branch instructions are issued in program order.

For branch instructions whose conditions are not known in the decode stage, POWER3 uses a 2,048-entry branch history table (BHT) to predict the branch direction. Because a branch is often resolved in the decode stage or soon thereafter, the benefit of the BHT when used to predict the current encounter of the branch is less in POWER3 than in designs with deeper pipelines. To better use the BHT, however, POWER3 uses the BHT to predict both the current and the next encounter of each conditional branch, using a branch target address cache (BTAC).

POWER3 uses rename registers for the general-purpose registers (GPR), floating-point registers (FPR), and the condition-code register (CCR) to allow out-of-order and speculative execution of most instructions. The few exceptions are stores and certain move-to-special-register instructions that are difficult to undo. Although instructions can be issued out-of-order, and thus, their operands can be read out-of-order from the registers, the rename registers eliminate anti- and output-dependencies by enabling the registers to be updated in program order.

POWER3 has two identical FPUs, each delivering up to two floating-point operations per cycle. POWER3's FPUs execute multiply-add instructions, as Table 2 shows, taking only one cycle throughput to calculate the frequently used (a*b+c) operation.

Table 2. POWER3's Low Execution Latencies

Instruction	Number of Cycles	
	32 bit	64 bit
Integer Multiply	3-4	3-9
Integer Divide	21	37
FP Multiply or Add	3-4	3-4
FP Multiply-Add	3-4	3-4
FP Divide	14-21	18-25
FP Square Root	14-23	22-31

The non-blocking caches support four outstanding L1 data demand requests and two outstanding L1 instruction demand requests in order to reduce the memory subsystem latency. The L1 cache also supports hits under misses, the L1 cache allows a fifth demand request which hits the cache to proceed even when there are four previous outstanding misses to the data cache. In

comparison, the POWER2 architecture allows only one outstanding cache miss without blocking. Cache hits are satisfied within a single cycle. The writeback data cache implements a four-state *MESI* cache coherence protocol (possible states: modified, exclusive, shared, and invalid) to support SMP environments.

POWER3 uses instruction- and data-prefetch mechanisms to reduce pipeline stalls due to cache misses. The instruction cache is two-way interleaved on cache-line boundaries, allowing one bank to be accessed for instruction fetches while the other bank is accessed for the next cache line. When the former access hits in the cache but the latter access does not, a prefetch request for this next cache line is issued to the L2 cache. Because the prefetch is still speculative, the request is not propagated to the main memory. If it misses in the L2 cache, this allows the request to be canceled upon detecting a mispredicted branch instruction. An instruction prefetch takes six cycles from the 200 MHz L2 cache.

For the data cache, the Model 260 can prefetch up to four *streams* of data from memory or L2 cache into L1 cache. To establish a prefetch stream, the prefetch mechanism monitors every access that misses in the data cache, searching for cache-miss references to two adjacent cache lines. For this purpose, a stream address filter queue of depth 10 is used, which contains the guessed next stream addresses. The filter is maintained by a least recently used (LRU) mechanism in order to age out seldom used prefetch streams. Upon finding such a pair of succeeding cache misses, it initiates a prefetch request for the next cache line. The stream addresses, along with the ascending or descending prefetch direction, is kept in a four-entry stream address buffer. Once a prefetch stream is identified, the address of every data-cache access is checked with the addresses in the stream address buffer. When a match is found, a prefetch request for the next cache line is made, and the address in the matching entry is updated with the address of the new prefetch request. A simplified view on the prefetch hardware is given in Figure 2.

When initially predicting the direction of a prefetch stream, it is assumed that if the word that causes the cache-miss occurs in the bottom half of the cache line, the next higher line will be required, but if the miss occurs in the top half, then the next lower line will be required. Then data is being prefetched in sequentially in either a forwards or backwards direction. If the initial prediction is wrong, the direction is corrected for the subsequent stream.

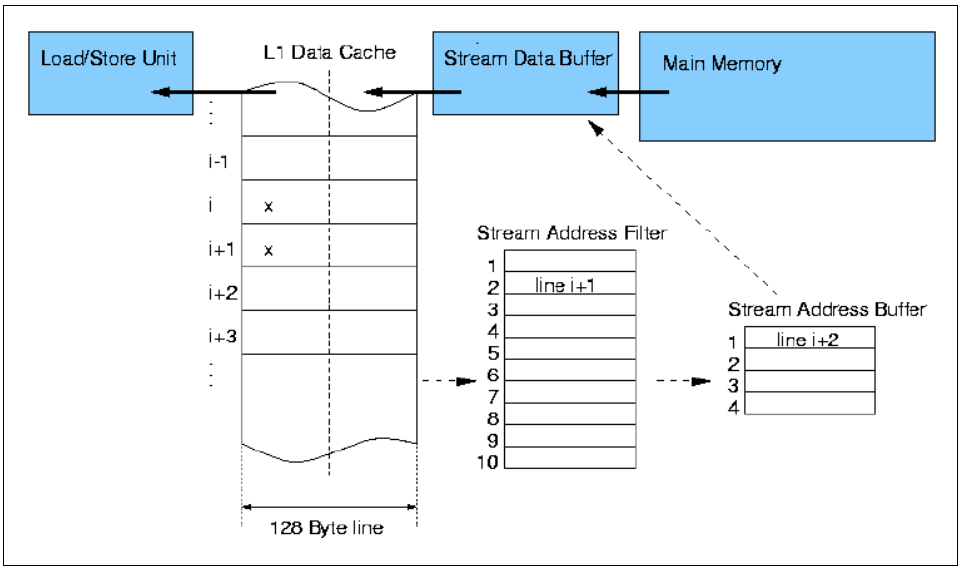


Figure 2. Data Prefetch Overview

The 64-bit address space is managed by using 80-bit virtual addresses and 40-bit real memory addresses, which support up to 1 terabyte. A 256-entry two-way set associative translation lookaside buffer (TLB) based on a least recently used replacement algorithm is used to access 4 KB memory pages.

The performance of many technical applications is mainly determined by the performance of the memory subsystem. POWER3 systems are designed to deliver industry leading memory bandwidth, which has already been a strength of the POWER2 architecture. The bandwidth, as listed in Table 3, in terms of GB/s depends on the actual clock frequency. As an example the DAXPY operation, y(i)=y(i)+a*x(i), yields a sustained memory bandwidth of 1.3 GB/s, close to the peak bandwidth of 1.6 GB/s of a POWER3 Model 260 system. DAXPY performance is analyzed in more detail in Chapter 7.3.3, “DAXPY” on page 98.

The load latency, due to either a data or instruction L1 miss that hits the L2 cache, amounts nine CPU cycles. A data access that misses the L1 and L2

cache causes a latency of about 35 cycles on a Model 260. However, this does not depend on the processor only, but also on the system.

Table 3. RS/6000 43P 7043 Model 260 Memory Bandwidth

Access	Interface Width [Bit]	Clock Frequency [MHz]	Bandwidth [Byte/cycle]	Bandwidth [GB/s]
Load Register from L1	128	200	2*8	3.2
Store Register to L1	64	200	8	1.6
Load/Store L1 from/to L2	256	200	4*8	6.4
Load/Store L1 from/to Memory	128	100	2*8	1.6

2.3 POWER3 Roadmap

The first generation of POWER3 based systems will operate at CPU speeds of 200 MHz and memory bus speeds of 100 MHz. The processor board will hold a direct mapped L2 cache of 4 MB per processor. The initial chip design does not support fractional processor-to-cache and processor-to-system clock ratios (such as 3:2 mode). But the second generation of POWER3 chips will remove this limitation. This will be the first design based on IBM's advanced CMOS-7S process. With help of this 0.2-micron process, which uses copper interconnects, clock speeds of more than 300 MHz will be achievable. The die size will shrink from 270 mm^2 to 160 mm^2, with a few additional functions.

IBM plans a second derivative of POWER3 chips in a 0.18-micron process, targeting speeds up to 500-600 MHz. This process may showcase IBM's unique Silicon-on-Insulator (SOI) technology. SOI protects the millions transistors on a chip with a *blanket* of insulation, reducing harmful electrical effects that consume energy and hinder performance. A floating-point and integer performance of SPECfp95 70+ and SPECint95 30+, respectively, is expected.

The faster POWER3 chips will support fractional bus modes (such as 5:2 and 7:2 for processor-to-bus and 3:2 for processor-to-cache interfaces) which will allow the core to run at its full speed. Using a set-prediction mechanism, the new chips will also support a four-way set-associative L2 cache.

Figure 3 on page 13 shows the high-level partition of logical units within the POWER3 chip.

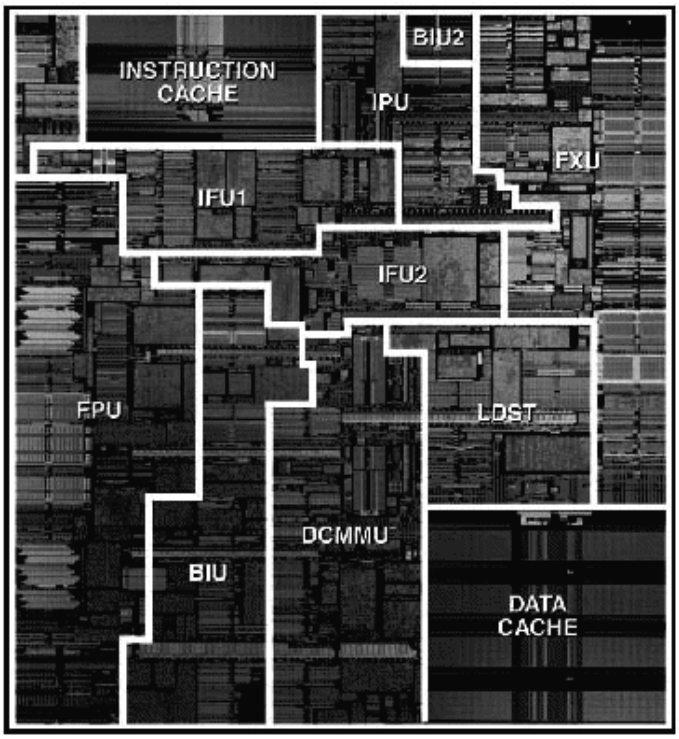


Figure 3. POWER3 Chip Layout: 270 mm^2 Die, 15 Million Transistors

2.4 POWER3-Based Systems

The POWER3 CPU will be featured in several different computer systems. There will be stand-alone workstations up through IBM RS/6000 SP nodes.

2.4.1 RS/6000 43P 7043 Model 260

The Model 260 is a desk-side RS/6000 system designed to perform as a high-performance technical workstation, visual client, or workgroup server. The mechanical package can accommodate up to two processor cards, two memory cards, and five PCI adapters. It also supports two hot-swap DASD bays (Ultra SCSI), two 5 1/4" media bays, and one floppy drive.

Each processor card carries one POWER3 chip running at 200 MHz.

The memory controller function is located on the planar. A system planar is shown in Figure 4 on page 14. The memory chipset supports a 128-bit data path to memory running at 100 MHz, giving the system a peak memory

bandwidth of 1.6 GB/s. The two processor cards have to share this bandwidth. The chipset is not only an interface to the memory but also to the 6XX-MZ mezzanine bus used for the I/O.

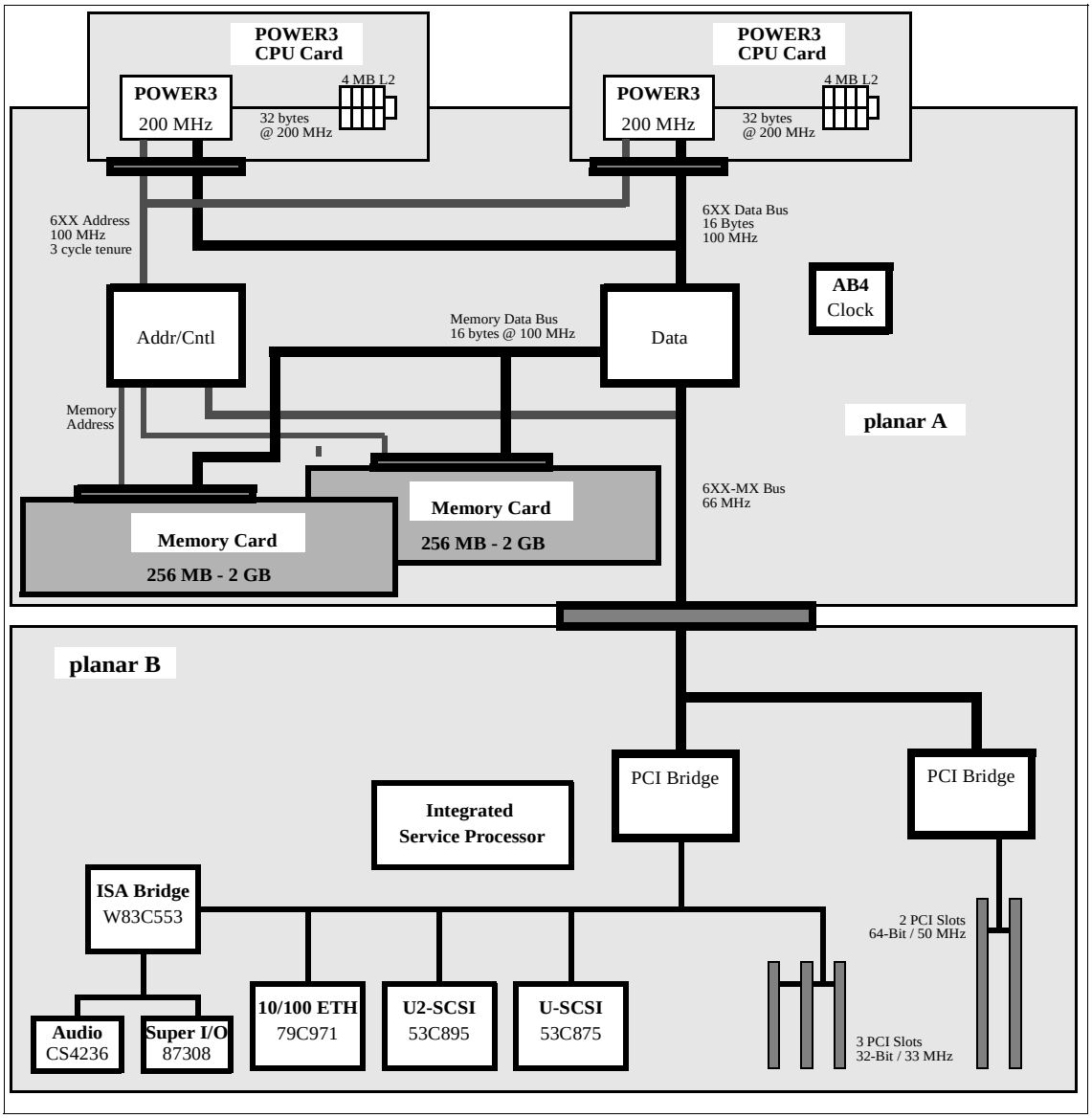


Figure 4. Logical View of the Model 260

Each memory card can carry between 256 MB and 2 GB of memory using 128 MB dual inline memory modules (DIMMs), giving the system a maximum total memory of 4GB. These sizes will double when 256 MB DIMMs become available.

2.4.2 IBM RS/6000 SP Nodes

In the future, there will be several nodes for the IBM RS/6000 SP available. The first one will be based on the RS/6000 43P 7043 Model 260. The differences between these node and the Model 260 are the form factor in order to fit into the IBM RS/6000 SP frame and the ability to connect it to the high performance switch. In order to fulfill the Accelerated Strategic Computing Initiative (ASCI) contract, IBM will also offer an eight- and later a 16-way SMP based on the POWER3 processor. These models are expected to contain several unique features and new design points.

2.4.3 DOE ASCI Project

On July 26, 1996, Lawrence Livermore National Laboratory (LLNL) announced it had selected IBM for an award of a $93 million contract to build the world's fastest supercomputer as part of the Department of Energy's (DOE) Accelerated Strategic Computing Initiative (ASCI) program, called ASCI Blue Pacific. The final configuration of the proposed system will consist of:

- 512 eight way POWER3 SMP nodes
- More than three teraflops peak performance
- 2500 GB total system memory
- 75 terabytes global disk capacity
- 6400 MB/s I/O bandwidth

In order to meet the increased need for computing power, the next step after ASCI Blue Pacific, called ASCI White, is already announced. The ASCI White System will consist of 8192 POWER3+ CPUs capable of peak speed of 10 trillion operations per second.

Both the ASCI Blue Pacific and the ASCI White project will drive the future RS/6000 and IBM RS/6000 SP system development in hardware as well as software. The result of this work will provide future gains through improved products for IBM Customers.

For more information about the ASCI project, visit the following Web pages:

```
http://www.doe.org
http://www.llnl.gov/asci/
```

Chapter 3. XL Fortran Version 5

XL Fortran Version 5 is the first XL Fortran compiler that has the ability to exploit SMP processors concurrently for improving the performance. It is also the first to produce object code that runs in 64-bit mode on AIX 4.3 or later. This chapter mainly describes differences between XL Fortran Version 5.1 and previous versions that users should be aware of for compiling and running programs on POWER3 hardware.

3.1 SMP Support

One of the most outstanding features of XL Fortran Version 5 is its support for SMP. The compiler automatically identifies DO loops that can be parallelized and makes object code that runs in a multi-threaded fashion. Or, you can give directives to the compiler in order to provide additional information on the code or to force the compiler to parallelize certain DO loops. Detailed explanations and examples will be given in Chapter 4, “Using the SMP Feature of XL Fortran” on page 29. An overview of compiler architecture is presented here. (See D. Kulkarni et al., “XL Fortran Compiler for IBM SMP Systems,” *AIXpert Magazine*, December 1997.)

Figure 5 on page 18 shows the path through the XL Fortran compiler when the parallelization facility is activated with the -qsmp option. The Fortran front end takes your program as input, checks the program syntactically and semantically, and produces an intermediate representation of it. The scalarizer transforms the Fortran 90 array language constructs into scalar DO loops.

The subsequent locality optimizer and serial and SMP optimizer perform optimizations, including loop reordering, array padding, loop tiling, loop unrolling, elimination of conditionals, and so on. If given the target architecture by the -qarch option, the compiler takes into account hardware specifics, such as cache size and cache line size. The parallelizer uses loop reordering transformations to automatically parallelize loops at outermost levels, which minimizes parallelization overheads, such as barrier synchronization at the end of parallel loops, and ensures larger computation granularity on each of the processors of the SMP system. The outliner does the converse of subroutine inlining. It converts DO loops, which are decided to be parallelized, into subroutines.

You can see how a program is outlined by reading the outlining report section of hotlist, which is generated by the -qreport=hotlist compiler option. An example of hotlist is given in 4.1, “How to Compile, Link, and Execute” on

page 29. By invocation of xlf_r or xlf90_r, the object code is linked with thread-safe libraries for parallel execution.

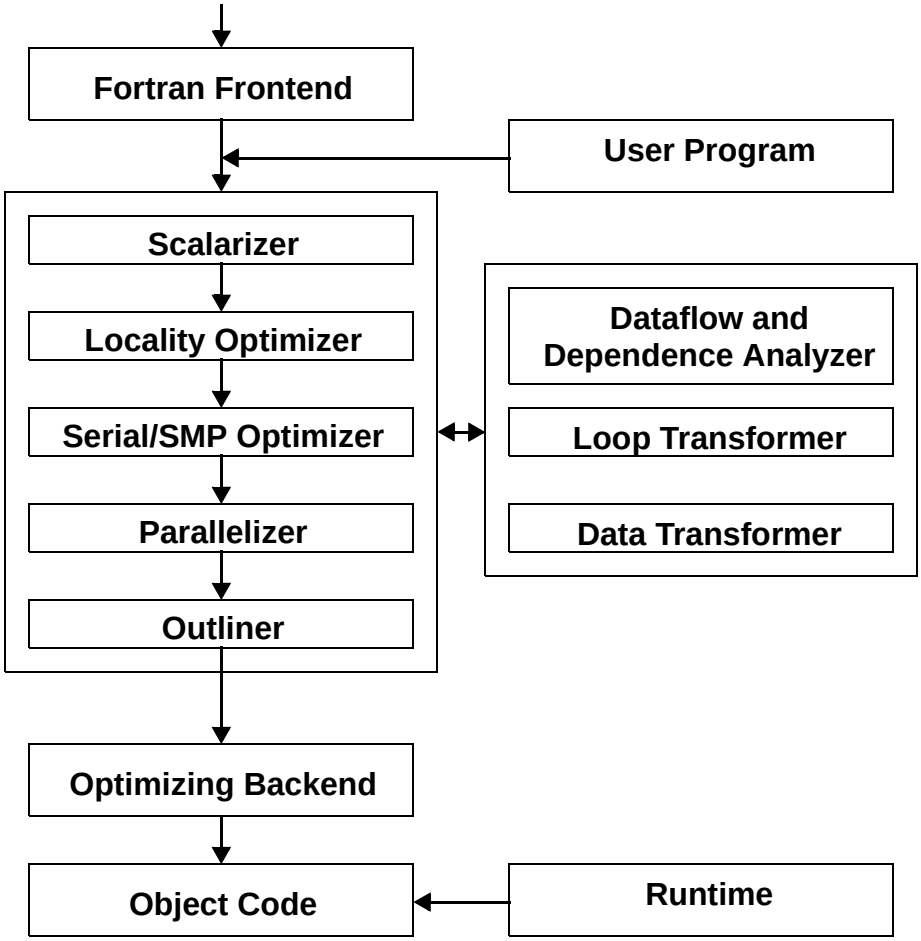


Figure 5. XL Fortran Version 5 Compiler Architecture

In addition to automatic parallelization, XL Fortran Version 5 provides the pthreads library module (f_pthread) as an interface to the AIX pthreads library. See *XL Fortran for AIX Language Reference Version 5 Release 1*, SC09-2607 or "XL Fortran Compiler for IBM SMP Systems," *AIXpert Magazine*, December 1997 for details.

3.2 Support for POWER3

The XL Fortran compiler has a -qarch option that tries to produce efficient object code that may contain machine instructions specific to the target architecture. The default value is -qarch=com, which means the executable can be run on any hardware platform of POWER and PowerPC, but in order to fully use the hardware's capability, it is recommended to use the appropriate -qarch option, especially for scientific and technical applications. In addition to architectures supported by XL Fortran Version 4.1 (that is, com, pwr, pwr2, ppc, and so forth), XLF Version 5.1 introduces two new architectures: pwr3 (V5.1.1) and rs64a (V5.1.0). Currently, RS/6000 Model 260 (and its corresponding SP nodes) and RS/6000 S70 (S7A) conform to pwr3 and rs64a, respectively.

Since XL Fortran does not optimize the program by default, you should specify appropriate options when compiling. To begin with, it is recommended to use the following combination of compiler options for POWER3 machines:

```
$ xlf -qarch=pwr3 -O3 -qtune=pwr3 yourprogram.f
```

The -O3 option instructs the compiler to do the highest level optimization. This optimization level has the potential to rearrange the semantics of the programs. Although it produces a mathematical equivalent result, it may not produce a bitwise identical result with the unoptimized code. If this is a concern, you can add the -qstrict option to ensure that you get the bitwise identical results with the unoptimized code. The -qarch and -qtune options both perform architecture-dependent optimization for the POWER3. Further tuning of compiler options should be carried out with these options as a starting point. More detailed discussions on compiler options are given in 8.2, "Recommended Compiler Options" on page 112.

3.3 64-Bit Support

In order to be able to exploit the huge address space offered by 64-bit addressing, Fortran programmers need to understand how memory is handled by the AIX and the XL Fortran compiler, both in 32-bit and 64-bit mode. This section gives both the background and some practical implications of 32-bit and 64-bit addressing.

In AIX, virtual memory is divided into segments. In 32-bit mode, a 32-bit address is divided into a 28-bit field, which gives the offset within a 256 MB (2^{28} bytes) segment, and a 4-bit field, which selects between 16 segments. In 64-bit mode, 28 bits are again used to address offsets within a 256 MB

segment, but the number of segments which may be addressed is vastly increased.

3.3.1 Fortran Storage Classes

Before explaining how segments are used, it is necessary to understand Fortran storage classes. Each variable belongs to one of the following storage classes:

Automatic	For variables not retained once the procedure ends
Static	For variables which retain memory throughout the program
Common	For common block variables
Controlled Automatic	For automatic arrays
Controlled	For allocatable arrays

From the point of view of the operating system, these classes are categorized as one of the following types:

data Initialized static and common variables

bss Uninitialized static and common variables

heap Controlled (or, allocatable) arrays

stack Controlled automatic arrays and automatic variables

The size of these types, where the size is known before execution begins, may be determined by running the `size` command against the executable as follows:

```
$ size -f a.out
a.out: 1132(.text) + 216(.data) + 134217744(.bss) + 452(.loader) +
12(.except) = 134219556      (32-bit executable)

$ size -X 64 -f a.out
a.out: 1112(.text) + 272(.data) + 134217760(.bss) + 559(.loader) +
20(.except) = 134219723      (64-bit executable)
```

Note that initialized static and common variables and arrays are stored in the data area of the executable file itself; so very large initialized arrays can lead

to very large executable files. These Fortran storage classes and types are mapped onto AIX segments, as listed in Table 4.

Table 4. Fortran Storage Classes and AIX Segments

Fortran Storage Class	Type	AIX segment (32-bit)	AIX segment (32-bit, with -bmaxdata)	AIX segment (64-bit)
Static	Data or BSS	seg. 2 (256 MB)	segs. 3-10 (2 GB)	segs. `0x10` `-0x6FFFFFFF` (4.5×10^5 TB)
Common				
Controlled	Heap			
Automatic	Stack	seg. 2 (256 MB)	seg. 2 (256 MB)	segs. `0xF0000000` `-0xFFFFFFFF` (6.5×10^4 TB)
Controlled Automatic				

Data, bss, and heap are generically termed *user data* and the permissible maximum size of user data is governed by the “data” process limit. Stack is governed by the “stack” process limit. Process limits can be set on a per-user basis in the file /etc/security/limits. Both hard and soft limits may be set in this file. You may then use the `ulimit` command to raise or lower the soft limit up to the hard limit, or to lower (but not raise) the hard limit.

3.3.2 32-Bit Mode

The default mode is the 32-bit mode. As seen from Table 4, all storage classes are allocated to segment 2, a single 256 MB segment. By default in AIX 4.3, the soft limit for user data is 128 MB and for stack is 64 MB. The hard limits are usually set to unlimited by the root user. The linker flags -bmaxdata and -bmaxstack may be used to increase the permissible data and stack sizes beyond the soft limits up to the hard limits, without setting the shell’s process limits using `ulimit`. Note, however, that use of the `-bmaxdata` flag selects the “Large Address Space Model”, described in 3.3.3, “32-Bit Mode, Large Address Space Model” on page 22. If a process exceeds its data limit, it will fail to load if the size of data is known from the object file, or an ALLOCATE statement will fail if the heap grows too large. If the stack limit of a program is exceeded at run time, it will fail with a “Segmentation fault” error message.

Take Note

Care should be taken when increasing the size of data and/or stack. The user data comes from the lower address area of segment 2, whereas the user stack area is allocated from the top of the segment. There are no checks made to ensure that the user stack area doesn't overlap with the user data area. If the stack overwrites the data area, it is possible either for the program to end abnormally, or worse, for the program to fail silently and produce incorrect results.

3.3.3 32-Bit Mode, Large Address Space Model

If the program is linked with the flag -bmaxdata:N, then N bytes are allowed for the user data area, and the user data area is moved from segment 2 to segments 3 through 10, allowing a total of eight segments, or 2 GB, of user data. For example, to allow up to 512 MB, or two segments, of user data, link with the flag -bmaxdata:0x20000000. Note that even if N is less than 256 MB, the user data area resides above segment 2.

As shown in Table 4 on page 21, the user stack area still resides in segment 2. In other words, in either 32-bit mode, the size of the stack (automatic variables and Fortran 90 automatic arrays) is limited to a little less than 256 MB.

Even if a program is linked to use the Large Address Space Model, it is still limited by its stack process limits and its hard data process limit, as explained above.

3.3.4 64-Bit Mode

XL Fortran introduced a new compiler option, -q64, in Version 5.1, which allows the object code to run in 64-bit mode. As seen from Table 4 on page 21, the permissible sizes of stack and user data are huge, although they are still limited by the process limits discussed above. And as with 32-bit mode, -bmaxstack and -bmaxdata may be used to go beyond the soft limits, up to the hard limits, without setting the shell's limits with the `ulimit` command. However, in this case the -bmaxdata flag does not change the addressing model.

3.3.5 Compiler Defaults and Limits

Although the potential size of stack and user data is effectively limited only by the physical memory and paging space installed, there are some other implications of using the -q64 option and 64-bit mode:

- The default size of an integer POINTER (often called Cray pointers or Sun pointers to distinguish them from standard Fortran 90 pointers) is 8 bytes in 64-bit mode.
- The maximum array size increases to approximately 2^{40} bytes.
- The maximum dimension bound range is extended to [-2^{63}, $2^{63}-1$].
- The maximum array size for array constants has not been extended and will remain the same as the maximum in 32-bit mode. The limit depends on the space used by the compiler for a particular program.
- Arrays with a size greater than $2^{31}-1$ cannot be initialized.
- The maximum iteration count for array constructor implied DO loops increases to $2^{63}-1$.
- The maximum character variable length extends to approximately 2^{40} bytes.
- The maximum length of character literals remains the same as in 32-bit mode. This is limited by the maximum length of a single (possibly continued) Fortran statement, currently 6700 characters.
- The LOC intrinsic function returns an INTEGER(8) value.

Important

The default INTEGER and the default REAL size remains 4 bytes in 64-bit mode.

The -q64 option can be combined with -qhot, -O4, -qsmp, and -qipa options in version 5.1.1. Currently, settings for the -qarch option that are compatible with the -q64 option are, -qarch=auto (if compiling on a 64-bit system), -qarch=com, -qarch=ppc, -qarch=rs64a, and -qarch=pwr3. Note that you cannot mix 32-bit and 64-bit object files to create an executable.

3.3.6 64-bit Integer Arithmetic Support

In order to use the POWER3's native 64-bit integer computation, you need to compile the program with the -q64 option, and define integers explicitly in the program as INTEGER*8 or use the -qintsize=8 compiler option to make the

default size of INTEGER to 8 bytes. Integer constants can have INTEGER*8 attribute by adding a suffix _8 as in 123456_8.

Important

In 64-bit mode, use INTEGER*8 loop variables for better performance.

3.4 Performance Improvements over Previous XL Fortran

This section presents results of a benchmark for a customer, and it shows the improved performance of XL Fortran Version 5.1 and the relative performance of the P2SC chip (160 MHz) and the POWER3 chip (200 MHz). The benchmark was done for the following 14 programs:

cfd	Computational fluid dynamics
finite	Finite element method structure analysis iterative eigenvalue solver
modyn	Molecular dynamics
ns3d	3-D computational fluid dynamics
pureg	Monte Carlo simulation of gauge theories QCD
bem3d	3-D transient enclosure flow
crystal	Computational physics software package
jcg3d	3-D solid structure FEM by J-CG solver static, Yale format
chamber	Time-dependent 3-D computational fluid dynamics
deft	Molecular dynamics
enzlong	Life science chemistry
cirta	Computational fluid dynamics
mopac93	Computational chemistry software package (IBM)
gamess	Computational chemistry software package

The programs were run *serial* and, for each program, the sum of user CPU time and system CPU time for the original version and the tuned version was

reported. The RS/6000 systems and the software used for the benchmark are listed in Table 5.

Table 5. The Benchmark Environment

	P2SC/XLF3	P2SC/XLF5	POWER3/XLF5
CPU Clock	160 MHz	160 MHz	200 MHz
Memory	1 GB	1 GB	2 GB
AIX	4.3	4.3	4.3
XL Fortran	3.2.4	5.1.1	5.1.1
Compiler option	-qarch=pwr2 -O3	-qarch=pwr2 -O3	-qarch=pwr3 -O3

The programs were linked with ESSL (for three tuned codes) and the MASS library. For execution on POWER3, a POWER3-enabled ESSL was used. The Fortran preprocessors used were VAST and KAP, which were both 1995 released versions.

Table 6 shows the results of original programs.

Table 6. CPU Time for Original Programs in Seconds

	P2SC/ XLF3 (A)	P2SC/ XLF5 (B)	POWER3/ XLF5 (C)	Ratio (A)/(B)	Ratio (B)/(C)	Prepro- cessor
cfd	125.5	115.3	101.1	1.09	1.14	vast
finite	296.5	289.4	184.0	1.02	1.57	
modyn	744.4	640.5	593.0	1.16	1.08	
ns3d	236.0	237.5	194.4	0.99	1.22	kap
pureg	666.2 676.8	697.0 659.3	532.7 505.3	0.96 1.03	1.31 1.30	kap
bem3d	372.7	347.9	284.4	1.07	1.22	vast
crystal	7901.1	7621.0	6177.8	1.04	1.23	
jcg3d	156.0	155.0	166.5	1.01	0.93	
chamber	28.1	24.5	18.7	1.15	1.31	
deft	9.5	8.4	7.6	1.13	1.11	
enzlong	80.1	67.6	65.2	1.18	1.04	vast
cirta	74.9	73.2	53.2	1.02	1.38	kap

	P2SC/ XLF3 (A)	P2SC/ XLF5 (B)	POWER3/ XLF5 (C)	Ratio (A)/(B)	Ratio (B)/(C)	Prepro-cessor
mopac93	4899.6	3840.2	3824.5	1.28	1.00	
gamess	317.0	352.2	218.7	0.90	1.61	
Average				1.07	1.23	

XL Fortran Version 5.1.1 shows a marked improvement in optimizing these programs on the average of seven percent over Version 3.2.5, and because of this improvement of the compiler, the Fortran preprocessors seem less effective. Only jcg3d became slower on POWER3 than P2SC, whose key kernel is sparse matrix-vector multiplication. The new cache organization and size of POWER3 was not able to hold the indirect addressing vector in cache. However, in general, the load/store units of POWER3 greatly enhanced kernels in these benchmark programs, and when comparing P2SC/XLF5 and POWER3/XLF5, POWER3 was faster by 23 percent on average. It was also observed that the majority of these programs gained performance improvement by using the MASS library.

Table 7 shows the results of tuned programs.

Table 7. CPU Time for Tuned Programs in Seconds

	P2SC/ XLF3 (A)	P2SC/ XLF5 (B)	POWER3/ XLF5 (C)	Ratio (A)/(B)	Ratio (B)/(C)	Note
cfd	69.6	67.3	64.3	1.03	1.05	
finite	114.0	111.6	107.6	1.02	1.04	
modyn	66.3	71.5	59.4	0.93	1.20	
ns3d	164.2	157.4	131.3	1.04	1.20	
pureg	183.4	184.2	167.6	1.00	1.10	
bem3d	69.8	66.1	55.0	1.06	1.20	
crystal						not tuned
jcg3d	87.0	87.1 76.9	72.7 63.4	1.00	1.20 1.21	tune 1 tune 2
chamber	18.9	16.5	15.7	1.15	1.05	
deft	6.3	6.4	6.1	0.98	1.05	

	P2SC/ XLF3 (A)	P2SC/ XLF5 (B)	POWER3/ XLF5 (C)	Ratio (A)/(B)	Ratio (B)/(C)	Note
enzlong	69.3	67.2	64.4	1.03	1.04	
cirta	60.9	52.2	36.8	1.17	1.42	
mopac93	2279.3	2257.5	2058.4	1.01	1.10	
gamess						not tuned
Average				1.03	1.14	

The tuned versions did not need the preprocessor for performance, and the improvement of the compiler had less impact on tuned programs, that is, P2SC/XLF5 was faster than P2SC/XLF3 by only three percent on the average. Still for tuned programs, POWER3/XLF5 was faster than P2SC/XLF5 by 14 percent on the average.

Chapter 4. Using the SMP Feature of XL Fortran

Starting with Version 5.1, the XL Fortran compiler provides an option, -qsmp, which instructs the compiler to automatically parallelize Fortran DO loops. This includes both DO loops coded explicitly by the user and DO loops generated by the compiler for array language constructs, such as FORALL and array assignment. However, the compiler will only automatically parallelize loops that are independent, that is, loops whose iterations can be computed independently of any other iterations.

While automatic parallelization might be sufficient for some users, the SMP directives give you an option of providing additional information about the source code to the compiler. The information you pass to the compiler will either be used during automatic parallelization or to specify that certain parts of the program can be parallelized. For example, a directive ASSERT(ITERCNT(100)) gives an estimate to the compiler about roughly how many iterations the DO loop will typically execute, and the PARALLEL DO directive specifies that the DO loop immediately following it should be executed in parallel.

Some of the directives available for XL Fortran 5.1 conform to the OpenMP Specification Version 1.0 which defines directives and APIs for SMP workstations. Currently, OpenMP is endorsed by more than 20 hardware and software vendors, including IBM. It is probably that the future releases of XL Fortran will become more compatible with OpenMP and that the portability of codes will increase. For details of OpenMP, visit `http://www.openmp.org/`.

In this chapter, only topics that are thought to be useful in parallelizing real applications are discussed. Not all of the SMP features of XL Fortran are explained. For comprehensive documentations, refer to *XL Fortran for AIX Language Reference Version 5 Release 1*, SC09-2607 and *XL Fortran for AIX User's Guide Version 5 Release 1*, SC09-2606.

4.1 How to Compile, Link, and Execute

As an example, consider the following code that adds all the positive integers up to 100:

sample.f

```
PROGRAM MAIN
PARAMETER (N=100)
INTEGER A(N), S
DO I=1, N
```

```
        A(I) = I
      ENDDO
      S = 0
!SMP$ PARALLEL DO REDUCTION(+:S)
      DO I=1, N
        S = S + A(I)
      ENDDO
      PRINT *, S
      END
```

The line beginning with !SMP$ is an example of XL Fortran directive that tells the compiler that the following DO loop should be executed in parallel and that the variable S is used for storing summation. Details of directives will be described later in this chapter. Typically, the preceding code is compiled as follows:

```
$ xlf90_r -qfixed -O3 -qstrict -qsmp sample.f
```

The option -qsmp specifies that the object code may be run in parallel, and that the invocation commands you use should be either xlf_r or xlf90_r so that the code is automatically linked with thread-safe libraries. Otherwise, you have to be responsible for linking with appropriate libraries. If you want two threads for execution, set the XLSMPOPTS environment variable as

```
$ export XLSMPOPTS=parthds=2
```

and if necessary, the value of parthds can be accessed from inside of the code by using the NUM_PARTHDS intrinsic function, whose usage will be illustrated in Section 4.7, “NUM_PARTHDS Intrinsic Function” on page 56. The default value of parthds is the number of on-line processors of the machine.

You can see how the code is parallelized (or not) by looking into the .lst file produced by the smplist suboption of the -qreport option:

```
$ xlf90_r -qfixed -O3 -qstrict -qsmp sample.f -qsource -qreport=smplist
```

Note that this report is produced before loop and other optimizations are performed. The contents of sample.lst are as follows. (The options section and tail sections are omitted.)

```
>>>>> SOURCE SECTION <<<<<
          1 |      PROGRAM MAIN
          2 |      PARAMETER (N=100)
          3 |      INTEGER A(N), S
          4 |      DO I=1, N
          5 |        A(I) = I
          6 |      ENDDO
```

```
          7 |      S = 0
          8 |!SMP$ PARALLEL DO REDUCTION(+:S)
          9 |      DO I=1, N
         10 |        S = S + A(I)
         11 |      ENDDO
         12 |      PRINT *, S
         13 |      END
** main   === End of Compilation 1 ===

>>>>> PARALLELIZATION AND LOOP TRANSFORMATION SECTION <<<<<

1585-107  *** SMP Parallelization Report ***

       program main()
       integer*4 :: main
       integer*4 :: a(1:100)
       integer*4 :: s
       integer*4 :: i
       address :: #ALLOCATEMP
       integer*4 :: #1
       integer*4 :: wct_1
       integer*4 :: SSA_STACK_2
       integer*4 :: SSA_STACK_4
       external :: main
         integer*4 :: main
       external :: __trap
       external :: _xlfBeginIO
         integer*4 :: _xlfBeginIO
       external :: _xlfWriteLDInt
       external :: _xlfEndIO
         integer*4 :: _xlfEndIO
       external :: _xlfExit
       external :: TRAP
       program main()
       #ALLOCATEMP = 0
C 1585-501  Original Source Line 4
       PARALLEL do i=1,100,1
         a(i) = i
       end do
       s = 0
C 1585-501  Original Source Line 9
       PARALLEL do i=1,100,1
         s = s + a(i)
       end do
       #1 = _xlfBeginIO(6,257,0,0,0,0,0)
       call _xlfWriteLDInt(#1,s,4,4)
       wct_1 = _xlfEndIO(#1)
       call _xlfExit(0)
       TRAP(3)
       return
       end
```

In the report, PARALLEL do indicates that the following loop is parallelized. In this case, both the initialization loop and the summation loop are parallelized as expected. In 4.4.2, “XL Fortran Messages Related to Parallelization” on page 44, you will see what kind of messages XL Fortran outputs when it does not parallelize particular loops. If you specify -qreport=hotlist, even more detailed information will be reported. The following is a part of the hotlist of

the sample code where the sum is calculated. (For explanation and readability, the line numbers are added and the line continuation is modified.)

```
>>>>> PARALLELIZATION AND LOOP TRANSFORMATION SECTION <<<<<

1585-103  *** Loop Transformation Report ***
...

1585-105  *** Outlining Report ***
...

   1        s = 0
   2 C 1585-501  Original Source Line 9
   3        if ((_xlsmpInCritical() .eq. 0 .and. _xlsmpInParallel() .eq. 0
   4      &      .and. (1)) .ne. 0) then
   5          __pardo_do_ctl_13(1) = int(1)
   6          __pardo_do_ctl_13(2) = int(100)
   7          __pardo_do_ctl_13(3) = int(1)
   8          __pardo_chunk_ctl_14(1) = 1
   9          __pardo_chunk_ctl_14(2) = 5
  10          __pardo_chunk_ctl_14(3) = 0
  11          __pardo_chunk_ctl_14(4) = 0
  12          __pardo_flags_15 = 3
  13          call _xlsmpParDoSetup(__pardo_flags_15,0,
  14      &                         __pardo_do_ctl_13,
  15      &                         __pardo_chunk_ctl_14,
  16      &                         __main_out_2,
  17      &                         NARGS(__main_out_2) - 1,
  18      &                         PERCENTARG(1,__main_out_2,2),
  19      &                         PERCENTARG(100,__main_out_2,3),
  20      &                         PERCENTARG(1,__main_out_2,4),a,s)
  21        else
  22 C 1585-501  Original Source Line 9
  23          do i=1,100,1
  24            s = s + a(i)
  25          end do
  26        end if
  27        #1 = _xlfBeginIO(6,257,0,0,0,0,0)
  28        call _xlfWriteLDInt(#1,s,4,4)
  29        wct_1 = _xlfEndIO(#1)
  30        call _xlfExit(0)
  31        TRAP(3)
  32        return
  33       contains
  34        subroutine __main_out_2(__lib_ctl_2,
  35      &                         __do_from_2,
  36      &                         __do_to_2,
  37      &                         __do_step_2,a_2,s_2)
  38        integer*4 :: __lib_ctl_2
  39        integer*4 :: __do_from_2
  40        integer*4 :: __do_to_2
  41        integer*4 :: __do_step_2
  42        integer*4 :: a_2(1:100)
  43        integer*4 :: s_2
  44        integer*4 :: __pardo_from_2
  45        integer*4 :: __pardo_to_2
  46        integer*4 :: __pardo_step_2
  47        integer*4 :: i_2
  48        integer*4 :: local_accum_s_2
  49        local_accum_s_2 = 0
```

```
50 C 1585-501  Original Source Line 9
51        do while (_xlsmpParDoChunk(__lib_ctl_2,
52      &                            __pardo_from_2,
53      &                            __pardo_to_2,
54      &                            __pardo_step_2) .eq. 1)
55          do i_2=__pardo_from_2,__pardo_to_2,__pardo_step_2
56            local_accum_s_2 = local_accum_s_2 + a_2(i_2)
57          end do
58          i_2 = i_2 - __pardo_step_2 + __do_step_2
59        end do
60        call _xlsmpGetDefaultSLock(__lib_ctl_2)
61        s_2 = s_2 + local_accum_s_2
62        call _xlsmpRelDefaultSLock(__lib_ctl_2)
63        return
64        end
65        end
```

The hotlist is a pseudo-Fortran listing, which is not meant to be compiled as it is, but you can see how the compiler outliner converts the DO loops into subroutines. The IF statement in lines 3 and 4 decides whether to execute the DO loop in parallel (lines 5-20) or in serial (lines 23-25). This is because the DO loop may not be executed in parallel depending on whether the loop is a nested parallel loop or whether the loop appears in a critical section. In addition, the IF clause may control whether a loop is parallelized (section 4.6.1.3, “IF” on page 53). You do not see any DO statements in lines 5-20. Instead, the DO loop is converted to a subroutine call to `_xlsmpParDoSetup`, which divides the work into chunks and assigns these chunks to threads indicating which procedure to execute, that is, `__main_out_2` defined in lines 34-64, and which arguments to pass to this subroutine (lines 18-20). In `__main_out_2`, each thread is supposed to calculate the sum of its assigned portion into the variable `local_accum_s_2`, and this local sum is added to the global sum, `s_2`. The lock mechanism (lines 60 and 62) assures that only one thread can change the value of `s_2` at a time.

4.2 Consideration of Storage Classes in 32-Bit Mode

When using the -qsmp option and running a program in parallel in 32-bit mode, it is important to understand the relationship between the types of variables that appear in the loop and the limits on their size. Table 4 on page 21 shows the XL Fortran storage classes and their corresponding AIX VMM segments.

Data in the user data area (that is, data, bss, and heap) are shared among all the threads that belong to the same process. On the other hand, data in the user stack area is assigned memory individually per procedure call and is not shared among threads (even if they are calling the same subroutine or function). Loop iteration variables, variables for reduction operations, and

temporary variables in a loop should not be shared among threads in order for the loop to be executed correctly.

As can be seen in the /etc/xlf.cfg file, xlf_r uses -qsave by default, whereas xlf90_r uses -qnosave. In other words, the default storage class is static when a module is compiled with xlf_r, and automatic with xlf90_r. According to the section on the -qsmp option in *XL Fortran for AIX User's Guide Version 5 Release 1*, SC09-2606, it is recommended to use the -qnosave option to make the default storage class automatic when a code is compiled by xlf_r with the -qsmp option. Therefore, when you use the -qsmp compiler option, the variables and arrays in your program are likely to be stored in the user stack area, which was not the case when you compiled programs with xlf for single thread execution.

Here, another complexity is introduced in 32-bit mode regarding the maximum size of data in the user data area and the user stack area, as explained in sections 3.3.2, "32-Bit Mode" on page 21 and 3.3.3, "32-Bit Mode, Large Address Space Model" on page 22. For data in the user data area, the maximum size is 256 MB (that is, the segment size of AIX) by default, but can be extended as much as 2 GB by using the -bmaxdata compiler option, which allows you to allocate memory across multiple segments. For data in the user stack area, however, the maximum size per procedure call is 256 MB and cannot go beyond the limit of AIX segment size. Table 8 summarizes the preceding argument.

Table 8. Storage Areas and Their Maximum Sizes

	User Data Area	User Stack Area
Variable Type	Variables in common block Variables with SAVE attribute (Default of xlf and xlf_r is -qsave) Allocatable arrays	Variables with NOSAVE attribute (Default of xlf90 and xlf90_r is -qnosave)
Characteristics	These variables are kept static in the user data area.	Memory area for these variables is allocated when a procedure is called, and will not be retained once the procedure ends.
Maximum size	AIX default value is 128 MB. Can be 256 MB by using the `ulimit` command. Up to 2 GB is possible by the -bmaxdata compiler option.	AIX default value is 64 MB. Can be 256 MB by using the `ulimit` command or by the -bmaxstack compiler option, but no more.

Now consider the following code that requires 512 MB of memory for array A:

storage.f

```
PROGRAM MAIN
PARAMETER (N=512*1024*1024/8)
REAL*8 A(N)
DO I=1,N
  A(I)=I
ENDDO
END
```

If this program is compiled with xlf -bmaxdata:600000000, there is no problem in running it serial in 32-bit mode. But if storage.f is compiled with the -qnosave option, as recommended when you use the -qsmp option, the resulting executable cannot be executed in 32-bit mode, because the operating system tries to store array A in the user stack area, but it has at most 256 MB of memory available. Since in parallelizing codes, XL Fortran divides loops into sets of disjoint iterations and allocates them to threads, the array A can be shared among threads without interfering with each other. Therefore, in this case, you can legitimately declare A as SAVE, which causes A to be stored in the user data area.

storage.f (modified)

```
PROGRAM MAIN
PARAMETER (N=512*1024*1024/8)
REAL*8 A(N)
SAVE A              ! The array A is stored in the user data area.
DO I=1,N
  A(I)=I
ENDDO
END
```

This code can be compiled as,

```
$ xlf90_r -qfixed -qsmp -bmaxdata:600000000 storage.f
```

or

```
$ xlf_r -qnosave -qsmp -bmaxdata:600000000 storage.f
```

and XL Fortran will automatically parallelize the DO loop and generates an executable for multi-threaded execution. In fact, the original version of storage.f can be compiled with the -qsave option and be executed in parallel, in this case, because the compiler automatically generates loop iteration variables that are local to threads.

In parallel execution in 32-bit mode, you should also be careful in sizing automatic arrays that are used in subroutines. The memory area that you need in the user stack area for a certain subroutine is (the number of threads executing the subroutine concurrently) x (the size of arrays).

Important

In 32-bit mode, the user stack area is limited by 256 MB. More user stack area will be consumed in parallel execution than in serial because (1) recommended storage class is NOSAVE and (2) each thread needs its own copy of stack. Never underestimate the size needed for the user stack area.

4.3 Conditions for Automatic Parallelization

Without directives, XL Fortran only tries to parallelize DO loops. Only DO loops with iteration variables are considered for parallelization.

Loop that will possibly be parallelized

```
      DO I=1,N
      ...
      ENDDO
```

Loops that will not be parallelized

```
C Infinite loop
      DO
      ...
      ENDDO

C DO-WHILE structure
      DO WHILE (...)
      ...
      ENDDO

C Non-DO loop
  100 CONTINUE
      ...
      GOTO 100
```

The compiler analyzes the loop to find out whether each iteration is independent of one another or not, and if it turns out to have parallelism, the compiler further estimates the benefit of parallelization by a cost-based

analysis to make the final decision. The former analysis (parallelism analysis) makes use of information that is available within the procedure that contains the loop under consideration. There are several conditions in order for a loop to be parallelized (the following is not a complete list):

1. Each iteration is independent of each other, that is, no variables that are written in some iteration will be read and/or written in another iteration.
2. The program will not exit from the loop before the last iteration is executed.
3. There are no I/O statements in the loop.
4. There are no ALLOCATE or DEALLOCATE statements in the loop.
5. In nested loops, at most one loop can be parallelized. Therefore, a loop in a certain nest level will not be parallelized if another level is.

More details will be discussed in 4.4, “Automatic Parallelization - Parallelism Analysis” on page 38 by showing examples. In the remainder of this section, general discussion on dependences between iterations are given (see Bacon, Graham, and Sharp, “*C*ompiler Transformations for High-Performance Computing,” *ACM Computing Surveys*, Vol. 26, 1994).

There are two kinds of dependences: control dependence and data dependence. Control dependence between statements s_1 and s_2 means that s_1 determines whether s_2 is executed, or vice versa. The following is an example of control dependence:

```
s1    IF (I>MAX) GOTO 100
s2    I=I+1
```

The condition 2 of the preceding list is more precisely expressed as,

2’. There are no control dependences between iterations.

Two statements have a data dependence if they cannot be executed simultaneously due to conflicting uses of the same variable. There are three types of data dependences: flow dependence, anti-dependence, and output dependence. A statement s_3 has a flow dependence on s_4 when s_3 must be executed first because it writes a variable that is read by s_4 as follows:

```
s3    A(I)   = A(I-1) + 1.0
s4    A(I+1) = A(I)   + 1.0
```

A statement s_6 has an anti-dependence on s_5 when s_6 writes a variable that is read by s_5:

```
s5    A(I-1) = A(I)   + 1.0
```

s_6 `A(I) = A(I+1) + 1.0`

In the preceding example, anti-dependence can be eliminated by storing the value of `A(I)` to a temporary variable, say T, before the execution of s_5 and s_6, and by using T instead of `A(I)` in s_5.

Statements s_7 and s_8 have output dependence if both write the same variable:

```
s7    T = A(I)
s8    T = A(I+1)
```

These three data dependences and combinations of them constitute cases that violate the first condition. They prohibit automatic parallelization in principle, but in some cases where dependent variables are only used temporarily and are insignificant outside the iteration, or when you can use directives, parallelization might be possible.

4.4 Automatic Parallelization - Parallelism Analysis

Ideally, the compiler parallelizes all the DO loops that can be parallelized at all. During compilation, there may be a lack of sufficient information in the code for the compiler to make an analysis, thus the compiler automatically parallelizes loops, and in the other, it needs assistance through the use of directives. In either cases, it is important that you know, to some extent, how the compiler tries to analyze and transform DO loops for parallel execution.

4.4.1 Examples of Parallelism Analysis

Subsections from 4.4.1.1, “Loops That Have Parallelism” on page 38 through 4.4.1.10, “Dynamic Allocations, and Pointer Substitutions” on page 44 show how structures in DO loops allow or disallow the compiler to parallelize them. Note that some of the examples might have too few iterations to pass the cost-based analysis following the parallelism analysis, but they are for explanation purposes only and loops that are automatically parallelized usually have more than the minimum number of iterations in their cases.

4.4.1.1 Loops That Have Parallelism

Each iteration in the following loop is independent of each other and can be parallelized automatically. By declaration of A and B, you (and the compiler) know that these two array do not overlap in memory, that is, no equivalence relation between any elements of A and B.

```
REAL*8 A(100), B(100)
DO I=1,100
```

```
  A(I)=B(I)
ENDDO
```

If this loop is to be executed concurrently by two threads, each thread should execute half of the whole iterations:

Thread 0

```
DO I0=1,50
  A(I0)=B(I0)
ENDDO
```

Thread 1

```
DO I1=51,100
  A(I1)=B(I1)
ENDDO
```

As mentioned in 4.2, “Consideration of Storage Classes in 32-Bit Mode” on page 33, the compiler takes care of the loop iteration variable regardless of whether it is static (SAVE) or automatic (NOSAVE). In the preceding example, symbolic names (I_0 and I_1) are used for non-shared thread-local loop iteration variables, but you can guess how the compiler actually translates this loop by looking into hotlist report as follows. (The line continuation is modified for readability.)

```
      subroutine ___main_out_1(__lib_ctl_1,
     &                         __do_from_1,
     &                         __do_to_1,
     &                         __do_step_1,a_1,b_1,i_1,CLLIV_4_1)
      integer*4 :: __lib_ctl_1
      integer*4 :: __do_from_1
      integer*4 :: __do_to_1
      integer*4 :: __do_step_1
      real*8 :: a_1(1:100)
      real*8 :: b_1(1:100)
      integer*4 :: i_1
      integer*4 :: CLLIV_4_1
      integer*4 :: local_i_1
      integer*4 :: local_CLLIV_4_1
      integer*4 :: __pardo_from_1
      integer*4 :: __pardo_to_1
      integer*4 :: __pardo_step_1
      integer*4 :: _do_executed_T_15
      local_CLLIV_4_1 = CLLIV_4_1
      local_i_1 = i_1
C 1585-501  Original Source Line 2
      do while (_xlsmpParDoChunk(__lib_ctl_1,
     &                           __pardo_from_1,
     &                           __pardo_to_1,
     &                           __pardo_step_1) .eq. 1)
        _do_executed_T_15 = 0
        do local_CLLIV_4_1=__pardo_from_1,__pardo_to_1,__pardo_step_1
          _do_executed_T_15 = 1
          local_i_1 = local_CLLIV_4_1
          a_1(local_i_1) = b_1(local_i_1)
```

```
      end do
      local_CLLIV_4_1 = local_CLLIV_4_1 - __pardo_step_1 + __do_step_1
      if ((_do_executed_T_15 .eq. 1 .and.
     &       (__do_step_1 .gt. 0 .and. local_CLLIV_4_1 .gt. __do_to_1
     &        .or.
     &        __do_step_1 .lt. 0 .and. local_CLLIV_4_1 .lt. __do_to_1)
     &      .and. (1)) .ne. 0) then
        i_1 = local_i_1
        CLLIV_4_1 = local_CLLIV_4_1
      end if
    end do
    return
    end
```

This hotlist report is generated with the -qsave option and it shows that the DO loop is converted into a subroutine so that each thread can execute its own assignment and that a loop iteration variable `local_i_1` which is local to thread, is used to avoid shared access by threads.

By default, loops are divided into a set of iterations in a block scheduling fashion, but you can choose cyclic scheduling, block-cyclic scheduling, or dynamic scheduling by specifying SCHEDULE directive, which will be explained in section 4.6.4, “SCHEDULE Compiler Directive” on page 54.

4.4.1.2 Loops That Have Flow Dependence

The following loop will not be parallelized because it has flow dependence:

```
DO I=2,N
  A(I)=A(I-1)+B(I)
ENDDO
```

When the loop is unrolled iteration by iteration, you can see the difficulty in parallelization:

```
A(2)=A(1)+B(2)       (iteration 2)
A(3)=A(2)+B(3)       (iteration 3)
A(4)=A(3)+B(4)       (iteration 4)
A(5)=A(4)+B(5)       (iteration 5)
...
```

The variable `A(3)`, for instance, is updated in the iteration 3 and this updated value is used in the iteration 4. This is a true recursive. Therefore the iterations 3 and 4 must be executed in this order and cannot be exchanged nor be executed concurrently. This is why loops with flow dependence cannot be parallelized automatically, but the preceding discussion has some suggestions in parallelizing them manually: suppose iterations 2 and 3 are assigned to thread 0 and 4 and 5 to thread 1. In this case, threads 0 and 1 can be executed concurrently *if* thread 1 uses the value of A(3) written by thread 0, that is, it is the variables on the boundary between threads that matter in parallelization, and you can get rid of this dependence by using the

same technique as prefix sum. In section 4.7, “NUM_PARTHDS Intrinsic Function” on page 56, you will see how to parallelize this loop using NUM_PARTHDS intrinsic function.

4.4.1.3 Loops That Have Anti-Dependence

Loops with anti-dependence also prevent the compiler from parallelization:

```
DO I=1,N-1
  A(I)=A(I+1)+B(I)
ENDDO
```

Again, for the purpose of illustrative understanding of the loop structure, the first few iterations are written down explicitly:

```
A(1)=A(2)+B(1)        (iteration 1)
A(2)=A(3)+B(2)        (iteration 2)
A(3)=A(4)+B(3)        (iteration 3)
A(4)=A(5)+B(4)        (iteration 4)
...
```

It is easy to see that these iterations cannot be executed concurrently. Although it might not be beneficial in a performance point of view, it is possible to parallelize the loop manually by using a temporary array:

```
DO I=1,N-1
  T(I)=A(I+1)
ENDDO
DO I=1,N-1
  A(I)=T(I)+B(I)
ENDDO
```

4.4.1.4 Temporary Variables

Temporary variables that appear in a loop can impose both anti-dependence and output dependence on the loop:

```
DO I=1,N
  T = B(I)
  A(I) = T
ENDDO
```

In the following, dependence is considered not in terms of statement but in terms of iteration, which is suitable for discussing loop parallelization. Look at iterations I and I+1. (Subscripts for Ts are for explanation purpose only.)

```
T1 = B(I)          (iteration I)
A(I) = T2
T3 = B(I+1)        (iteration I+1)
A(I+1) = T4
```

Iteration I has anti-dependence on I+1 because of T_2 and T_3. At the same time, both iterations have output dependence since they write to the same variable (T_1 and T_3). By preparing two variables, one for T_1 and T_2, the other for T_3 and T_4, the dependences can be eliminated because statements in each iteration is assured to be executed in order. In this simple case, the compiler automatically parallelizes the loop by providing local temporary variables for each thread regardless of whether T is static (SAVE) or automatic (NOSAVE). If the value of T is referred after the loop, the compiler makes sure that the variable T holds the same value as when the loop is executed serially. Suppose a loop with temporary variables is not parallelized automatically and you force the compiler to parallelize it. Depending on whether these temporary variables are referred to after the loop, there are two clauses to the PARALLEL DO directive, namely PRIVATE and LASTPRIVATE, which will be explained in section 4.6.1.1, “PRIVATE and LASTPRIVATE” on page 51.

4.4.1.5 Conditions

The following loop will not be parallelized because the first occurrence of I such that IFLAG(I) equals 1 affects the remaining iterations:

```
T=0.0
DO I=1,N
  IF (IFLAG(I)==1) T=1.0
  A(I)=T
ENDDO
```

On the other hand, the compiler automatically parallelizes the following:

```
DO I=1,N
  IF (IFLAG(I)==1) THEN
    T=1.0
  ELSE
    T=0.0
  ENDIF
  A(I)=T
ENDDO
```

In the current implementation of XL Fortran 5.1.1, dependence between iterations including IF statements must be observable to the compiler syntactically, not semantically, for automatic parallelization. For instance, the following code is not parallelized:

```
DO I=1,N
  IF (IFLAG(I)==1) T=1.0
  IF (IFLAG(I)/=1) T=0.0
  A(I)=T
ENDDO
```

Future IBM implementations of XL Fortran might behave in a different manner.

4.4.1.6 Reduction Operations

XL Fortran automatically parallelizes the following code if the -O3 option is set. Obviously, there are flow dependence and output dependence between iterations in reduction operations, but the compiler transforms the loop for parallel execution.

```
S=0.0
DO I=1,N
  S=S+A(I)
ENDDO
```

The additional compiler flag -qstrict will prevent parallelization of the preceding loop. In concurrent execution, threads calculate the subtotal of array A, and those subtotals are added to produce the total. The order of summation could be different from what it will be if the loop is executed serially; thus the option -qstrict must not be set for parallelization. Indeed the results could be different from execution to execution within numerical error depending on the order in which subtotals are added up. In 4.6.1.2, “REDUCTION” on page 52, a method is presented to force the compiler to parallelize the reduction operation. The following shows examples of reduction operations that can be parallelized:

- Scalar = scalar *op* expression

  ```
  S = S + A(I)
  S = S * A(I)
  S = S + A(I)*B(I)
  ```

- Scalar = *func*(scalar, expression)

  ```
  AMAX = MAX(AMAX, A(I))
  AMIN = MIN(AMIN, A(I))
  ```

4.4.1.7 Indirect Addressing

The compiler does not parallelize the following loop because it cannot determine whether there is an output dependence or not:

```
DO I=1,N
  A(INDEX(I))=B(I)
ENDDO
```

If there exist J and K such that $1 \le J, K \le N$, $J \ne K$, and INDEX(J)=INDEX(K), the loop has indeed an output dependence. If you know that it is not the case, you can tell the compiler of this fact by giving the PERMUTATION directive (Section 4.6.3, “PERMUTATION Compiler Directive” on page 54).

4.4.1.8 Subroutine Calls

The compiler does not automatically parallelize a loop containing subroutine calls and/or function calls. In the smplist report, the compiler outputs messages as follows:

```
C 1585-108  SMP: Did not parallelize this loop potentially because:
C 1585-111  Side effects of procedure call(s) cannot be determined.
```

It is your responsibility whether to parallelize the loop with directives, such as PARALLEL DO and CNCALL, or not.

4.4.1.9 I/O Operations

The compiler does not parallelize a loop having I/O statements.

4.4.1.10 Dynamic Allocations, and Pointer Substitutions

The compiler does not parallelize the following loops:

loop 1

```
REAL, ALLOCATABLE :: A(:)
DO I=1,100
  ALLOCATE(A(1000))
  ...
  DEALLOCATE(A)
ENDDO
```

loop 2

```
POINTER P
TARGET A(100)
DO I=1,100
  P=>A(I)
  ...
ENDDO
```

4.4.2 XL Fortran Messages Related to Parallelization

There are several messages that the compiler outputs regarding parallelization when a source code is compiled with the -qreport=smplist option. When a DO loop is automatically parallelized, you will see a listing like the following:

```
PARALLEL DO I=1,100,1
   A(I) = B(I)
END DO
```

On the other hand, if the compiler fails in automatic parallelization, it puts messages before the loop; this accounts for the reason why the loop was not parallelized.

```
C 1585-108  SMP: Did not parallelize this loop potentially because:
C 1585-113  Data dependence prevents parallelization.
       DO I=2,100,1
         ScRep_3 = A(I - 1) + B(I)
         A(I) = ScRep_3
       END DO
```

The first message `1585-108  SMP: Did not parallelize this loop potentially because:` is common and a detail reason is given in the following message(s). The whole list is listed as follows:

- 1585-109 Granularity of computation is relatively small.
- 1585-110 Loop has loop carried control dependence.
- 1585-111 Side effects of procedure call(s) cannot be determined.
- 1585-112 Dependence information is not precise.
- 1585-113 Data dependence prevents parallelization.
- 1585-114 Parallelization may result in poor cache locality.
- 1585-115 Loop nest needs to be serial for better cache locality.

Messages from 1585-110 to 1585-113 show that the loop was not parallelized by parallelism analysis and the others show that the compiler decided not to parallelize it by cost-based analysis, which is explained in 4.5, “Automatic Parallelization - Cost-Based Analysis” on page 45.

4.5 Automatic Parallelization - Cost-Based Analysis

Even if the parallelism analysis found that a DO loop could be executed in parallel, that DO loop must pass cost-based analysis in order to be parallelized. The logic of cost-based analysis is not documented in manuals, but obviously, it takes into consideration cache locality (that is, stride) and granularity of work assigned to each thread. What is described in the following sections is based on experiments run on XL Fortran Version 5.1.1 and is subject to change in any future release of XL Fortran or service update.

4.5.1 Cost-Based Analysis - Single Loops

In the cost-based analysis, the compiler primarily takes into account the number of iterations of DO loops:

```
SUBROUTINE SUB1(A,MAX)
PARAMETER (N=10)
DIMENSION A(MAX)
DO I=1,N
  A(I)=I
ENDDO
END

SUBROUTINE SUB2(A,IMAX)
DIMENSION A(10)
DO I=1,IMAX
  A(I)=I
ENDDO
END
```

In subroutine SUB1, the number of iterations is explicitly given within the subroutine because variables defined by PARAMETER statements are replaced by actual values. In subroutine SUB2, the value of IMAX is unknown but the compiler assumes that it is the same as the dimension of A, that is, 10.

The compiler parallelizes an unnested DO loop when the number of iterations is unknown, or is greater than or equal to a certain threshold value. Otherwise the loop is not parallelized and smplist reports the reason as `C 1585-109 Granularity of computation is relatively small`. The default threshold value is 100 in XL Fortran 5.1.1.

4.5.2 Cost-Based Analysis - Nested Loops

In case of nested loops, the compiler decides to parallelize them, or not, based on the numbers of iterations of all nested levels. Examine the double loops first. In the loop below, whether it is parallelized or not depends on both JMAX and IMAX:

```
DO J=1,JMAX
  DO I=1,IMAX
    A(I,J)=B(I,J)
  ENDDO
ENDDO
```

In this simple example, it is always the outer loop that is parallelized, if the nested loop is parallelized at all. If the outer loop has no parallelism and the inner one does, the compiler tries to parallelize the inner loop according to the same criteria for single loops:

```
DO J=1,JMAX                ! Not parallelized
  CALL SUB
```

```
  DO I=1,IMAX              ! Might be parallelized
    A(I,J)=B(I,J)
  ENDDO
ENDDO
```

For loops with more than two levels, the same argument applies except that the compiler may possibly change the order of loops. For instance, a nested loop:

```
DO K=1,4
  DO J=1,1000
    DO I=1,1000
      A(I,J,K)=B(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

is parallelized as follows:

```
DO J=1,1000                ! Parallelized
  DO K=1,4                 ! Not parallelized
    DO I=1,1000            ! Not parallelized
      A(I,J,K)=B(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

Note that loop J and loop K are exchanged.

4.5.3 How to Affect the Decision of Cost-Based Analysis

XL Fortran estimates the benefit of parallelization according to its own logic, which is not always ideal. There are cases where DO loops are not parallelized while they should be, or DO loops are parallelized even if the performance degrades. In this section, some techniques are presented for changing how the compiler estimates loops.

As described in 4.5.1, “Cost-Based Analysis - Single Loops” on page 45 and 4.5.2, “Cost-Based Analysis - Nested Loops” on page 46, the information that the compiler uses in cost-based analysis is the number of iterations of loops. Therefore, if the compiler presumed the value that you wish for the number of iteration of some loop, the compiler would behave as you wish regarding parallelization of the loop. For that purpose, there is a directive called ASSERT(ITERCNT(N)) that tells the compiler to use n in the evaluation of the number of iterations of the loop immediately following the directive. Since the value of n is used only in the cost-based analysis, you can specify a different

number from the one that the loop actually iterates. The following are examples of how to use this directive.

The DO loop in the following subroutine is parallelized because the value of N is unknown:

```
      SUBROUTINE SUB(A,N)
      DIMENSION A(N)
      DO I=1,N                  ! Parallelized
        A(I)=0.0
      ENDDO
      END
```

If you know that the value of N is small and that parallelization will degrade the performance, you can give the compiler a small value and serialize the loop, or you can force the compiler to serialize the loop by the DO SERIAL directive:

```
      SUBROUTINE SUB(A,N)
      DIMENSION A(N)
!SMP$ ASSERT(ITERCNT(1))         ! DO SERIAL also works
      DO I=1,N                   ! Not parallelized
        A(I)=0.0
      ENDDO
      END
```

Even if the number of iterations is explicitly given in the code, the directive can be used. The following is a case where for some reason you parallelize a loop against compiler's decision:

```
      SUBROUTINE SUB(A,N)
      DIMENSION A(N)
!SMP$ ASSERT(ITERCNT(1000))
      DO I=1,10                  ! Parallelized
        A(I)=0.0
      ENDDO
      END
```

But in the current implementation of XL Fortran, if the compiler knows that the size of an array is below threshold value, it neglects the directive:

```
      SUBROUTINE SUB(A)
      DIMENSION A(10)
!SMP$ ASSERT(ITERCNT(1000))
      DO I=1,10                  ! Not parallelized
        A(I)=0.0
      ENDDO
      END
```

In parallelizing nested loops, you may have to write an ASSERT directive for a loop which is not the one you want to parallelize:

```
      SUBROUTINE SUB(A)
      DIMENSION A(2,1000)
      DO K=1,1000                ! Not parallelized (but should be)
        DO J=1,2                 ! Not parallelized
          A(J,K)=0.0
        ENDDO
      ENDDO
      END
```

It is the few iteration of J that prevents the compiler from parallelizing K's loop:

```
      SUBROUTINE SUB(A,N)
      DIMENSION A(N,1000)
      DO J=1,1000                ! Parallelized
!SMP$   ASSERT(ITERCNT(1000))
        DO I=1,2                 ! Not parallelized
          A(I,J)=0.0
        ENDDO
      ENDDO
      END
```

Note that you need to hide A's first dimension size from the compiler in order for the directive to work.

There is a trick that does not use ASSERT directive. Suppose that you want to parallelize the inner loop in the following subroutine:

```
      SUBROUTINE SUB(A,M,N)
      DIMENSION A(M,N)
      DO J=1,4                   ! Not parallelized
        DO I=1,1000              ! Not parallelized
          A(I,J)=0.0
        ENDDO
      ENDDO
      END
```

As mentioned in 4.5.2, "Cost-Based Analysis - Nested Loops" on page 46, if the outer loop does not have parallelism, the compiler tries to parallelize the inner one:

```
      SUBROUTINE SUB(A,M,N)
      DIMENSION A(M,N)
      DO J=1,4                   ! Not parallelized
        CALL DUMMY
        DO I=1,1000              ! Parallelized
          A(I,J)=0.0
```

```
  ENDDO
ENDDO
END

SUBROUTINE DUMMY
END
```

A dummy subroutine call prevents the automatic parallelization (Section 4.4.1.8, “Subroutine Calls” on page 44) and the inner loop is parallelized as desired.

If you use the PARALLEL DO directive, you can also parallelize a specific loop that you want. But there is a considerable difference between ASSERT and PARALLEL DO: ASSERT is an assertion directive, that is, it is still up to the compiler whether to parallelize the loop or not. On the other hand, PARALLEL DO is a prescriptive directive that forces the compiler to parallelize the loop regardless of parallelism and cost-based analyses, and it is you who has to take care of variables (other than loop iteration variables) in the loop with appropriate clauses, such as PRIVATE and REDUCTION.

4.6 Directives

When XL Fortran does not parallelize a certain part of a code, you can force or give a hint to the compiler to parallelize that part by using directives. Directives related to parallelization are classified into three categories. The asterisks indicate directives that are described in the following sections.

1. Assertion directives that provide information to the compiler about the source code that the compiler would not necessarily be able to determine on its own:
   - ASSERT
   - CNCALL
   - INDEPENDENT
   - PERMUTATION*
2. Prescriptive directives that specify how and when the compiler should parallelize the code:
   - CRITICAL
   - PARALLEL DO*
   - PARALLEL SECTIONS*
   - SCHEDULE*

- DO SERIAL

3. Thread-safing directive that allocates thread-specific COMMON areas at run time:
   - THREADLOCAL*

Directives are triggered by !SMP$, !$OMP, !IBM*, or others by default, but !SMP$ is used throughout the chapter. The next several sections describe the directives that are considered to be used most frequently in parallelizing real codes. For complete reference of all the directives, see Chapter 11 of *XL Fortran for AIX Language Reference Version 5 Release 1*, SC09-2607.

4.6.1 PARALLEL DO Compiler Directive

When you know that no iteration of a DO loop can interfere with any other iteration and the compiler fails to parallelize the loop automatically, you can specify a PARALLEL DO directive to parallelize the loop. In real codes, it is often the case that it is not enough to write a PARALLEL DO directive alone. In addition to the PARALLEL DO directive a, PARALLEL DO clause, such as PRIVATE or REDUCTION, might be necessary. In this section, parallelization of DO loops having subroutine calls and/or function calls is not described.

The following subsections describe PARALLEL DO clauses.

4.6.1.1 PRIVATE and LASTPRIVATE

A variable should be specified with the PRIVATE attribute, if its value is used during the calculation of a single iteration of a loop, and that value is not dependent on any other iteration of the loop. Copies of the PRIVATE variable exist locally on each thread. All DO loop iteration variables within the lexical extent of the PARALLEL DO directive are given the PRIVATE attribute by default. (The lexical extent of a PARALLEL DO directive includes the corresponding DO loop and the code that is enclosed in that DO loop.) The following is an example where you force the compiler to parallelize a DO loop for some reason, although the compiler automatically parallelizes this simple case without directives:

```
!SMP$ PARALLEL DO PRIVATE(P,Q)
      DO I=1,N
        P=A(I)
        Q=B(I)
        C(I)=P
        D(I)=Q
      ENDDO
```

A variable in the PRIVATE clause must not:

- Be a pointer, or
- Be an assumed-size array, or
- Be an assumed-shape array, or
- Be a THREADLOCAL common block variable.

The LASTPRIVATE clause functions in a similar manner to the PRIVATE clause and should be specified for variables that match the same criteria. The exception is the status of the variable upon exit from the loop. The compiler determines the value of the variable at the final iteration and takes a copy of that value. The copy of the value is then saved in the named variable for use after the loop.

4.6.1.2 REDUCTION

The REDUCTION clause specifies named variables that appear in reduction operations. The compiler will maintain local copies of such variables, but will combine them at loop exit. The intermediate values of the REDUCTION variables are combined in random order, dependent on which threads finish their calculation first. There is, therefore, no guarantee that bit-identical results will be obtained from one parallel run to another, even if the parallel runs use the same number of threads and the same scheduling type and chunk size. The syntax of REDUCTION clause is

REDUCTION([*op_fnc* :] *named_variable_list*)

where *op_fnc* is one of the reduction operators: +, -, *, .AND., .OR., .EQV., .NEQV., .XOR. or one of the reduction functions: MAX, MIN, IAND, IOR, IEOR. In order to maintain compatibility with OpenMP, *op_fnc* must be specified when the directive is triggered by $OMP. The following is an example:

```
!SMP$ PARALLEL DO REDUCTION(+:S1,S2),
!SMP$&            REDUCTION(MAX:CMAX)
      DO I=1,N
        S1=S1+A(I)
        S2=S2+B(I)
        CMAX=MAX(CMAX,C(I))
      ENDDO
```

In the following loop, it is the outer loop that is parallelized, and you need to declare S as PRIVATE rather than REDUCTION:

```
!SMP$ PARALLEL DO PRIVATE(S)
      DO J=1,N                  ! Parallelized
        S=0.0
        DO I=1,N                ! Not parallelized
```

```
      S=S+A(I,J)
    ENDDO
    B(J)=S
  ENDDO
```

4.6.1.3 IF

The IF clause performs a run time test to choose between executing the loop in serial or parallel:

```
!SMP$ PARALLEL DO IF(N>1000)
      DO I=1,N
        A(I)=0.0
      ENDDO
```

4.6.1.4 SCHEDULE

The SCHEDULE clause in a PARALLEL DO directive specifies the chunking method for parallelization of the DO loop immediately following it, whereas a directive starting with SCHEDULE applies to all loops in the scoping unit that do not already have explicit scheduling types specified. Section 4.6.4, “SCHEDULE Compiler Directive” on page 54, describes types that you can choose in the SCHEDULE directive.

4.6.2 PARALLEL SECTIONS Compiler Directive

The PARALLEL SECTIONS directive is used to define independent blocks of code that the compiler can execute concurrently. In using this directive, you have to keep in mind the following:

- The larger the granularity of the independent blocks is, the smaller the relative overhead of parallel execution will be.
- On the other hand, if the granularity of the independent blocks is large, the assigned work for each thread will likely be unbalanced, in general.
- The number of sections is given statically in the code. That is, the number of threads for parallel execution of this part of the code is no more than the number of sections, unless the prescriptive parallel construct (PARALLEL SECTIONS or PARALLEL DO) is nested and you compiled the program with -qsmp=nested_par option. Of course, in either case, the number of threads cannot exceed the value of parthds run-time option. See 4.1, “How to Compile, Link, and Execute” on page 29 for parthds.

The following is a simple example of the PARALLEL SECTIONS directive:

```
!SMP$ PARALLEL SECTIONS
!SMP$ SECTION
```

```
      CALL SUB1
!SMP$ SECTION
      CALL SUB2
!SMP$ END PARALLEL SECTIONS
```

If necessary, you must add the appropriate clause to the PARALLEL SECTIONS directive, such as PRIVATE and SHARED. For details, see *XL Fortran for AIX Language Reference Version 5 Release1*, SC09-2607.

4.6.3 PERMUTATION Compiler Directive

As mentioned in section 4.4.1.7, “Indirect Addressing” on page 43, indirect addressing of array elements prevents the compiler to parallelize the loop containing it. If you know that there are no repeated values in the array used for addressing, you can tell the compiler of that information by using the PERMUTATION directive:

```
!SMP$ PERMUTATION(INDEX)
      DO I=1,N                    ! Parallelized
        A(INDEX(I))=A(INDEX(I))+B(I)
      ENDDO
```

4.6.4 SCHEDULE Compiler Directive

The SCHEDULE directive specifies how the iterations of a DO loop are divided and assigned to threads. The syntax for the SCHEDULE directive is as follows:

```
SCHEDULE(sched_type [, n])
```

where *n* is an integer and *sched_type* is one of AFFINITY, DYNAMIC, GUIDED, RUNTIME, or STATIC. When using RUNTIME, *n* must not be specified. The following shows how each scheduling policy assigns iterations to threads for the case where the number of iterations is 1000 and the number of threads is four:

STATIC If *n* has been specified, say *n*=50, the iterations of a loop are divided into chunks containing 50 iterations. Each thread is assigned chunks in a round-robin fashion. This is known as block cyclic scheduling. If the value of n is 1, then the scheduling type is specifically referred to as cyclic scheduling.
If *n* has not been specified, the iteration is divided into four chunks containing 1000/4=250 iterations and each thread is assigned one of these chunks. This is known as block scheduling.

DYNAMIC If *n* has been specified, say *n*=50, the iterations of a loop are divided into chunks containing 50 iterations each. Otherwise, the chunk size will be 1000/4=250. Threads are assigned these chunks on a first-come, first-do basis until all chunks have been assigned.

GUIDED If *n* has been specified, the iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iteration is reached. If *n* has not been specified, the default value for *n* is 1 iteration. The first chunk contains 1000/4=250 iterations. The subsequent chunks contain (1000-250)/4=750/4=188 iterations, (750-188)/4=562/4=141 iterations, (562-141)/4=106 iterations, and so forth. Available threads are assigned chunks on a first-come, first-do basis. Chunks of the remaining work are assigned to available threads, until all work has been assigned.

AFFINITY The iterations of a loop are initially divided into four partitions containing 1000/4=250 iterations. Each partition is initially assigned to a thread, and is then further subdivided into chunks containing *n* iterations, if *n* has been specified. Otherwise, each partition is subdivided into two chunks. When a thread becomes free, it takes the next chunk from its initially assigned partition. If there are no more chunks in that partition, the thread takes the next available chunk from a partition initially assigned to another thread.

RUNTIME Determine the scheduling type at run time. At run time, the scheduling type can be specified using the environment variable XLSMPOPTS. If no scheduling type is specified by XLSMPOPTS, STATIC is used as default.

If you specify more than one method of determining the scheduling type, the compiler will follow in the order of precedence:

1. SCHEDULE clause of the PARALLEL DO directive (for example, !SMP$ PARALLEL DO SCHEDULE(STATIC,1))
2. SCHEDULE directive (for example, !SMP$ SCHEDULE(STATIC,1))
3. The schedule suboption to the -qsmp compiler option (for example, -qsmp=schedule=static=1)
4. XLSMPOPTS run-time option (for example, XLSMPOPTS=schedule=static=1)
5. Run-time default, that is, STATIC

4.6.5 THREADLOCAL Compiler Directive

In general, data in a COMMON block is shared among all the threads that belong to the same process. The THREADLOCAL directive is used to ensure that a COMMON block is local to each thread but is global within the thread. If a common block is declared as THREADLOCAL within a scoping unit, any subprogram that declares or references the common block, and that is directly or indirectly referenced by the scoping unit, must be executed by the same thread executing the scoping unit.

4.7 NUM_PARTHDS Intrinsic Function

The NUM_PARTHDS intrinsic function returns the number of parallel Fortran threads at run time. With this function and the PARALLEL DO directive, you can determine how the work is divided to threads, which gives more flexibility to you than the SCHEDULE directive does. In this section, an example is given that shows how NUM_PARTHDS is used in parallelizing a loop having flow dependence.

The following subroutine cannot be parallelized because it has flow dependence (see Section 4.4.1.2, “Loops That Have Flow Dependence” on page 40):

```
SUBROUTINE SUB(A,B,N)
DIMENSION A(0:N),B(N)

DO I=1,N
  A(I)=A(I-1)+B(I)
ENDDO
END
```

When the loop exits, the array A has the following values:

```
A(1)=A(0)+B(1)
A(2)=A(0)+B(1)+B(2)
A(3)=A(0)+B(1)+B(2)+B(3)
...
A(N)=A(0)+B(1)+B(2)+B(3)+...+B(N)
```

The idea in parallelizing this loop is as follows:

1. Divide the array B into chunks,
2. Let each thread calculate the subtotal of its assigned chunk, and
3. Calculate the array A in parallel.

The manually parallelized code is as follows:

```
      SUBROUTINE PSUB(A,B,N)
      DIMENSION A(0:N),B(N)
      INTEGER,ALLOCATABLE :: ISTA(:),IEND(:)

      NTHDS=NUM_PARTHDS()
      ALLOCATE(ISTA(0:NTHDS-1),IEND(0:NTHDS-1))
      CALL PARA_RANGE(1,N,NTHDS,ISTA,IEND)

!SMP$ PARALLEL DO PRIVATE(S)
      DO ID=0,NTHDS-1            ! Parallelized
        S=0.0
        DO I=ISTA(ID),IEND(ID)
          S=S+B(I)
        ENDDO
        A(IEND(ID))=S
      ENDDO
      DO ID=0,NTHDS-1            ! Serial
        A(IEND(ID))=A(IEND(ID))+A(ISTA(ID)-1)
      ENDDO
!SMP$ PARALLEL DO
      DO ID=0,NTHDS-1            ! Parallelized
        DO I=ISTA(ID),IEND(ID)-1
          A(I)=A(I-1)+B(I)
      ENDDO
      ENDDO
      END
```

The existence of a subroutine `PARA_RANGE` is assumed, which, in this case, assigns integers from 1 to N to `NTHDS` threads in block scheduling fashion and stores the initial and the final values of each chunk to arrays `ISTA` and `IEND`. Roughly speaking, the running time in the unit of addition is N for serial and is $p+2N/p$ for parallel where p is the number of threads.

4.8 XLSMPOPTS Environment Variable

The XLSMPOPTS environment variable specifies run-time options related to parallel execution. Section 4.1, “How to Compile, Link, and Execute” on page 29 describes the parthds option, which specifies the number of threads to be used for parallel execution of the code, and its default value is the number of

on-line processors. For instance, if you want to execute your program using four threads, set this environment variable as follows:

```
$ export XLSMPOPTS=parthds=4
```

Section 4.6.4, “SCHEDULE Compiler Directive” on page 54 also mentions the schedule option, which takes one of the following forms:

```
$ export XLSMPOPTS=schedule=affinity[=n]
$ export XLSMPOPTS=schedule=dynamic[=n]
$ export XLSMPOPTS=schedule=guided[=n]
$ export XLSMPOPTS=schedule=static[=n]
```

When you need multiple options, separate each option by a colon:

```
$ export XLSMPOPTS="parthds=4:schedule=static"
```

In addition, there are three options (spins, yields, and delays) that control busy-wait and sleep states of XL Fortran run-time library routines. In execution, each thread tries to look for its work in the following steps:

1. Scan the work queue up to spins number of times. If no work is found in a scan, then loop delays number of times before starting a new scan.
2. If work has not been found, then yield the current time slice.
3. Repeat the above steps up to yields number of times.
4. If no work has been found, then go to sleep.

The syntax for specifying these options is as follows.

- spins=*n* where *n* is the number of spins before a yield (default: spins=100)
- delays=*n* where *n* is the number of delays while busy-waiting (default: delays=500)
- yields=*n* where *n* is the number of yields before a sleep (default: yields=10)

By setting spins=0 and yields=0, you can force complete busy-waiting, sacrificing other processes’ CPU time. Normally in a benchmark test on a dedicated system, both of these options would be zero, but note that complete busy-waiting does not always improve the performance.

4.9 OpenMP Porting Considerations

The OpenMP initiative was launched in 1997 in order to provide a simple and flexible application program interface (API) for developing portable multi-platform shared-memory parallel applications on UNIX platforms and

Microsoft Windows NT architectures. At the time of writing, the OpenMP Fortran API specification Version 1.0 is available. A C/C++ API is under development. For further details refer to URL `http://www.openmp.org`. IBM is part of a multi-company initiative supporting this standard.

The current release 5.1.1.0 of XL Fortran provides a subset of the OpenMP. A complete OpenMP implementation is expected with future releases of XLF, driven by the ASCI project. XLF also offers an interface to the pthread library, which does help to cover certain OpenMP function when porting codes to the RS/6000 platform. This section will give some hints concerning differences between XLF and OpenMP and the usage of the pthread library module in this context.

In particular, XLF has partial support for the CRITICAL, END CRITICAL, PARALLEL DO, PARALLEL SECTIONS, SECTION, and END PARALLEL SECTIONS directives. To ensure the greatest portability of code, it is recommended to use these directives whenever possible. These directives should be used with the OpenMP directive sentinel !$OMP. SMP directives are recognized by the compiler if either xlf_r or xlf90_r is used and the -qsmp option is specified.

XLF does not recognize the OpenMP conditional compilation, for example, triggered by the directive sentinel !$. Nor does XLF define the C preprocessor macro _OPENMP to be used for conditional compilation (see #ifdef _OPENMP). If appropriate _OPENMP can be defined through the compiler command line flag -WF,-D_OPENMP.

XLF does not provide the OpenMP END PARALLEL DO directive. This is a minor difference since the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO in OpenMP as well.

For explicit process synchronization, XLF relies on CRITICAL and END CRITICAL directives. Besides, there is an implied barrier at the end of a parallel region, since only the master thread continues execution. The PARALLEL DO and PARALLEL SECTIONS directives are shortcuts of the OpenMP PARALLEL REGION construct, which is not available in XLF. In particular, no OpenMP BARRIER directive is available. The BARRIER directive can be substituted by a common pthread construct, as shown in the example program at the end of this section.

The XLF THREADLOCAL directive makes named common blocks private to a thread but global within a thread. It is a possible method of ensuring that access to data contained within COMMON blocks is serialized. Threads can be created in one of the following ways: explicitly through pthread library calls

or implicitly by the compiler for parallel loop or parallel section execution. The THREADLOCAL directive does not require the -qsmp compiler option.

The semantics of the XLF THREADLOCAL directive slightly differs from the OpenMP THREADPRIVATE directive. The THREADLOCAL attribute is not allowed in a pure subprogram. Members of a THREADLOCAL common block must not appear in NAMELIST statements. A common block that is use-associative must not be declared as THREADLOCAL in the scoping unit that contains the USE statement. A THREADLOCAL common block may have the SAVE attribute. In OpenMP, the data in THREADPRIVATE common blocks is guaranteed to persist only if the OpenMP dynamic thread mechanism has been disabled and if the number of threads is the same for all parallel regions.

For clarification, objects within THREADLOCAL common blocks may be used in parallel loops and parallel sections. However, these objects are implicitly shared across the iteration of the loop and across code blocks within parallel sections. In other words, within a scoping unit, all accessible common blocks, whether declared as THREADLOCAL or not, have the SHARED attribute within parallel loops and sections in that scoping unit.

XLF 5.1.1.0 does not support the OpenMP Execution Environment Routines, such as OMP_SET_NUM_THREADS(), OMP_GET_NUM_PROCS(), OMP_SET_DYNAMIC(), nor the OpenMP Lock Routines, such as OMP_SET_LOCK(). As a substitute, the programmer can use the XLF intrinsic functions NUM_PARTHDS() and NUM_USRTHDS() to inquire the run-time environment, and the pthread mutex constructs to create and destroy locks.

The function NUM_PARTHDS() returns the number of parallel Fortran threads the run time should create during execution of a program. This value is set by using XLSMPOPTS PARTHDS run-time option. If not set the run-time environment will return the number of processors on the machine, or, if specified, the value of the run-time option USRTHDS. The function NUM_USRTHDS() returns the number of threads that will be explicitly created by the user during execution of a program. This value is set by using the XLSMPOPTS USRTHDS run-time option. The default value is 0. To be noticed, the compiler option -qsmp has to be specified, otherwise NUM_PARTHDS() will always return a value of 1.

The following simple example shows how to use the pthread Fortran90 module to apply locks and barriers. The program was written for demonstration purposes only. It was not intended to present the most efficient

or complete implementation of a barrier. Indeed this version assumes a constant number of parallel threads. For briefness, return codes are ignored.

As shown in the example, it is straight forward to mix SMP compiler directives and pthread calls. In this case, the parallel threads are created implicitly by the PARALLEL DO directive. At the beginning of the program, mutex objects and condition variables have to be initialized. A lock is set though the function call f_pthread_mutex_lock(mutex_name), similar to the OpenMP OMP_SET_LOCK(var_name) subroutine call. The barrier is implemented as a Fortran module. The subroutine fpth_barrier_set() uses a typical pthread construct to build up a barrier.

The execution overhead to set up a barrier through the pthread interface increases with the number of threads. It amounts to less than 33 micro seconds compared to less than 8 micro seconds for a f_pthread_mutex_lock() call or less than 12 micro seconds for a CRITICAL directive, as found at least for eight or less parallel threads on a two processor machine, using the defaults for the XLSMP run-time variables spins, yields, and delays.

```
      program hello_omp

      use f_pthread
      use fpth_barrier

      implicit none

      integer, parameter      :: maxtask=20

      integer                 :: ntask, sg
      type(f_pthread_mutex_t) :: lock_mutex
      integer                 :: itask(maxtask), it, nb, rc

      common /global/ sg, lock_mutex


!     --- init
      lock_mutex = pthread_mutex_initializer
      call fpth_barrier_init()
      ntask = num_parthds()
      sg = 0

      nb = 10

!     --- parallel threads
!$OMP PARALLEL DO
      do it=1,ntask
        itask(it) = it-1
        call sub(nb, itask(it))
      end do

!     --- clean up
      rc = f_pthread_mutex_destroy(lock_mutex)
      call fpth_barrier_destroy()

      end program hello_omp
```

```
! ************************************************************************

      subroutine sub(nb, itask)

      use f_pthread
      use fpth_barrier

      implicit none

      integer                 :: nb, itask

      integer                 :: sg
      integer                 :: s, i, istart, iend
      type(f_pthread_mutex_t) :: lock_mutex
      integer                 :: rc

      common /global/    sg, lock_mutex
      common /local/     s

!IBM* THREADLOCAL /local/


!$OMP CRITICAL(crit0)
      print *, 'task ', itask, 'starting ... '
!$OMP END CRITICAL(crit0)

!     --- compute partial sum
      istart = 1+itask*nb
      iend   = (itask+1)*nb

      s = 0
      do i = istart, iend
        s = s + i
      end do

!     --- update global sum
      rc = f_pthread_mutex_lock(lock_mutex)
      sg = sg + s
      rc = f_pthread_mutex_unlock(lock_mutex)

!     --- wait until all tasks are finished
      call fpth_barrier_set()
      if (itask .eq. 0) then
         print *, 'Sum is ', sg
      end if

      end subroutine sub

! ************************************************************************

      module fpth_barrier

      use f_pthread

! --- global vars ------------------------------------------

      type(f_pthread_mutex_t) :: barrier_mutex
      type(f_pthread_cond_t)  :: barrier_cond

      integer                 :: taskcounter
      integer                 :: numtasks
```

```
      contains


! --- init barrier -----------------------------------------

      subroutine fpth_barrier_init()

      use f_pthread

      implicit none


      taskcounter = 0
      numtasks    = num_parthds()

      barrier_mutex = pthread_mutex_initializer
      barrier_cond  = pthread_cond_initializer

      end subroutine fpth_barrier_init


! --- set barrier ------------------------------------------

      subroutine fpth_barrier_set()

      use f_pthread

      implicit none
      integer :: rc


      rc = f_pthread_mutex_lock(barrier_mutex)

      taskcounter = taskcounter + 1
      if (taskcounter .eq. numtasks) then
         taskcounter = 0
         rc = f_pthread_cond_broadcast(barrier_cond)
      else if (taskcounter .lt. numtasks) then
         rc = f_pthread_cond_wait(barrier_cond, barrier_mutex)
      end if

      rc = f_pthread_mutex_unlock(barrier_mutex)

      end subroutine fpth_barrier_set


! --- destroy barrier --------------------------------------
      subroutine fpth_barrier_destroy()

      use f_pthread

      implicit none
      integer :: rc


      rc = f_pthread_mutex_destroy(barrier_mutex)
      rc = f_pthread_cond_destroy(barrier_cond)

      end subroutine fpth_barrier_destroy

      end module fpth_barrier
```

Chapter 5. Performance Libraries

There are several challenges to write programs that perform well on all machines, since different architectures require different tuning techniques.

One solution is to have a unique program version for each architecture it is intended to run on. This will, in general, increase the complexity of the code as well as the complexity of the development. But even if you manage to maintain only one version, the tuning itself tends to increase the complexity of your program. As a more complex program increases the effort to maintain the code, the development costs will also increase.

Therefore, commercial programs tend to be unoptimized. They will have a few general optimization techniques implemented, but this will not give them the performance they can expect, especially from a new processor like the POWER3.

One way to solve this problem and get performance across different architectures is to use standard libraries that are specifically tuned for each platform.

This chapter describes two libraries, ESSL and MASS, that increase the performance on all platforms they are used on, without losing portability. The MASS library is a replacement of some FORTRAN intrinsic like EXP(). ESSL provides significantly higher functions, such as linear algebra and FFTs.

5.1 The ESSL Library

The family of Engineering and Scientific Subroutine Library (ESSL) is a collection of highly tuned routines you can use in your program. The ESSL family for the AIX operating system consists of:

- Parallel Engineering and Scientific Subroutine Library (Parallel ESSL) for Advanced Interactive Executive (AIX), program number 5765-C41
- Engineering and Scientific Subroutine Library (ESSL) for AIX, program number 5765-C42

These products are state-of-the-art collections of mathematical subroutines that provide a wide range of functions for many different scientific and engineering applications.

Parallel ESSL runs under the IBM RS/6000 SP and clusters of IBM RS/6000 workstations. It offers mathematical subroutines in the six computational areas and has one extra area for utilities, namely:

- **Level 2 Parallel Basic Linear Algebra Subprograms (PBLAS)**

  Level 2 PBLAS include a subset of the standard set of distributed memory parallel versions of the Level 2 Basic Linear Algebra Subprograms (BLAS). The Level 2 subroutines of BLAS perform vector-matrix operation.

- **Level 3 PBLAS**

  Level 3 PBLAS include a subset of the standard set of distributed memory parallel versions of the Level 3 BLAS. The Level 3 subroutines of the BLAS subroutines perform matrix-matrix operations.

- **Linear Algebraic Equations**

  Linear Algebraic Equations Subroutines consist of dense, banded, and sparse subroutine, and include a subset of the ScaLAPACK subroutines.The ScaLAPACK library can be found at:

  `http://www.netlib.org/scalapack/`

  The routines in PESSL includes:

  - Dense Linear Algebraic Equations Subroutines provide solutions to linear systems of equations for real and complex general matrices and their transposes, and for positive definite real symmetric and complex Hermitian matrices.
  - Banded Linear Algebraic Equations Subroutines provide solutions to linear systems of equations for real positive definite symmetric band matrices, real general tridiagonal matrices, diagonally-dominant real general tridiagonal matrices, and real positive definite tridiagonal matrices.
  - Sparse Linear Algebraic Equations Subroutines and their utility subroutines provide iterative solutions to linear systems of equations for real general sparse matrices.

- **Eigensystem Analysis and Singular Value Analysis**

  Eigensystem Analysis and Singular Value Analysis Subroutines provide solutions to the algebraic eigensystem analysis problem for real symmetric matrices and the ability to reduce real symmetric and real general matrices to condensed form. These subroutines include a subset of the ScaLAPACK subroutines.

- **Fourier Transforms**

  Fourier Transform Subroutines perform mixed-radix transforms in two and three dimensions.

- **Random Number Generation**

  Random Number Generation Subroutine generates uniformly distributed random numbers.

- **Utilities**

  Utility Subroutines perform general service functions, rather than mathematical computations.

ESSL runs on the following platforms:

- POWER3
- IBM RS/6000 POWER, PowerPC, symmetric multiprocessing (SMP) PowerPC, and POWER2 Processors
- IBM RS/6000 SP

It offers mathematical subroutines in nine computational areas and on an extra utility area:

- **Linear Algebra Subprograms**

  Linear Algebra Subprograms consist of vector-scalar, sparse vector-scalar, matrix-vector, and sparse matrix-vector linear algebra subprograms:

  - Vector-Scalar Linear Algebra Subprograms include a subset of the standard set of Level 1 BLAS and subroutines for other commonly used computations. Both real and complex versions of the subprograms are provided.
  - Sparse Vector-Scalar Linear Algebra Subprograms operate on sparse vectors; only the nonzero elements of the vectors need to be stored. These subprograms provide functions similar to those of the vector-scalar subprograms and represent a subset of the sparse extensions to the Level 1 BLAS. Both real and complex versions of the subprograms are provided.
  - Matrix-Vector Linear Algebra Subprograms operate on a higher-level data structure, matrix-vector rather than vector-scalar, using optimized algorithms to improve performance. These subprograms represent a subset of the Level 2 BLAS. Both real and complex versions of the subprograms are provided.

- Sparse Matrix-Vector Linear Algebra Subprograms operate on sparse matrices; only the nonzero elements of the matrix need to be stored. These subprograms provide functions similar to those of the matrix-vector subprograms.

- **Matrix Operations**

  Matrix Operations Subroutines include Level 3 BLAS, as well as the commonly used matrix operations: addition, subtraction, multiplication, and transposition.

- **Linear Algebraic Equations**

  Linear Algebraic Equations Subroutines consist of dense, banded, sparse, and linear least squares subroutines:

  - Dense Linear Algebraic Equations Subroutines provide solutions to linear systems of equations for real and complex general matrices and their transposes, positive definite real symmetric and complex Hermitian matrices, and triangular matrices. Some of these subroutines correspond to the Level 2 and Level 3 BLAS.
  - Banded Linear Algebraic Equations Subroutines provide solutions to linear systems of equations for real general band matrices, real positive definite symmetric band matrices, real or complex general tridiagonal matrices, real positive definite symmetric tridiagonal matrices, and real or complex triangular band matrices.
  - Sparse Linear Algebraic Equations Subroutines provide direct and iterative solutions to linear systems of equations, both for general sparse matrices and their transposes and for sparse symmetric matrices.
  - Linear Least Squares Subroutines provide least squares solutions to linear systems of equations for real general matrices. Two methods are provided: one with a singular value decomposition and another with a QR decomposition with column pivoting.

- **Eigensystem Analysis**

  Eigensystem Analysis Subroutines provide solutions to the algebraic eigensystem analysis problem Az = wz and the generalized eigensystem analysis problem Az = wBz. These subroutines give you several options for computing eigenvalues or eigenvalues and eigenvectors.

- **Fourier Transforms, Convolutions and, Correlations:**

  - Fourier Transform Subroutines perform mixed-radix transforms in one, two, and three dimensions.

- Convolution and Correlation Subroutines offer a choice between Fourier methods or direct methods. The Fourier-method subroutines contain a high-performance mixed-radix capability. Also, several direct-method subroutines provide decimated output.

- **Sorting and Searching**

  Sorting and Searching Subroutines operate on three types of data: integer, short-precision real, and long-precision real. The sorting subroutines perform a sort with or without index designations. The searching subroutines perform either a binary or a sequential search.

- **Interpolation**

  Interpolation Subroutines provide capabilities for polynomial interpolation, local polynomial interpolation, and both one- and two-dimensional cubic spline interpolation.

- **Numerical Quadrature**

  Numerical Quadrature Subroutines provide one-dimensional methods for integrating a tabulated function and a user-supplied function over a finite, semi-infinite, or infinite region of integration by Gaussian quadrature methods. They also provide a two-dimensional quadrature capability within a rectangular boundary.

- **Random Number Generation**

  Random Number Generation Subroutines generate uniformly or normally distributed random numbers.

- **Utilities**

  Utility Subroutines perform general service functions, rather than mathematical computations.

Several versions of most subroutines are provided, depending on the type of data you are processing. These may include a short- and long-precision real version, a short- and long-precision complex version, and an integer version.

The following Web pages contains more information on ESSL and PESSL:

`http://www.rs6000.ibm.com/software/Apps/essl.html`
`http://www.rs6000.ibm.com/software/sp_products/esslpara.html`

5.1.1 Benefits of Using ESSL

The main benefits include:

- Portability

Since there exists an ESSL for each RS/6000 machine, you can move the code between different machines and different architectures, without changing the source code. It is also compatible with public domain subroutine libraries such as BLAS, Scalable Linear Algebra Package (ScaLAPACK), PBLAS, making it easy to migrate from these libraries to an ESSL product.

- Performance

  The ESSL routines are written to perform well on each RS/6000 architecture. There is also a SMP version of the ESSL, in which a subset of the functions are thread enabled. By using this version of ESSL, your code would take advantage of all SMP features without any new development.

5.1.2 How to Use ESSL

For access to the Guide and Reference see the following Web page:

`http://www.rs6000.ibm.com/resource/aix_resource/sp_books/`

Porting Fortran between CRAY and the IBM RISC System/6000:

`http://www.software.ibm.com/ad/fortran/xlfortran/cray.htm`

5.1.3 Performance Examples of ESSL

This section will discuss the performance of some of the ESSL routines. The official *ESSL and PESSL Performance Report* can be found on:

`http://www.rs6000.ibm.com/software/sp_products/performance/pesslperf.html`

Notice

The ESSL used in this publication is an early beta of a POWER3-enhanced library, please refer to Appendix D, “Special Notices” on page 199 regarding the performance numbers.

5.1.3.1 Dcopy

The following three approaches for copying the double precision array A into array B are compared:

Table 9. Four Different dcopy Approaches

Simple	Prefetch	ESSL	C memcpy
`B(I)=A(I)`	`B(I)=A(I)+B(I)*ZERO`	`CALL DCOPY()`	`CALL memcpy()`

How the prefetch works is shown in Figure 2., “Data Prefetch Overview” on page 11.

In Figure 6, the performance of these copy routines using one POWER3 CPU are shown. The numbers shown are the best out of three runs and both the L1 as the L2 cache are flushed between each measurement. The most efficient one is the ESSL. The simple approach from above is the second slowest, but surprisingly, the slowest one is the C memcpy() function. By unrolling the copy routine 16 times and multiplying only the first element with zero, almost the same performance as the ESSL library is obtained. The main difference is the drop of performance at 700000 KB. This event is currently being investigated.

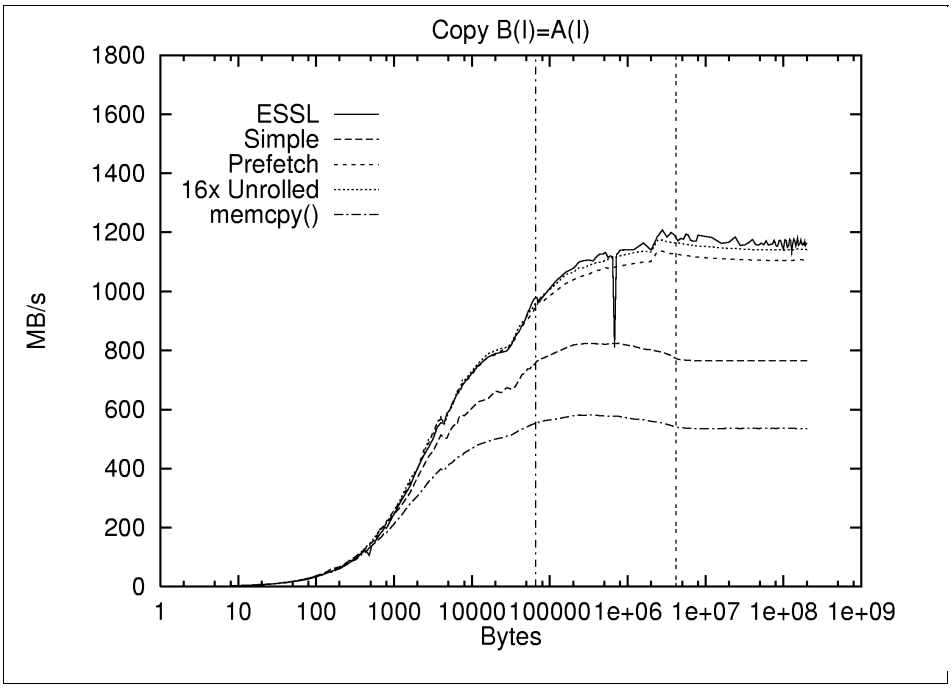


Figure 6. Copy Rates of a Double Precision Array

5.1.3.2 DAXPY

The three DAXPY versions shown in Table 10 are compared.

Table 10. Three DAXPY Versions:

Simple	4x Unrolled	ESSL

`Y(I)=Y(I)+A*X(I)`	`Y(I)=Y(I)+A*X(I)`
	`Y(I+1)=Y(I+1)+A*X(I+1)`
	`Y(I+1)=Y(I+2)+A*X(I+2)`
	`Y(I+1)=Y(I+3)+A*X(I+3)`
		`CALL DAXPY()`

Only the best run out of three is used, and the caches are flushed between each run. As can be seen in Figure 7, the ESSL version is slower than the handwritten versions for a vector length up until 5300. Above 5300, it is the fastest one. This is an effect of the overhead in the ESSL routine, as it checks the input arguments.

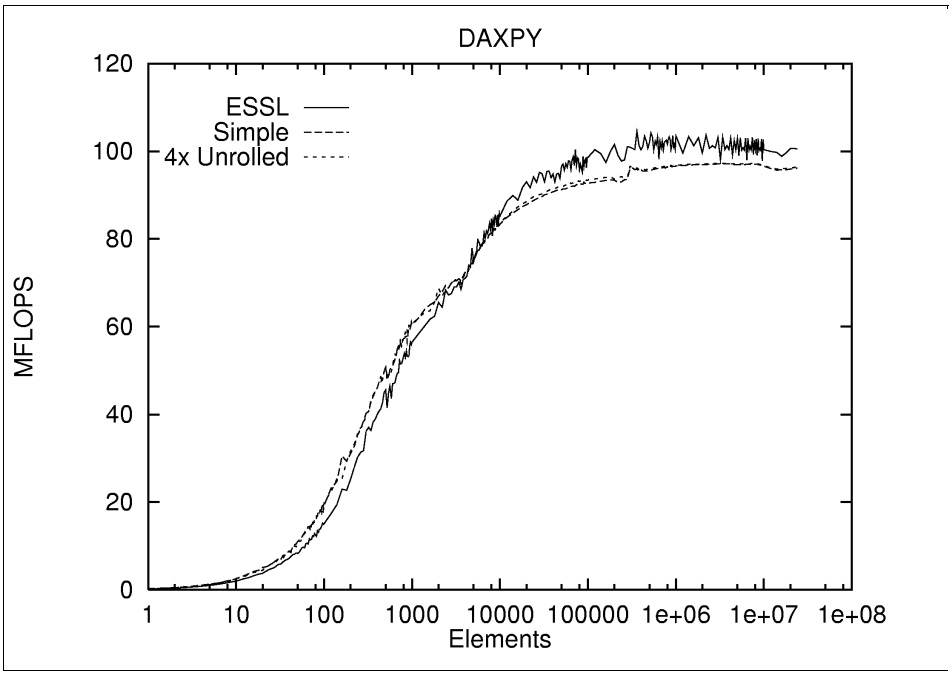


Figure 7. DAXPY Comparison

5.1.3.3 DGEMM

For an example and performance numbers using DGEMM, consult section 9.3, “Case Study: Matrix Multiplication” on page 151.

5.1.3.4 Sorting of an Array

One way to optimize your program is to change the algorithm you are using. Consider the following example of sorting an array. The simplest way of doing it is by a *bubble sort.* Another algorithm is *quick sort*, which XL Fortran provides a version of. ESSL has several sort algorithms, dsort was selected here. In Figure 8, the three algorithms are compared. The timings are the best out of three runs, and the caches are flushed between each run. All timings over 30 seconds were excluded. As can be seen in Figure 8, the bubble sort is only good for a few thousand values. The ESSL dsort routine is faster than the qsort provided by XL Fortran.

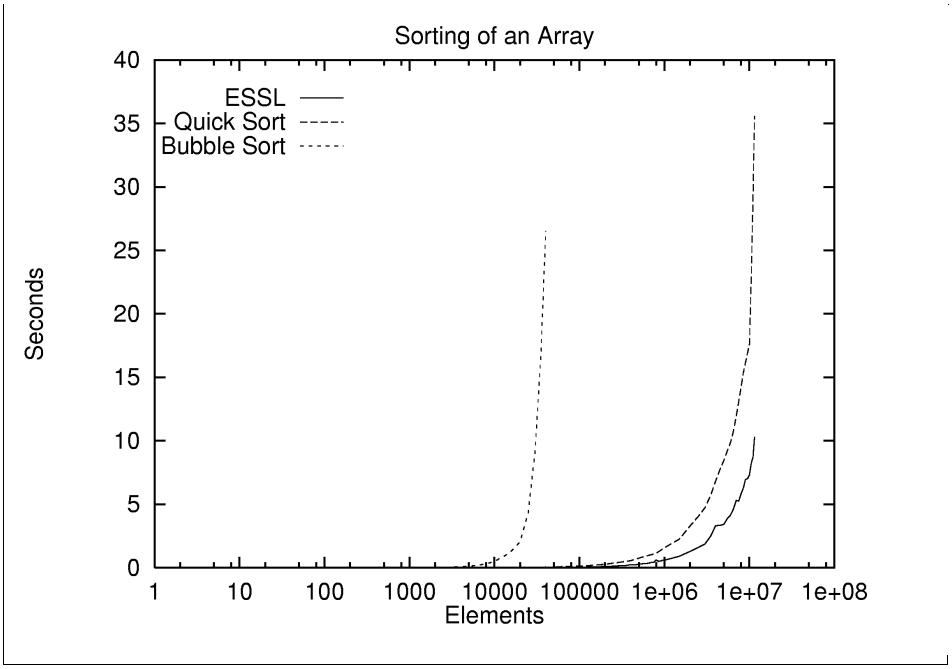


Figure 8. Three Sorting Algorithms

5.2 MASS

The Mathematical Acceleration SubSystem (MASS) library is another approach to increase the performance of a code. It provides high performance versions of a subset of Fortran intrinsic functions. To do this, it sacrifices a small amount of accuracy. Compared to the standard mathematical library, libm.a, the MASS library can only differ in the last bit. This is not significant in most programs. The libmass.a library can be used

with either Fortran or C applications and will run under AIX on all of the IBM RS/6000 processors. As all functions in the MASS library use the same syntax as the standard functions it replaces, you do not have to make any changes in the source code to use it.

MASS also offers a *vector* version for some of the functions. The vector functions are more efficient than the scaler ones, but require that the source code is rewritten. There are two versions of the vector MASS library. The first library, libmassv.a, contains vector function subroutines that will run on the entire IBM RS/6000 family. The second library, libmassvp2.a, contains the subroutines of libmassv.a and adds a set that is tuned for and based upon the POWER2 architecture. As code or a library that is compiled using the -qarch=pwr2 flag will not run under POWER3, you cannot use this library on Model 260. At the moment, there is no specific tuned version for the POWER3.

All versions of the MASS library can be downloaded from:
`http://www.rs6000.ibm.com/resource/technology/MASS/`

Version 2.4 is used for all tests in this section. This version is not thread safe.

The accuracy of the functions in the MASS library can be found on the MASS Web page mentioned above.

5.2.1 How to Use the MASS Library

To use the standard MASS library, relink your program using the linker option `-lmass`:

```
xlf -o my.exe -O3 -qarch=pwr3 -lmass
```

This assumes that the MASS library is in a directory included in your library search path. If this is not the case, you have to give the location of the library with the -L linker option. As -lmass replaces some of the function in -lm, you must link it *before* you link with -lm.

If the use of standard MASS is successful, the chance to further increase the performance using the vector version of MASS is high. Please note that as you frequently have to include extra arrays in your code, there will be more memory operations to fill compared to the original version of your code. These extra operations could decrease the overall performance even if the calculation is done faster.

In order to guarantee the portability of a code using the vector MASS library, the MASS package also provides Fortran source for all vector MASS library functions which can be used on different platforms.

5.2.2 Performance of the MASS Library

Take Note

The MASS library used was not tuned for the POWER3 processor. The generic version of the MASS library is used for measurements in this section.

In order to see the performance gain of using the MASS libraries, the number cycles used to perform some often used mathematical functions were counted. The results are listed in Table 11.

In general, there is a speedup factor between 1.2 and a little over 2 by going from the standard math library to the scalar MASS library and another factor between 2 and 5 by going to the vector MASS library. There are two important exceptions:

1. The POWER3 has the square root function implemented in the hardware of the CPU. When compiling for POWER3 using the compiler option -qarch=pwr3, the compiler will not generate a call to the sqrt() library function but use the hardware instead. Therefore, there is no difference in the cycle count comparing the standard math library with the MASS library. Figure 2 on page 11 shows the number of cycles for a hardware sqrt to be 22. As each POWER3 processor has two FPU, it can calculate two sqrts simultaneously. The Fortran compiler generates codes, which dispatches the sqrt calls very well; so only 11 cycles cost per sqrt call are required.

   When compiling with -qarch=com, which disables the hardware sqrt , a speedup from about 20percent is obtained, by using MASS compared to libm. The vector MASS is almost six times faster than the libm.

2. Table 2 on page 9 also shows that the number of cycles needed for a double precision division is 18. Again, as the POWER3 has 2 FPUs, two divisions can be performed in parallel and the compiler does a very good job in dispatching them to get only 9 cycles cost per division, but the

MASS library is able to speed up it slightly. Note that this could be improved if a POWER3 optimized library is developed.

Table 11. Cycles of Some Functions

Function [1]	R [2]	libm.a	MASS		Vector MASS	
		Cycles	Cycles	Speedup	Cycles	Speedup
X(I)=SQRT(A(I))	A	11.0	11.0	1.0	9.4	1.2
X(I)=SQRT(A(I)) [3]	A	58.9	45.4	1.3	9.4	6.3
X(I)=EXP(A(I))	D	64.3	33.3	1.9	11.0	5.8
X(I)=LOG(A(I))	C	83.0	53.4	1.6	11.5	7.2
X(I)=SIN(A(I))	B	37.7	15.7	2.4	6.6	5.7
X(I)=SIN(A(I))	D	50.0	31.5	1.6	16.4	1.9
X(I)=COS(A(I))	B	37.2	15.7	2.4	5.8	6.4
X(I)=COS(A(I))	D	48.3	32.7	1.5	16.3	3.0
X(I)=TAN(A(I))	D	84.1	50.1	1.7	18.4	4.6
X(I)=TAN(A(I))	D	80.8	50.3	1.6	18.4	4.4
X(I)=A(I)/B(I)	B,D [4]	9.2	9.2	1.0	7.1	1.3
X(I)=1.0/A(I)	D	9.0	9.0	1.0	7.0	1.3

Remarks:

1. The arrays A and X each have 1024 elements.
2. R describes the range the value of input arguments can take:
   - A: 0<A(i)<1
   - B: -1<A(i)<1
   - C: 0<A(i)100
   - D: -100<A(i)<100.
3. Compiled with -qarch=com.
4. A is in range B, B is in range D.

The cycle numbers show the performance gain by a given vector length. But you will get a different speedup for different vector length. As an example consider the following simple loop using the exponential function:

```
DO I=1,N
 A(I)=EXP(B(I)
END DO
```

with A and B are double precision arrays. Compare this loop using the standard exp() function from libm, the standard MASS exp() function, and with the vector MASS function, You get by changing the loop into a function call:

```
CALL EXPV(A,B,N)
```

The speedup gained is seen in Figure 9 on page 77. The two horizontal lines mark the positions the arrays size exceeds the size of the L1 and L2 cache. The time for the standard version of the exp() function is normalized to one. The middle curve is the speedup for the standard MASS library. The speedup is around 1.8 for large N's. The upper curve is the speedup gained by using the vector mass library. It has a peak of 5.9 at a vector length of 3000 elements. The speedup for very large values of N approximates 4.97.

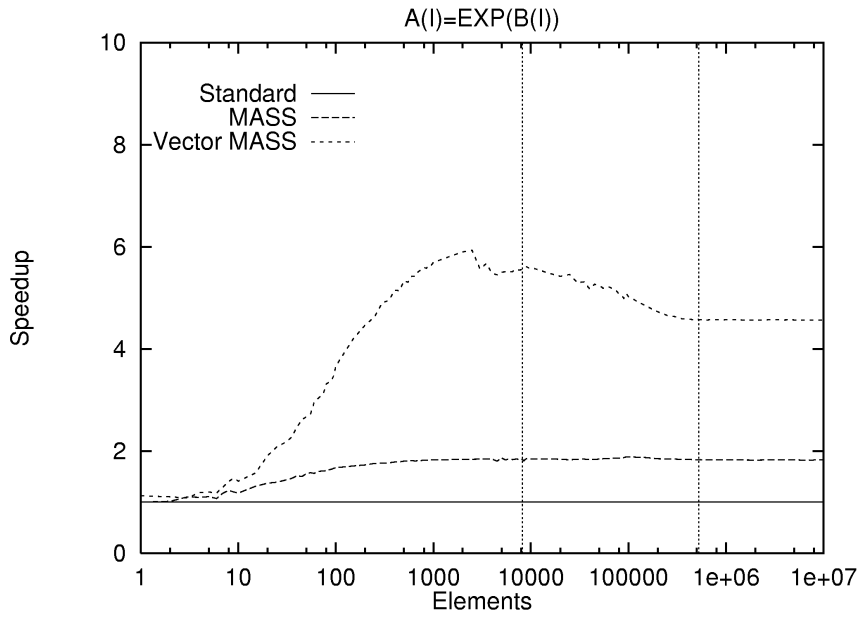


Figure 9. MASS Use of Exp( )

5.2.3 Further Tuning Possibilities Using Vector MASS

As some functions provided by the standard math library can be rewritten using a different algorithm which benefits from the vector MASS library, it is worthwhile to demonstrate the following.

5.2.3.1 The Complex Exp() Function

The standard Fortran exp() function can be used in conjunction with complex numbers. You would simply write (where X and Y are complex numbers):

```
DO I=1,N
    X(I)=EXP(Y(I))
   END DO  (ex.1),
```

By splitting y(j) into its real and imaginary parts a(j) and b(j) with y(j)=a(j)+ib(j) and using Euler's equation, you get the following formula for the complex exp() function:

$$x_i = \exp(a_i + ib_i) = \exp a_i \cdot \exp ia_i = \exp a_i \cdot (\cos b_i + i\sin b_i)$$

By coding this into Fortran, you get:

```
X(I)=CMPLX(
   EXP(REAL(A(I)))*COS(IMAG(A(I))),
   EXP(REAL(A(I)))*SIN(IMAG(A(I)))) (ex.2)
```

In order to use the vector MASS library, you must rewrite it one more time using the vector MASS subroutine vsincos. This subroutine calculates both the sin and the cosine value of one argument in one call:

```
   DO I=1,N
    RA(I) = REAL(A(I))
    IA(I) = IMAG(A(I))
   END DO
   CALL VEXP(RA, REA, N)
   CALL VSINCOS(ISA, ICA, IA, N)
   DO I=1,N
    X(I) = CMPLX(REA(i)*ICA(i), REA(i)*ISA(i))
   END DO  (ex.3)
```

This is an example in which you have to introduce more arrays in order to use the vector library. Please note that this code fragment is written to see the relationship to the Euler's equation. It is, simply, to reduce the numbers of arrays needed.

Examples 1,2, and 3 are programmed, and the number of cycles needed to solve the problem are counted. As Table 12 shows, the results are very good. By using the vector MASS, you could speedup the calculation by a factor between 5.2 and 5.9 depending on the vector length.

Table 12. Complex Exponential Function

Ex.	using	n=16		n=64		n=256		n=1024	
1	libm.a	179.0	1.0	171.0	1.0	169.5	1.0	168.5	1.0
1	MASS	85.2	2.1	84.8	2.0	84.7	2.0	84.7	2.0
2	MASS	75.0	2.4	74.7	2.3	74.4	2.3	74.5	2.3
3	vector	34.7	5.2	30.3	5.6	30.0	5.7	28.8	5.9

5.2.3.2 The Power Function Using The Vector MASS Library

Another frequently used function is the power function x(i)=a(i)**q (ex.1). The power function is one of the most expensive intrinsic. Replace the standard Fortran power function with the following equation:

$$x_i = y_i^q = \exp(q \cdot \log y_i)$$

This can be written in Fortran:

```
DO I=1,N
 X(I)=EXP(Q*LOG(Y(I)))
END DO    (ex.2)
```

In this example, you don't have to introduce an extra array in order to use the vector MASS, since you can use X for temporary values:

```
CALL VLOG(X,Y,N)
DO I=1,N
 X(I)=X(I)*Q
END DO
CALL VEXP(X,X,N) (ex.3)
```

As can be seen in the Table 13, you get a speedup factor close to ten by using the vector MASS library compared to the standard power function.

Table 13. Power Function

Ex.	using	n=16		n=64		n=256		n=1024	
1	libm.a	228.5	1.0	227.8	1.0	224.0	1.0	224.8	1.0
2	libm	152.4	1.5	150.8	1.5	150.1	1.5	150.4	1.5
1	MASS	98.2	2.3	97.9	2.3	97.8	2.3	97.7	2.3
2	MASS	90.6	2.5	90.5	2.5	90.4	2.5	90.4	2.5
3	vector	29.1	7.9	24.7	9.2	23.7	9.5	23.4	9.6

Chapter 6. Message Passing Interface

A large number of programs have already been written for parallel processing using MPI. Many of these programs will be running on the single processor nodes of IBM SP configurations. These programs can be run without change on the two processors of the Model 260 simply by using MPI on the 260.

The following sections consider the use of MPI in a mixed shared memory and distributed memory environment, followed by some measurements of MPI data transmission rates on the Model 260.

6.1 MPI in an SMP Environment

This section takes a look at how existing MPI programs, written for distributed memory systems, can make the best use of both SMP and distributed memory systems. A number of different scenarios are considered below:

1. **MPI only**

   IBM's MPI currently uses IP for message passing between processes on the same node and between processes on different nodes. This incurs relatively high latencies and IP overheads.

   With the multiple userspace version of MPI, the overhead will be reduced, but it may still be higher between processes on the same node than using shared memory.

   Eventually, it is expected that a version of MPI will be available that will use shared memory for processes on the same node and userspace (or IP) for processes on different nodes. However, it is expected that overall performance will still be limited by communication between the nodes. This could be reduced for group operations (such as broadcast) by having one processor per node handle all the internode communication. This process would use shared memory to collect and distribute data to other processes on the same node.

   Since the different processes on the same node have different address spaces they will communicate though a shared memory segment. This mean either a double copy of the data (into and out of the shared memory segment), or each process must *keep* its data in the shared memory segment (which will require some degree of reprogramming).

   For scenarios that only require SMP processing, the public domain software from Argonne (MPICH) is currently available. This uses shared memory to communicate data and has low latency and high data transfer rates.

Generally, no reprogramming is required.

2. **MPI and SMP Fortran**

   In this scenario, there are fewer MPI processes than processors per node. (For the Model 260, this means one MPI process.) Fortran can be used to parallelize the code between SMP calls. However, the overhead of Fortran parallelization is similar to that of MPI data transfers; so care must be taken to parallelize sufficiently large chunks.

   A small amount of reprogramming may be required.

3. **MPI and Large Chunk Threads**

   In this scenario, there is only one MPI process per node. The initial process (or master thread) creates threads which, instead of issuing MPI calls, use pthread techniques to transfer data between themselves and the master thread. The master thread uses MPI to transfer all data between the nodes.

   Data does not have to be copied between threads, since they all use the same address space. Synchronization can be achieved either with standard pthread calls, or, with even less overhead, by using spin loops and the atomic fetch_and_add function (which guarantees that only one thread at a time can update a variable).

   The total number of messages between nodes is reduced, and hence, delays due to latency are reduced. Since the master thread handles all messages, it should perhaps be coded to do less work than the other threads

   However, all of this may imply considerable reprogramming. The program may have used the MPI task-ID to create its arrays and organize its data. The threads will have to arrange this differently, because they share the same task-ID, and are using the same address space.

The advantages and disadvantages of these scenarios are summarized in Table 14

Table 14. Advantages and Disadvantages of Msg Passing Techniques

	Advantages	Disadvantages
MPI only	No program changes. MPI copy between processes on same node.	Double copy between processes on same node. Not all functions available yet.
MPI and SMP Fortran	MPI exchanges reduced.	Some reprogramming required. May not be possible to fully use the CPUs.

	Advantages	Disadvantages
MPI and Large Chunk Threads	MPI Exchanges reduced. Exchanges and overhead between threads reduced.	Considerable reprogramming may be required.

In conclusion, all of the scenarios are viable. The scenario chosen for any particular application will depend on the requirements. Going from the top to the bottom of the table, the efficiency of the solution increases, but the amount of reprogramming required also increases.

The last, and most efficient, scenario is the one used by the sPPM ASCI benchmark code. More information about this can be obtained from:

```
http://www.llnl.gov/asci_benchmarks/asci/limited/ppm/sppm_readme.html
```

6.2 MPI Communication Rates

Three varieties of MPI were investigated to look at the communication rates obtainable:

1. **IBM MPI**

   This was run in IP mode using loopback mode (obtained by inserting loopback in the host.list file). Various IP options were set as follows:

   ```
   /usr/sbin/no -o thewall=16384
   /usr/sbin/no -o sb_max=1310720
   /usr/sbin/no -o tcp_sendspace=327680
   /usr/sbin/no -o cp_recvspace=327680
   /usr/sbin/no -o udp_sendspace=65536
   /usr/sbin/no -o udp_recvspace=655360
   /usr/sbin/no -o rfc1323=1
   ```

   The maximum transmission unit (MTU) value for the loopback interface was 16896. Generally, a lower value is used for Ethernet connections between workstations, and a higher value for IP connections over the SP switch.

2. **MPICH**

   This is a portable implementation of MPI developed at the Argonne National Laboratory. It can run either over a cluster of workstations, using IP for communication, or on a single workstation with multiple processors, using shared memory for communication. MPICH was run using the shared memory option.

3. **Test MPI**

   For processing that involves processors on the same and different systems, it will be good to have an MPI version that uses shared memory for processors on the same workstation or SP node, but uses the network interface to communicate with other workstations or SP nodes.

   Such a version is currently under development, but in order to get an idea of the performance that might be achievable, a simple shared memory implementation of MPI blocking and non-blocking send and blocking receive (MPI_SEND, MPI_ISEND, and MPI_RECV) was written. This implementation was used to measure the performance for calls between processors on the same node.

   If these calls require communication between different nodes, the current IBM MPI can be invoked. For an IBM SP, the high performance userspace option of IBM MPI would be used.

The results for synchronous send and receive (that is, with one processor issuing a blocking send followed by a blocking receive, and the other processor issuing matching receive followed by a send) are shown in Figure 10 on page 85.

The results for asynchronous send and receive (that is, with both processors issuing a non-blocking send followed by a blocking receive) are shown in Figure 11 on page 85.

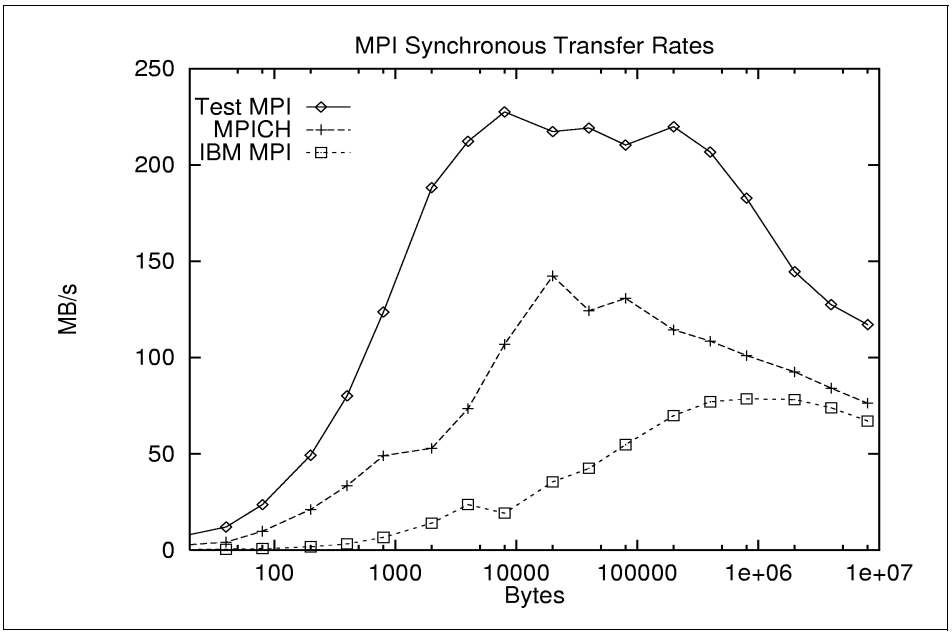


Figure 10. MPI Synchronous Transfer Rates

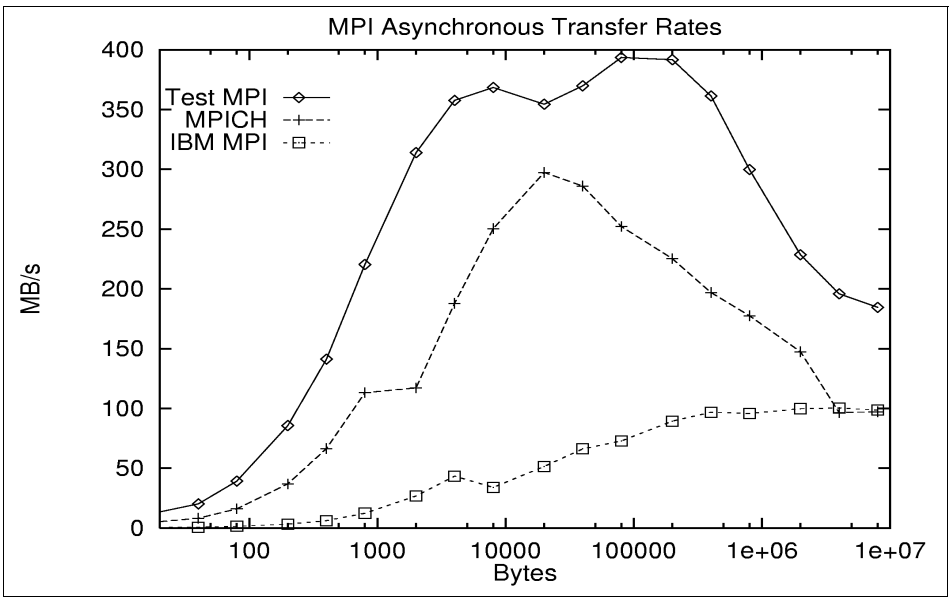


Figure 11. MPI Asynchronous Transfer Rates

Clearly, the test MPI implementation is the fastest. Since many programs issue a relatively small number of different MPI calls, it is felt that the test MPI implementation could be fairly rapidly extended to cover a subset of the most frequently used calls. This would enable SMP nodes in an SP environment to be used to their fullest advantage in the very near future.

A subset of frequently used calls might be:

```
MPI_SEND, MPI_ISEND
MPI_RECV, MPI_IRECV
MPI_BCAST, MPI_REDUCE
MPI_ALLREDUCE, MPI_ALLTOALL
```

A further enhancement might be to make one processor on each node a dedicated communicator. The dedicated communicator could collect and combine all messages from processors on the same node that went to other nodes. This would decrease the number of messages passed between nodes, thus decreasing the overall delay due to latency.

The minimum latency time and maximum transfer rates are summarized in Table 15.

Table 15. Synchronous versus Asynchronous Transfer Times

	Synchronous Transfer		**Asynchronous Transfer**	
	Latency (microsec)	**Rate (MB/s)**	**Latency (microsec)**	**Rate (MB/s)**
Test MPI	3	227	2	393
MPICH	8	142	6	297
IBM MPI	1044	78	55	100

Chapter 7. Performance and Tuning Analysis

This section investigates some of the main design features of POWER3 that are relevant to tuning scientific and technical programs running on Model 260.

It is intended to provide all the information necessary for experienced Fortran programmers, who have a general understanding of tuning techniques, to tune their programs for POWER3 and the Model 260. For programmers with relatively little experience of tuning for the IBM RS/6000 RISC, Chapter 8, “Fortran Tuning Guide for Maximum Megaflops” on page 107, will be more appropriate.

After summarizing relevant information, tuning for the CPU is discussed, and then tuning for memory access. A few basic examples are presented.

7.1 Relevant Information

The information below is relevant for tuning purposes. To some extent, it is a summary of information presented previously in Chapter 2, “The POWER3 Processor” on page 7, and Chapter 3, “XL Fortran Version 5” on page 17.

- **Floating-point units**

  There are two floating-point units. The length of the pipeline in each is three or four cycles. It will be four cycles if the input data for one pipe comes from the other pipe. Since this is difficult to influence, it is best to assume four cycles for planning purposes.

  Each unit can deliver one result per cycle.

- **Other Processing Units**
  - In addition to the two floating-point units, there are three fixed point units, and two load/store units, all of which can execute in parallel.
  - Instructions must complete in order, although results from an instruction can be used by other instructions prior to completion of the previous instruction.
  - A maximum of 32 instructions can be handled simultaneously, where *handling* includes other operations (such as instruction fetch, decode and dispatch, rename buffer allocation, and write back to architected registers) as well as execution.
  - A maximum of four instructions can be completed per cycle.

- **Data transfer rates**

  Data transfer rates between memory, cache and processor are listed in Table 16.

Table 16. Data Transfer Rates for L1, L2, and Memory

	Memory to L2 or L1	L2 to L1	L1 to Registers
Width	16 bytes/2 cycles	32 bytes/cycle	2 x 8 bytes/cycle
Rate	1.6 GB/s	6.4 GB/s	3.2 GB/s
Latency	35 cycles (approximately)	6 to 7 cycles (approximately)	1 cycle

  Note that the transfer rate from memory to L2 (or L1) is the total aggregate rate for the memory subsystem to both L2s (or L1s). The other transfer rates are for each processor.

- **Prefetch**

  The Model 260 can prefetch streams of data from memory into L1 cache. When data is prefetched, it is not put into the L2 cache.

  Four prefetch streams can be active at any one time. Activation of a stream is performed using a set of ten stream address filters. Following a load-cache-miss for a cache line, a prediction is made of which line will be required next. This prediction is entered into the least recently used filter. If a subsequent cache-miss actually agrees with one of the ten entries, then one of the four prefetch streams is activated, and the next cache line is read into a prefetch buffer.

  For an active prefetch-stream, a new line is read into the prefetch buffer as soon as the prefetch buffer is accessed.

  The prediction, referred to above, is made by assuming that if the word that causes the cache-miss occurs in the bottom half of the buffer, the next higher line will be required, but if the miss occurs in the top half, then the next lower line will be required. If data is being accessed sequentially in either a forwards or backwards direction, then if the first prediction is wrong, it is easy to see that the next prediction will be correct. (It is left as an exercise for the reader to verify this.)

- **L1 Cache**

  The L1 cache contains 512 lines of 128 bytes each and is 128-way associative. This means that any word in memory can be loaded into any one of 128 lines in a specific *congruence class* (determined by bits 55 and 56 of the address to be specific).

Cache lines are loaded, and replaced, in their entirety, starting with the word that is referenced. Lines are replaced on a round-robin basis within each congruence class.

- **L2 Cache**

  Each processor has a private L2 cache of 4 MB. Real addresses are directly mapped to the cache. This means that 128 byte line in memory has just one place in the cache into which it can be loaded. Since the loading and replacement of lines depends on real addresses, replacement may appear to be random as far as a program is concerned.

- **L1 Interleaving**

  The L1 cache is 8-way interleaved to achieve multiple accesses per cycle. There is 4-way interleaving on cache lines and 2-way interleaving on double-words. This means that a pair of load accesses to the cache can execute in the same cycle with the following exception: successive accesses to two even double words or to two odd double words (same bit 59) that are in the same congruence class (same bits 55 and 56) cause one of the accesses to be delayed by one cycle.

  In addition, if there are two or more lines in the cache with identical addresses in bits 43 though 54, the cache access method allows only one of them to be accessed without penalty. The other(s) will incur a delay of approximately seven cycles.

- **Translation Lookaside Buffer**

  The translation lookaside buffer (TLB) contains 256 entries and is 2-way associative. Each entry provides the resolution between a virtual and real memory address for a 4 KB page. If there is an appropriate entry in the TLB, a virtual address can be translated to a real address without any additional cycles.

  However, only 1 MB of memory can be covered by the TLB entries, and in the absence of a TLB entry, a table entry group, occupying 64 bytes, must be fetched from memory. This may in itself cause a cache-miss. Also, the address of the TLB entry is found by a hashing algorithm, and so the entry may not be found at the first attempt.

- **Fortran Compiler Flags**

  This following is a small selection of the compiler flags that have been found to be most relevant for tuning.

  -O2 will optimize the program but maintain the semantics. That is, it will not change the order of computation specified by the program if this may cause the results to be non-bit wise identical. It will do a minimal amount of unrolling.

-O3 will optimize the program and may change the order of computation to give mathematically equivalent (but non-bitwise identical) results. It may implement a significant amount of unrolling.

-qstrict can be used with -O3 to obtain optimization benefits, but to maintain bitwise identical results.

-qarch=pwr3 will use the new POWER3 instructions, including those for single precision computation. It will not use the POWER2 quad-word load instruction.

-qfloat=hsflt will enable divides to be calculated by computing the reciprocal followed by one or more multiplies. This can result in significant speedups if two or more divides are replaced in this way.

Note that without -qarch=pwr3, hsflt also removes all checking when double precision numbers are converted to single precision. This speeds up computation but is not safe if a single precision exponent may exceed its limits.

However, with -qarch=pwr3, this is not the case. Checking is implemented, or POWER3 single precision instructions are used.

-q64 tells the compiler to use 64 bit integer instructions for integers that have been declared as 8 bytes (either with -qintsize=64 or INTEGER*8). Note that -q64 also has many other implications (see 3.3, “64-Bit Support” on page 19).

In the following discussions, -O3 and -qarch=pwr3 optimization is assumed unless stated otherwise.

7.2 CPU Tuning

The POWER3 processor is similar to that of the POWER2. Differences are mainly due to the increase in the floating-point pipe length from two or three to three or four cycles. There is also an additional integer unit and two load/store units.

7.2.1 Unrolling

Since the Model 260 has floating-point pipes of three or four cycles long, up to six to eight instructions, which are not dependent on each other, should be scheduled successively. Also, for best performance, the number of loads plus

the number of stores should not exceed the number of floating-point operations.

Usually, the Fortran compiler does an excellent job of unrolling the loops (particularly when using the -O3 flag). This facilitates the overlapping of independent operations. However, there are some occasions when the compiler does not succeed, and some assistance in unrolling the loops is beneficial. It is difficult to give any firm rule about this because the compiler improves with each release, but a couple of examples (using V5.1.1.0 of the compiler) are given in the following sections.

Further examples are also presented in Chapter 8, “Fortran Tuning Guide for Maximum Megaflops” on page 107.

7.2.1.1 Convolution

The convolution algorithm is frequently used in signal processing. It is included here as an example of how unrolling can be used to achieve nearly maximum possible performance. The basic code shown below runs at only about 150 MFLOPS:

```
DO I=1,1500
  DO J=1,150
    C(I)=C(I)+B(I+J-1)
  ENDDO
ENDDO
```

If the code is unrolled as shown below, up to ten independent floating-point multiply/add operations can be overlapped. Also, for a total of 20 floating-point operations, only 13 loads are required. Theoretically, the Model 260 can process load/store operations at the same rate as floating-point operations, but, because of L1 Interleaving (see 7.1, “Relevant Information” on page 87), it is generally better to have fewer load/store operations.

```
DO I=1,1500,10
  S0=0.E0
  S1=0.E0
  ...
  S9=0.E0
  DO J=1,150,2
    C0=C(I+J-1)
    C1=C(I+J)
    ....
    C10=C(I+J+9)
    B0=B(J)
    B1=B(J+1)
    S0=S0+B0*C0
```

```
      S1=S1+B0*C1
      ....
      S9=S9+B0*C9
      S0=S0+B1*C1
      S1=S1+B1*C2
      ....
      S9=S9+B1*C10
    ENDDO
    C(I)=S0
    C(I+1)=S1
    ....
    C(I+9)=S9
  ENDDO
```

This code ran at 785 MFLOPS, which is very close to the theoretical maximum of 800 MFLOPS. The compiler flags used were -O2 and -qarch=pwr3. Interestingly, this code was originally written for the POWER1 in 1990 and still produces exceptional performance.

7.2.1.2 Multiple Loads

The following example is used to demonstrate both unrolling for good processor performance (when the data is in the cache) and the effect of prefetch on multiple streams of data when the data is not in the cache (see “Multiple Streams” on page 97).

The basic loop for L streams of data in arrays A1 through AL is:

```
DO I=1,N
  S = S + A1(I)*A2(I) + A3(I)*A4(I) + ...AL(I)
ENDDO
```

This type of summation has a history of giving the compiler problems, and this loop does not perform well with the current version of the compiler (V5.1.1.0), either with -O3 or -O4. The loop can be rewritten as below to give improved performance:

```
DO I=1,N
  S1 = S1 + A1(I)*A2(I)
  S2 = S2 + A3(I)*A4(I)
  ....
ENDDO
S = S1 + S2 + ...
```

Results for both loops are shown in Figure 12 on page 93, where each data array is only 8 KB so that all data arrays will be kept in the cache.

The performance of the loop is limited by the data transfer rate. The maximum theoretical rate is 3.2 GB/s. This is almost achieved for two streams by both the basic and the hand unrolled loop. The basic loop also achieves 3.2 GB/s for eight streams, although the hand unrolled loop gives as good or better results except for the other streams.

It is not easy to fully understand how the compiler unrolls and overlaps instructions, and since this is likely to change from release to release, no attempt is made to do so here. Suffice to say, that if, for any loop, the compiler does not do as well as expected, then it may be beneficial to unroll the loop by hand. The right thing to do is to try it and see.

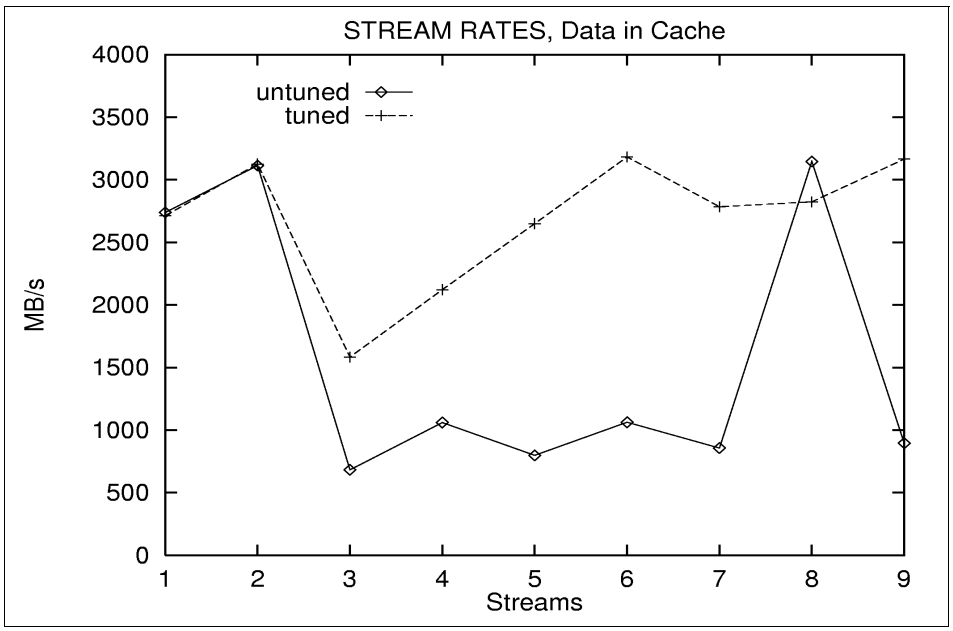


Figure 12. Stream Rates for Data in Cache

7.2.2 Divides

As always, divides take a lot of cycles (14 for single precision floating point, and 18 for double precision), and should be avoided where possible. However, where they cannot be avoided, they frequently take a large part of the processing time, and every effort should be made to minimize their effect.

If there is more than one divide using the same denominator in a loop, then if hsflt is specified, the Fortran compiler does a good job of taking the reciprocal and then multiplying by the reciprocal. Where possible, the compiler also

does a good job of scheduling two divides (or reciprocals) together so as to use both floating-point pipes, often unrolling the loop to achieve this. However, in the following fragment from the LU decomposition of a tridiagonal solver, this is not possible:

```
DO I=1,N
  T    = B(I)+A(I)*B(I-1)
  B(I) = -C(I)/T
  Y(I) = (Y(I)-(A(I)*Y(I-1))/T
ENDDO
```

The compiler will take the reciprocal of T and multiply it twice. It will also unroll the loop, but because the loop is recursive, the reciprocal in one loop is dependent on the reciprocal in the previous loop, and so two reciprocals cannot be scheduled together. To enable the compiler to achieve this, two (or more) tridiagonal solutions have to be coded together. For example:

```
DO I=1,N
  T     = B (I)+A (I)*B (I-1)
  T1    = B1(I)+A1(I)*B1(I-1)
  B (I) = -C (I)/T
  B1(I) = -C1(I)/T1
  Y (I) = (Y (I)-(A (I)*Y (I-1))/T
  Y1(I) = (Y1(I)-(A1(I)*Y1(I-1))/T1
ENDDO
```

This enables the compiler to schedule two divides together, and the solver to run up to two times faster.

If it is decided not to compile with hsflt because it may be unsafe (see “Relevant Information” on page 87), then the reciprocal computation should be hand coded.

7.2.3 Floating Point to Integer Conversion

Floating point to integer conversion is particularly important in many seismic codes, where it is used to create an index for table lookup. Floating point to integer conversion is implemented by hardware instructions, but still takes a relatively long time. For example, a loop containing:

```
J(I) = INT(S(I)+A)
```

takes over 5 cycles. By contrast:

```
J(I) = ISHFT((JS(I)+IA),-10)
```

takes only about 1.5 cycles. The array `JS` has been initialized to contain the same values as the array `S`, multiplied by 1024 to preserve accuracy, and then

accessed many times with the above code. Much greater accuracy can be obtained if the 64-bit integer arithmetic capability of POWER3 is used. 64-bit integers can store numbers up to $8x1024^6$; so the JS array could contain the values of S multiplied by two or three powers of 1024.

To use 64-bit arithmetic, the required arrays must be declared INTEGER*8, and the -q64 compiler flag must be used to tell the compiler to use 64 bit integer instructions. Note that the whole program must be recompiled with -q64 in order for all routines to link correctly.

7.2.4 Fractional Part of a Number

There is a useful trick (provided by Jim Shearer - one of the authors of the MASS Library - from the IBM Watson Research Laboratory at Yorktown) that can be used to obtain the fractional part of a floating-point number. This is often required for interpolation purposes. The fractional part of a double precision number would normally be obtained, within a loop, by:

```
F(I) = A(I) - FLOAT(DNINT(A(I))
```

This can be done approximately 8 times more quickly by using:

```
PARAMETER( RND=2D0**52+2D0**51)
. . .
F(I) = A(I) - (RND + A(I) - RND)
```

The code was compiled using -O3 and -qstrict. It was necessary to use -qstrict to prevent the compiler changing the order of computation, and setting F(I) equal to zero.

7.3 Memory Tuning

The Model 260 memory subsystem has a major advantage over previous POWER2 systems in that it can support four concurrent cache-misses and four prefetch streams.

This capability is discussed in the following sections.

7.3.1 Copy

A straightforward copy is as follows:

```
DO I=1,N
  X(I)=Y(I)
ENDDO
```

However, a store-miss is not prefetched. A store-miss occurs when a store instruction causes a line miss. The correct line must then be fetched into cache before the store instruction can store the data.

This delay can be overcome by taking advantage of the load-miss prefetch capability. The array X is loaded and multiplied by zero before it is stored. The tuned code becomes:

```
DO I=1,N
  X(I)=Y(I)+ZERO*X(I)
ENDDO
```

Results are shown in Figure 13 on page 96.

The multiply/add instruction does not require any extra time because it is overlapped with the load and store instructions, but when the data is in the L1 cache (that is when the data arrays are each less than 32 KB), the additional load causes the copy to run more slowly. This is because there are now three load/store operations and there are only two load/store pipes.

However, when the data is not in the L1 cache, the advantage of prefetch predominates and the copy runs more quickly.

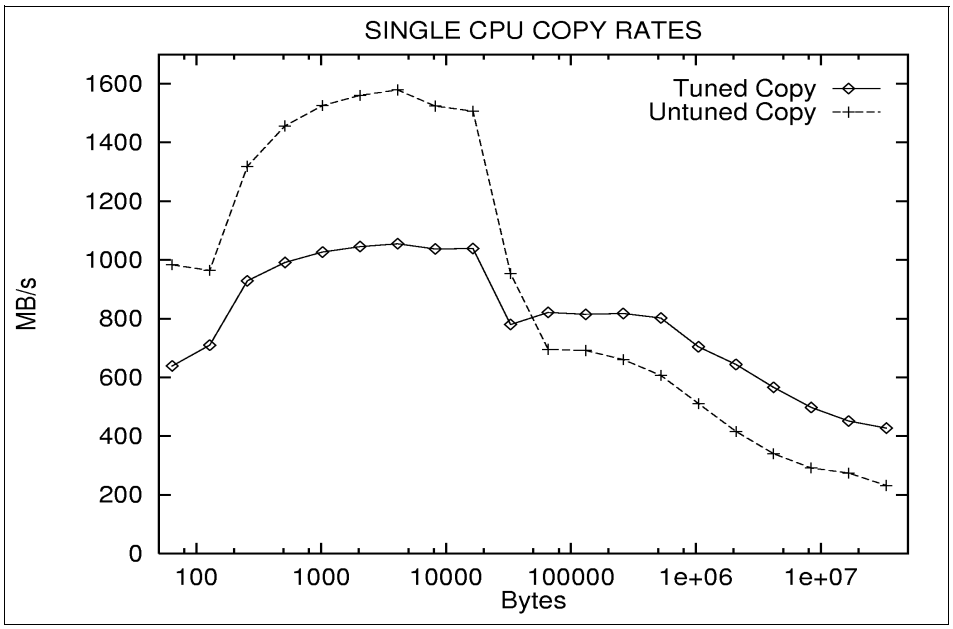


Figure 13. Single Processor Copy Rates

7.3.2 Multiple Streams

The hardware will prefetch up to four streams of data. The Fortran loops considered here have been described in 7.2.1.2, “Multiple Loads” on page 92. The effects of accessing multiple streams of data, when the data is not in the L1 (or L2) cache, is shown in Figure 14.

As can be seen, the maximum aggregate rate is achieved with four streams.This to be expected since the Model 260 implements up to four prefetch streams. The maximum rate achieved is 1.36 MB/s, which compares well with the theoretical maximum of 1.6 MB/s.

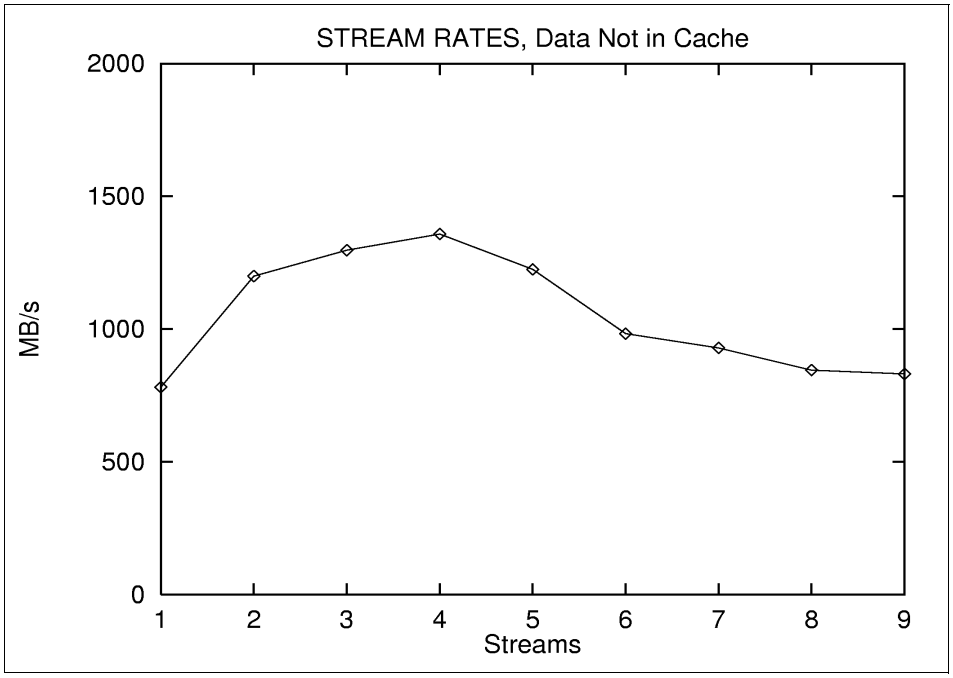


Figure 14. Stream Rates for Data Not in Cache

With only one stream, the prefetch mechanism cannot keep up with the cache line requests. Each prefetch begins when the previous line is accessed (see Figure 15 on page 98). As can be seen, the data transfer is overlapped but latency is not, and since the latency is about 35 cycles, the expected rate is about

```
128bytes/(35x5nsec) = 730 MB/s
```

This agrees well with the measured rate in Figure 14.

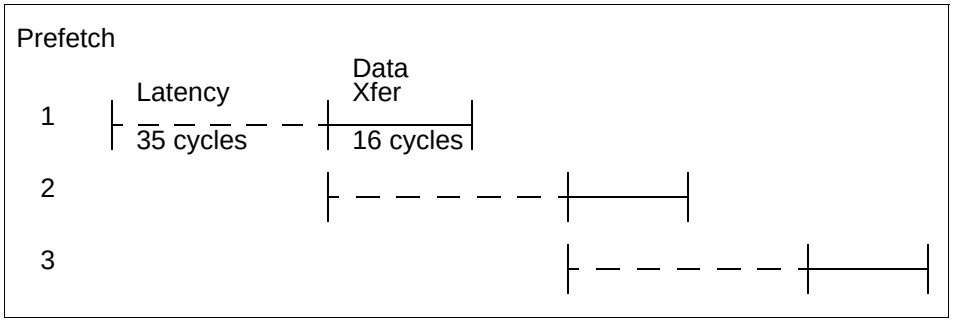


Figure 15. Single Stream Prefetch

7.3.3 DAXPY

The DAXPY algorithm is frequently used as a measure of memory performance because the MFLOP rate is limited by the storage access rate. The Fortran code is:

```
DO I=1,N
  Y(I)=Y(I)+A*B(I)
ENDDO
```

Note that it is similar to the tuned copy (see 7.3.1, “Copy” on page 95, and Figure 13 on page 96). Measurement of the performance of this loop provides some interesting results.

The first measurement is shown in Figure 16 on page 99. Note that there are several downward spikes, occurring when the number of bytes is too large for the L2 cache. Further measurements showed that the exact position of the spikes varied with each run. Combining the best results for each individual point from four runs, gave the results in Figure 17 on page 100.

The presence of the downward spikes is due to the overlaying of 128 byte lines in the L2 cache. This varies with each run because the L2 addresses are directly mapped to memory, and the allocation of the program’s virtual memory to real memory changes with each run.

The results in Figure 17 still show a sharp dip at 4 KB. This was because the start of the X and Y arrays are separated by a large power of two plus 4 KB. The explanation is as follows. If the addresses of any two lines in the L1 cache have the identical values for bits 43 through 54, then only one of them can be accessed without penalty. When the other is accessed, a delay of approximately seven cycles occurs. This means that if two arrays, which are

exactly 2 MB apart, are sequentially accessed, then the seven cycle delay will continually occur.

The two arrays were therefore separated by an additional 64 KB (the L1 cache size), and the results in Figure 18 on page 100 were obtained.

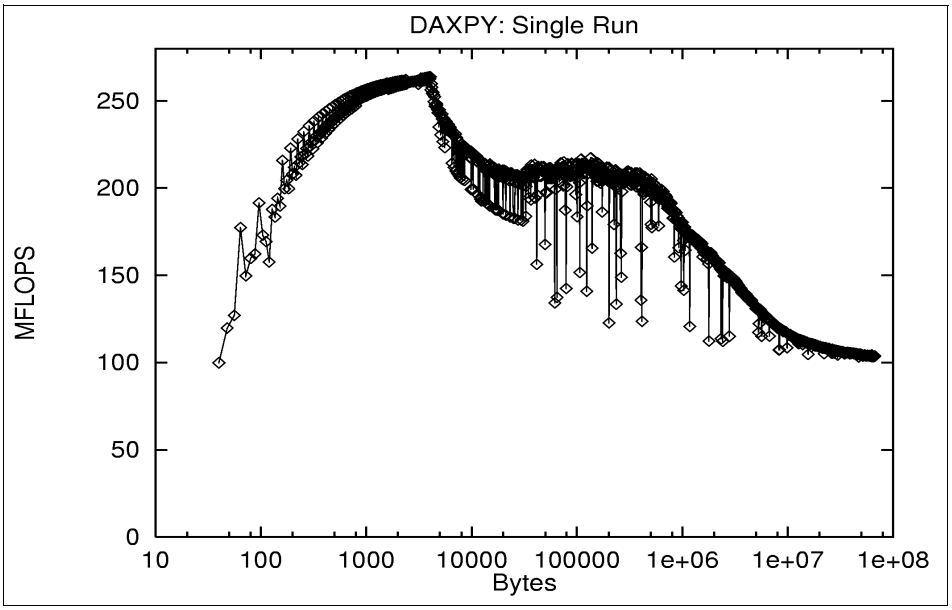


Figure 16. DAXPY: Single Run

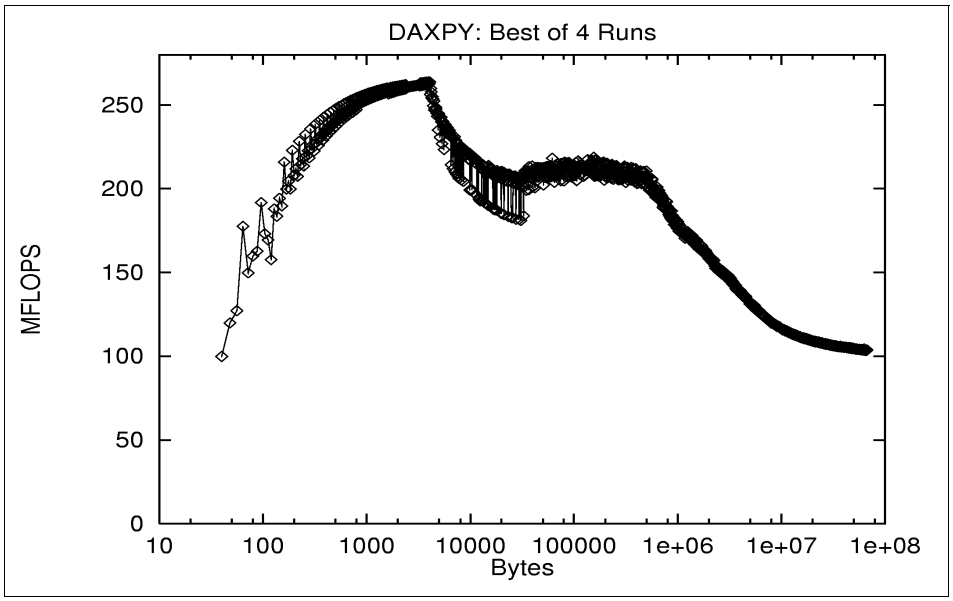


Figure 17. DAXPY: Best of 4 Runs (1)

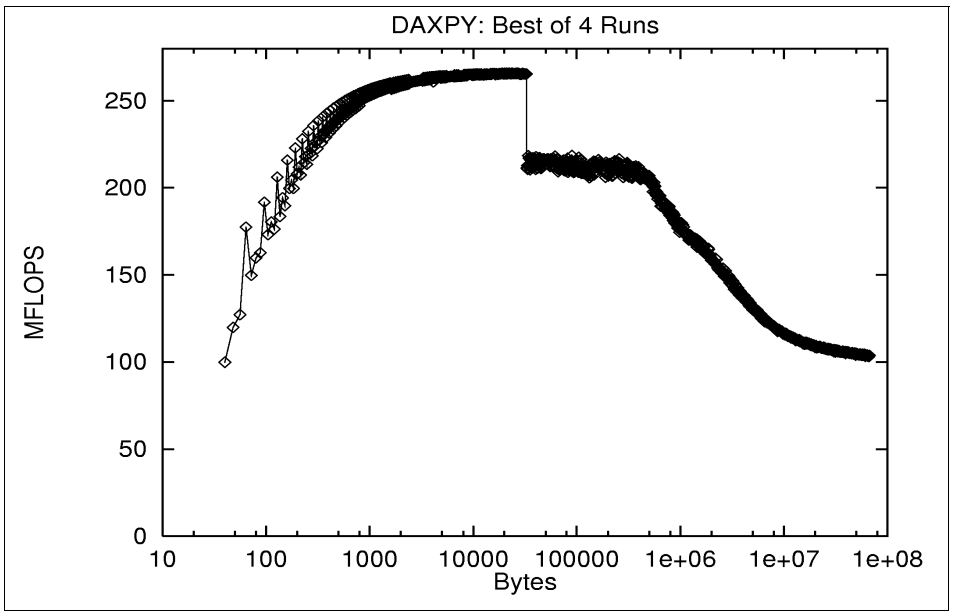


Figure 18. DAXPY: Best of 4 Runs (2)

7.3.4 Loads and Stores

It is very apparent, by comparing the MFLOPS rates obtainable using the DAXPY algorithm with the MFLOPS that can be obtained using the convolution algorithm, that loads and stores can severely inhibit achievable MFLOPS. A simplified example of the type of construction which is often found in vector processing legacy code is:

```
DO J=1,N
  B(J)=XYZ*A(J)
ENDDO
DO J=1,N
  C(J)=ABC*B(J)
ENDDO
```

Fairly obviously, this should be replaced by:

```
DO J=1,N
  C(J)=ABC*XYZ*A(J)
ENDDO
```

thus saving one store and one load.

7.3.5 Prefetching Individual Cache Lines

It is possible to prefetch individual cache lines. The following three loops give an example of this. (Note that the loops are entirely for the purpose of illustration and are in no way meant to represent a real code.)

```
DO I=1,N
  Y=A(I)
  S=S+SQRT(SQRT(SQRT(Y))) ! Automatic Prefetch
ENDDO

DO I=1,N
  Y=A(IND(I))
  S=S+SQRT(SQRT(SQRT(Y))) ! No Prefetch
ENDDO

DO I=1,N
  Y=X
  X=A(IND(I+1))
  S=S+SQRT(SQRT(SQRT(Y))) ! Hand Coded Prefetch
ENDDO
```

The array `A` covers 8 MB, which is too large to fit into the L2 cache.The first loop accesses data sequentially. The second loop access data randomly with the statement `X=A(IND(I))`, where the array `IND` contains randomly ordered

indices into the array A. The third loop also accesses data randomly, but prefetches the cache lines with the statement X=A(IND(I+1)).

The number of cycles taken by each iteration of the loops is 97,152, and 99, respectively.

Comparing the first and second loop shows that the time to fetch a cache line (including the time to fetch the TLB table entry group), which is incurred by the second loop, takes approximately 152 - 97 = 55 cycles. Comparing the third loop with the first and second loops shows that fetching a cache line is almost entirely overlapped by the computation.

Practical examples of this are harder to achieve because of the previously mentioned restrictions (7.1, “Relevant Information” on page 87) concerning the number of operations that may be simultaneously in progress, the completion order, and the number of completions per cycle.

7.4 Large Stride

For large stride, the effects of the cache and TLB become apparent. These are discussed in the following sections.

7.4.1 Cache Effects

The time taken to access a double word of data varies considerably with the loop count and the stride. This was measured by a loop similar to:

```
REAL*8 A(M,*)
DO J=1,N
  S=A(1,J)
ENDDO
```

where N is the loop count and M is the stride. Actually, the loop was unrolled by hand and compiled with -O2, since -O3 would have optimized away the loop completely.

If this loop is iterated many times, then for small loop counts and stride 1, the data will be in the L1 cache. When the loop count becomes greater then 8K, the data exceeds the cache size, and the cache is flushed every iteration.

As the stride increases, less data can be kept in the cache, and when the stride is 16 or greater, one cache line (of 16 double words) is required for each item of data.

The L1 cache structure is summarized in 7.1, “Relevant Information” on page 87.

A visual representation of the effects of the L1 cache is shown in Figure 19 on page 104. Stride goes from left to right from 1 to 300, and loop count goes from 1 to 600 from top to bottom. The time taken to access data is represented by the greyness. Light tones represent the fastest access and dark tones the slowest.

The dark area at the top of Figure 19 represents relatively slow access because of the overhead of setting up the loop.

Strides of 2, 4, 6, and 10 are slower because of double word interleaving. A stride of 8 is good because successive alternate words are in different congruence classes and do not suffer from interleaving.

Strides that are multiples of 32 words perform poorly if the loop count is higher than 256 because they map to only two of the cache’s four 128-entry congruence classes.

Strides which are multiples of 64 words are worse if the loop count is higher than 128 because they map to only one of the one of the cache’s four 128-entry congruence classes.

7.4.2 Translation Lookaside Buffer Effects

The TLB structure is summarized in 7.1, “Relevant Information” on page 87.

A visual representation of the effects of the translation lookaside buffer (TLB) for a range of strides and loop counts (using the loop described in section 7.4.1, “Cache Effects” on page 102) is shown in Figure 20 on page 105. Stride goes from left to right from 1 to 96 KB, and the loop count goes from 1 to 300 from top to bottom.

The TLB contains 256 entries and is two-way associative. This means the entry to resolve the virtual to real address of any 4 KB page can go into just two slots. Virtual page addresses that are multiples of 512 KB apart must compete for the same two slots. Thus, a stride of 64 KB will incur a TLB miss after a loop count of 16, a stride of 32 KB after a loop count of 32, and so on. This is exactly what is shown by Figure 20.

For other strides, the effect is a good example of chaos theory. Very small differences in stride have a very large effect on performance. However, the chaos is actually predictable, and a program that, using the TLB structure described earlier, recreates the results with remarkable accuracy was written.

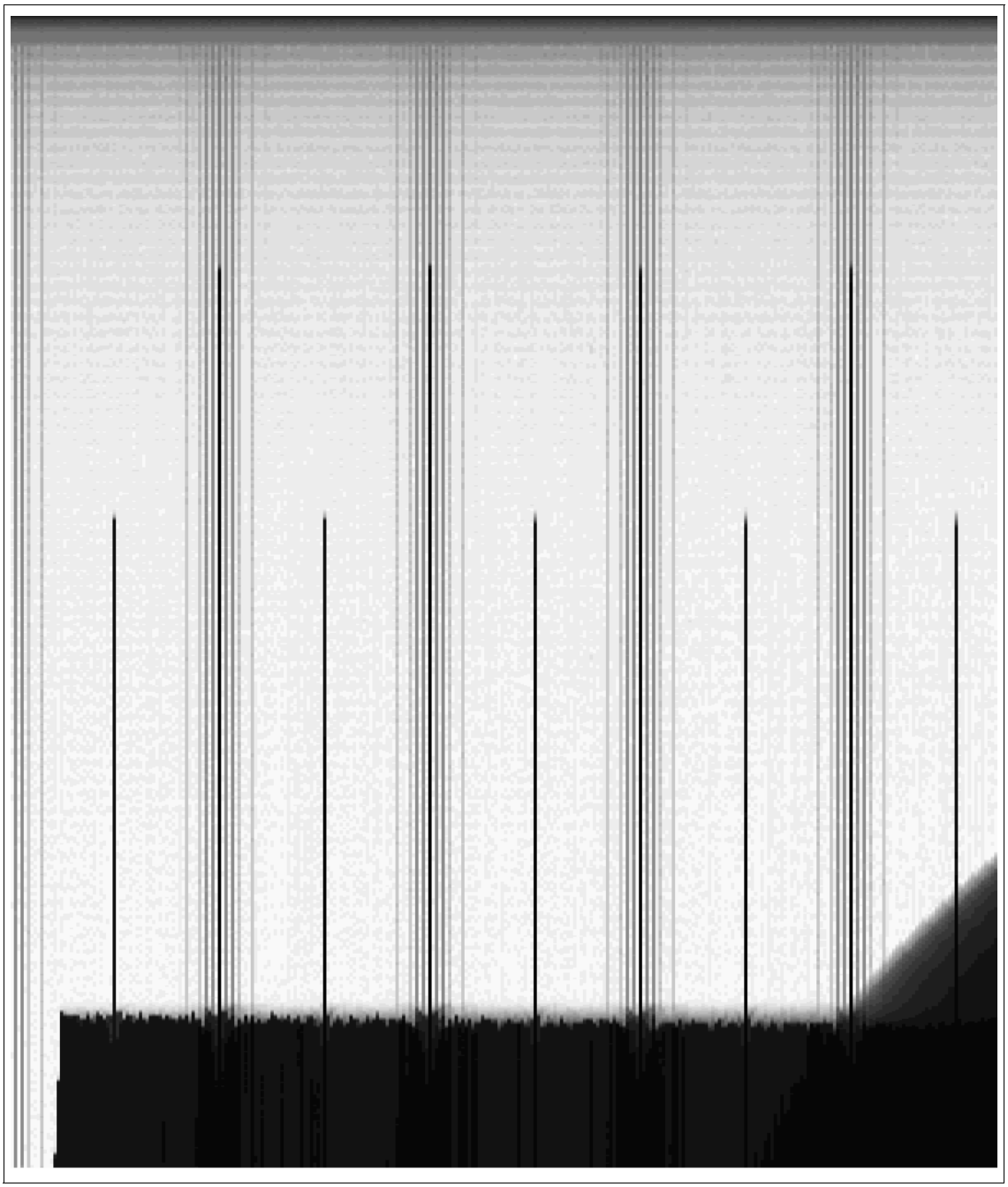


Figure 19. Stride versus Loop Count for L1 Cache

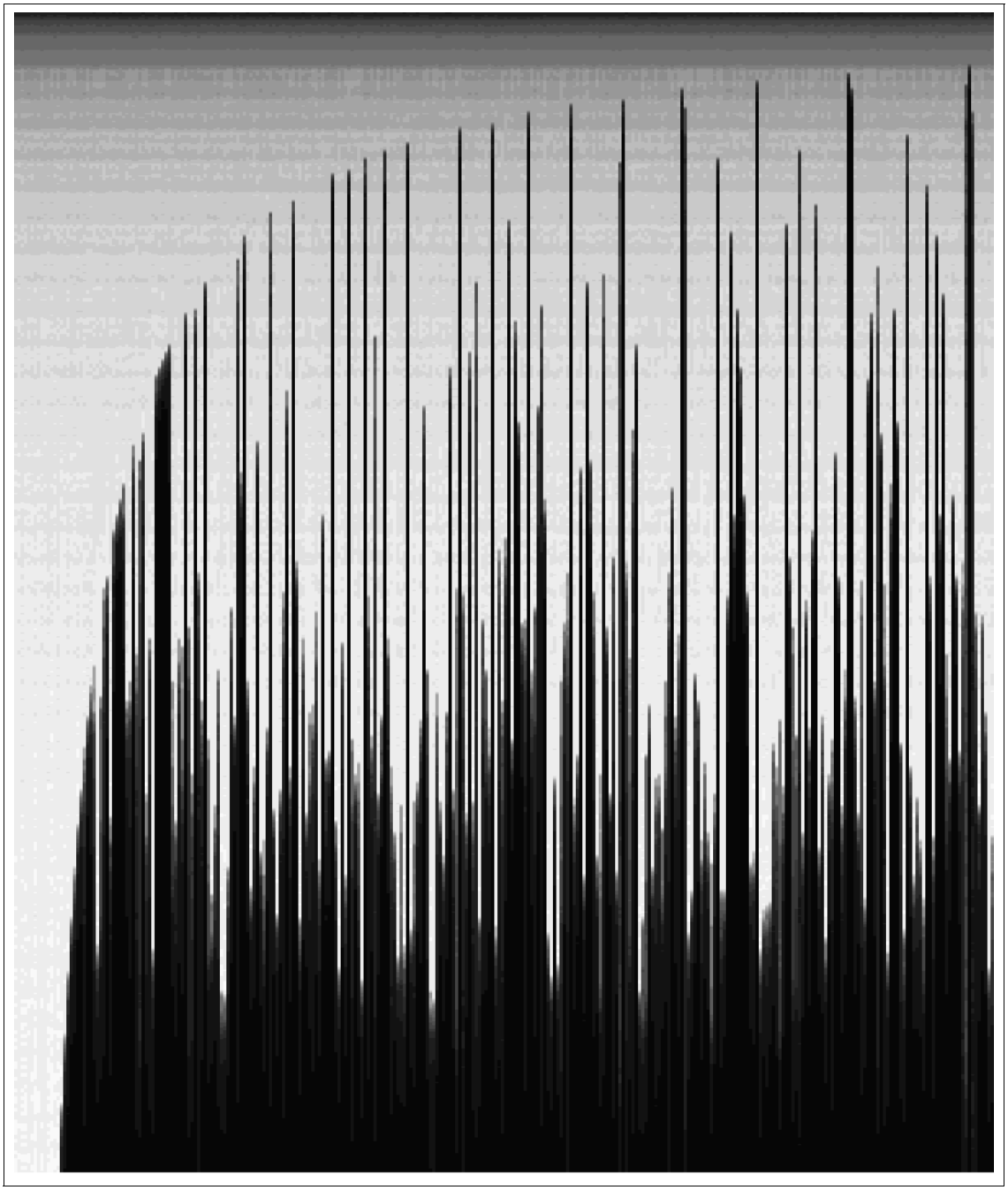


Figure 20. Stride versus Loop Count for TLB

Chapter 8. Fortran Tuning Guide for Maximum Megaflops

This chapter constitutes the basic Fortran tuning guide for POWER3. It is intended for Fortran programmers who have relatively little experience of tuning for IBM RS/6000 RISC architecture, in contrast to Chapter 7, “Performance and Tuning Analysis” on page 87, which was intended more as a POWER3 update for programmers already experienced in tuning for POWER2.

The subject of tuning is covered in much more detail in *Optimization and Tuning Guide for Fortran, C, and C++*, SC09-1705. This chapter is, nevertheless, intended to be complete in itself and to cover that subset of optimization and tuning techniques for POWER2 and POWER3 that those working in the field regard as key, together with new material relating to POWER3.

It is structured as follows:

- The tuning process
- Recommended compiler options for performance
- Architecture-independent hand-tuning review
- Key aspects of POWER3 architecture:
  1. The L1 data cache
  2. The L2 data cache
  3. The translation lookaside buffer (TLB)
  4. The superscalar floating point units (FPUs)
- Tuning for peak megaflops on POWER3:
  1. Avoid the negative. Tune for the data cache and TLB.
  2. Exploit the positive. Tune for the superscalar FPUs.
- Some comments on SMP parallel tuning for POWER3.

8.1 The Tuning Process

The following steps summarize the process, in approximate order of importance. A short section on each step follows.

1. Consider whether I/O is significant and tune if necessary.
2. Use the best set of compiler optimization flags.

3. Locate the hot-spots in the program.

   This step is very important. Do not waste time tuning code that is hardly ever executed.

4. Use the MASS library for intrinsic function references and code calls to pre-tuned libraries, such as ESSL where possible. MASS and ESSL are described in Chapter 5, “Performance Libraries” on page 65.

5. Hand-tune the code.

   This is the subject of the remaining sub-sections.

8.1.1 Tuning for I/O

This item is considered first since, if I/O is a significant part of the program, it may well dominate the overall run time and render CPU tuning unproductive. Some guidelines for efficient I/O in Fortran are given in the list following the next paragraph, but the main advice is simply to eliminate or minimize I/O as much as possible. If I/O is your performance bottleneck, then using the best hardware and software options (SSA disks, striping over multiple devices and adaptors, and asynchronous I/O, for example) may be the best tuning option. A detailed discussion of this is outside the scope of this publication.

Paging is a special case of I/O. You can measure paging rates using `vmstat`. A certain amount of paging during start-up or when the program changes from one phase to another is to be expected. But any measurable paging rate over a sustained period during program execution is an indication that you are over-committing memory or are on the edge of doing so. This is likely to cause serious performance problems. The only solution is to reduce the level of memory over-commitment. Either tune the program so as to use less memory - or run on a computer with more memory (or fewer users).

Some guidelines for efficient I/O in Fortran follow:

- Use long record lengths.

  At least 100 KB if possible, preferably 2 MB or more.

- Prefer Fortran unformatted I/O to formatted.

  This reduces binary to decimal conversion overhead.

- Prefer Fortran direct files to sequential.

  This avoids Fortran record length and overflow checking. A Fortran direct file in AIX is a simple sequential series of data bytes. A Fortran sequential file has record length indicators at both ends of each record.

- Reduce the number of calls to the I/O subsystem.

For example, the following three ways of writing the whole of a 2-D array to a sequential file differ very considerably in performance. As well as performing very slowly, Case 3 will create a file almost twice as large as Case 1 (if A is REAL*8) because of the extra record length indicators.

```
      DIMENSION A(N,N)
          .
          .

Case 1. Best. 1 record of N*N values.
      WRITE(1)A

Case 2. N records, each of N values.
      DO I=1,N
         WRITE(1)(A(J,I),J=1,N)
       ENDDO

Case 3. Worst. N*N records, each of one value.
      DO I=1,N
        DO J=1,N
            WRITE(1)A(J,I)
         ENDDO
      ENDDO
```

- Use asynchronous I/O to overlap computation with I/O activity. This is newly implemented in XL Fortran Version 5 using the ASYNCH keyword on OPEN and the WAIT statement.
- If you write a large temporary file sequentially and need to read through it again at a later stage in processing, make it a direct access file and then try to read the end records of the file first. Ideally, read it sequentially backwards. This is because AIX will automatically use memory to buffer the file. Assuming the file is larger than memory, after the write is completed, memory is likely to contain a large number of buffers corresponding to the last part of the file. If you then read these records, AIX will supply them to the program from memory without physically reading the disk. If you read the file forwards, the incoming records from the front of the file will flush out the in-memory buffers before you reach them.

8.1.2 Locating the Hot Spots (Profiling)

Profiling tells you how the CPU time used by a program during execution is distributed over the code. It identifies the active subroutines and loops so that tuning effort can be applied most effectively.

It is important to understand that a profile relates just to the particular run of the program for which the profile was obtained. The same program run with different data will produce a different profile. Some numerically intensive programs produce very consistent profiles with widely varying sets of input data. Others produce quite different profiles when the data is changed.

From the point of view of the person tuning the code, the ideal situation is a consistent profile with very pronounced concentrations of time spent in a few routines. Tuning effort can then be concentrated on those routines.

The AIX tools available for profiling the programs include:

- The AIX `prof` and `gprof` commands
- The AIX `tprof` command

The `prof` and `gprof` commands provide profiling at the procedure (subroutine and function) level. The `tprof` command uses the AIX trace facility to interrupt your program at each tick (10 milliseconds) of the AIX CPU clock and construct a trace table that contains the hardware instruction address register. At the end of your program execution, `tprof` creates a report (using the trace table) showing the number of ticks that relate to each line of your source code.

To use `prof` and `gprof`, do the following:

1. Compile your program with the -p or -pg option in addition to the normal compiler options
2. Run the program (this produces the gmon.out file)
3. Run prof or gprof by entering prof > *filename* or gprof > *filename*

The standard output, *filename*, of prof will contain the following information:

- The percentage of the program's CPU time used by the procedure.
- The time in seconds required for all references to the procedure.
- The cumulative total of seconds required for all procedures in the list.
- The number of times the procedure was called and the time required to perform each call.

The output of `gprof` contains all the information provided by `prof`, and in addition the timing information of the calling tree for the procedures in the program.

To use `tprof` on a program myprog.f, do the following:

1. Compile your program with the -g option
2. Run tprof on the program: tprof -p myprog -x myprog

This procedure creates two output files, namely __myprog.all and __t.myprog.f. The first file shows all the processes involved in running your program and provides a count of the timer ticks associated with each process. It also lists the percentage of ticks that are associated with the complete program. The second file is only produced if you compile your program with the -g option. It is an annotated version of your source file, that indicates the CPU ticks associated with each line of the source code being executed.

For more details on how to use `prof`, `gprof`, and `tprof`, see *Optimization and Tuning Guide for Fortran, C, and C++*, SC09-1705.

By far the most user-friendly and powerful tool, providing graphically assisted profiling down to the Fortran or assembler statement level, is `Xprofiler`, which is a development of `xgprof`. `Xgprof` is an unsupported IBM Internal tool. `Xprofiler` is a supported IBM product distributed as part of Parallel Environment - normally used only for the distributed memory RS/6000 SP. If you are running on a workstation where PE is not installed, your option is to use `prof`, `gprof`, or `tprof`.

To use `Xprofiler` (or `xgprof`), compile and link as for `gprof` with

```
-g -pg
```

together with -O3 or whatever other optimization you are using. It is important to use the same optimization options as you will use for production, since changing the optimization is highly likely to also change the profile.

Then simply run the executable against the chosen test data. This will produce the standard gmon.out file containing the profiling data. Then run `Xprofiler`. Graphics will appear showing the subroutine tree of the program, with each subroutine represented by a rectangle. The area of each rectangle is roughly proportional to the CPU time spent in that routine, giving an immediate visual indication of hot-spot location. Clicking on a rectangle will produce a set of options, one of which creates a Fortran source code listing with each statement annotated with the amount of CPU time (in units of 1/100 s) used. This enables the active loops to be easily identified.

8.1.3 Use Pre-tuned Code, Such As ESSL

Do not spend time duplicating tuning work that has already been done. If your program performs standard functions, such as matrix multiply, equation

solving, other BLAS functions, FFTs, convolution, and so on, then modify your code to call the equivalent ESSL function. ESSL is described in 5.1, “The ESSL Library” on page 65, and contains probably the most highly tuned code available for RS/6000 numerically intensive functions. Other commercially and publicly available libraries, such as NAG, IMSL, LAPACK, and so on, have also been tuned for cache-based superscalar architectures.

8.1.4 Hand Tune the Code

Hand tuning is a last resort, since it is likely to require a lot of time and effort. Nevertheless, the remainder of this Tuning Guide section is devoted to it.

8.2 Recommended Compiler Options

The XL Fortran compiler is constantly improving, and you should beware that recommendations in this section are likely to become out of date. Currently, however, the following represents the practical experience of people working in this field and relates to XL Fortran Version 5.1.

Recommended sets of options are given first, followed by a detailed set of notes that justify the recommendations.

Recommended set of performance options for POWER3:

```
-O3 -qarch=pwr3 -qtune=pwr3 [-qcache=auto]
```

or

```
-O3 -qstrict -qarch=pwr3 -qtune=pwr3 [-qcache=auto]
```

- Only specify -qcache=auto if compiling on POWER3.
- Use -qstrict if you are worried about non-bitwise identical results.
- Try -O4 selectively and check to see if performance improves.
- Consider using -qipa and -Q.
- Consider using -qfloat=hsflt (but beware that it can be unsafe - see the notes which follow).
- Use -O3 -qarch=com -qtune=pwr3 if you want to tune for POWER3 but have the executable run on other platforms.

The following detailed set of notes explains the reasoning behind these recommendations:

- Unless you are debugging, *always* use at least the -O2 flag (or -O) (note that -O1 is not implemented). The performance of unoptimized code will almost always be very poor and render pointless any hand tuning you might do.
- The -O3 option is reliable (in terms of giving correct answers) and usually (though not absolutely always) gives improved performance over -O2. Therefore, use it as a matter of course in preference to -O2, unless you have good reason for believing it to be generating faulty code or degrading performance. See also the discussion of -qstrict that follows in this list.
- In the present release of the compiler (Version 5.5.1), the -O4 option is a short-hand for

  ```
  -O3 -qhot -qipa -qarch=auto -qtune=auto -qcache=auto
  ```

  That is, there is no optimization enabled by -O4 that is not given by that set of options. This may not remain true in future releases.
- The -qhot (high order transforms) option, implied by -O4, is excellent for blocking and transforming simple loops for optimum cache and TLB performance. See 8.4, “Key Aspects of POWER3 (Model 260) Architecture” on page 119, for a detailed discussion of the data cache and TLB.

  Early experience with XL Fortran Version 5 shows that -qhot is significantly improved over Version 4. For example, it now does an excellent job at optimizing untuned matrix multiply coding. However, -qhot is less successful with more complex loops. Practical experience (with XL Fortran Version 4) has indicated that, with real production codes, -qhot degrades performance more often than it improves it. Although improved in Version 5, -qhot should probably still not be recommended for routine use. Rather, use it selectively, on key subroutines, after you have verified by measurement that performance is improved.
- -qstrict is used with -O3 and higher optimization levels to ensure that results are obtained that are bitwise identical to those from unoptimized code (and -O2). To do this, XL Fortran defines a strict computational ordering based on Fortran’s rules for operator hierarchy and left to right operation. Without -qstrict, optimization levels above 2 allow such semantics changes in the interests of performance.

  For example, when evaluating the expression

  ```
  A*B*C + B*C*D
  ```

  the compiler might recognize that B*C is a common sub-expression and evaluate it once only. However, -qstrict would inhibit this optimization since it would violate the left to right ordering rule on A*B*C. In general, (A*B)*C

does not yield a bitwise identical result to A*(B*C). Which is more accurate - closer to the mathematically exact value - depends on the precise floating points values involved.)

- -qarch=pwr3 and options such as -qfloat=hsflt.

  The various sub-options of -qfloat (hsflt, hssngl, and so on) are primarily intended for single precision (REAL*4) operation on POWER2 architecture, and, since this publication is based on POWER3, details will not given here. Suffice it to say that, for single precision floating point arithmetic on POWER2, hsflt is the highest performing option but that it is unsafe since exponent overflow can go undetected and produce wrong results. The highest performing safe option is -qfloat=hssngl.

  However, there is a potentially important compiler optimization (reciprocal multiply) that is enabled only if -qfloat=hsflt is specified. There is a strong argument for not making this optimization dependent on an unsafe option, and the compiler developers are considering a change. The -qfloat=hsflt option is, however, safe in practice if:

  - You use double precision exclusively (with POWER2 or POWER3), or
  - You specify -qarch=pwr3 and use single precision exclusively (that is, do not mix single and double), or
  - You can guarantee that no expression will ever have a value outside the single precision exponent range (about 1.0E-38 through 1.0E+38).

  Therefore, if you find that reciprocal multiply is of significant benefit for your code, you could consider enabling it with hsflt. However, hand-tuning for reciprocal multiply is usually relatively easy, and this is probably the better option.

8.3 Architecture Independent Hand Tuning Review

Before giving a detailed description of the performance implications of key parts of POWER3 (RS/6000 43P 7043 Model 260) architecture, this section reviews some tuning techniques that have been found to be commonly effective in a wide range of programs. These techniques could be described as common sense. They simply do things in a more efficient way: maybe by reducing the amount of computation to achieve the same result, maybe by eliminating unnecessary overhead.

Most of these techniques are likely to be effective, whatever hardware platform the code is run on, in contrast to the architecture dependent techniques discussed later.

8.3.1 Basic Coding Practices for Performance

Sections on do's and don't's follow.

8.3.1.1 Good Coding Practices for Performance

Write a clean and straightforward program to enable the compiler to do its optimization work.

- Access data sequentially (unit stride), see 8.5.3.1, "Stride Minimization: Case Study T1" on page 129
- Keep size of DO-loops manageable
- Keep common sub-expressions recognizable by the compiler
- Reduce expensive operations (such as divides, exponentiation, and so on)
- Minimize IF statements in loops
- Inline short routines
- Avoid subroutine calls in loops (give routine its own loop)
- Do not EQUIVALENCE critical variables
- Simplify array subscripts
- Prefer scalar temporaries over scratch arrays
- Avoid implicit type conversions
- Keep the number of parameters passed to subroutines and functions small
- Avoid leading array dimensions equal to a power of two
- If coding multiple IF tests, evaluate the most likely first

8.3.1.2 Coding Practices to be Avoided

For performance-critical DO-loops, do *not* do the following:

- Access data with large stride (see 8.5.3.1, "Stride Minimization: Case Study T1" on page 129).
- Create recurrences.
- Do too few iterations of the loop.

And within performance-critical DO-loops do *not* use the following:

- I/O statements
- Subroutine calls
- Non-intrinsic function references
- CHARACTER or LOGICAL assignment statements

- ASSIGN or ASSIGNED GOTO or computed GOTO
- GOTO which exits the loop
- GOTO backwards in the loop
- PAUSE, RETURN, or STOP
- Too many or too complex nested IFs
- Complex loop-dependent array subscripts (induction variables)
- Non-INTEGER or INTEGER*8 DO-loop variables
- EQUIVALENCEd data items
- Non-optimizable data types: LOGICAL*1, BYTE, INTEGER*1, INTEGER*2, REAL*16, COMPLEX*32, CHARACTER, INTEGER*8 in 32-bit mode

8.3.2 Commonly Occurring Examples

These are some examples of how to correct some inefficient coding practices that have been repeatedly found in real codes:

Invariant IF float-out

```
      Untuned
      -------
DO I=1,N
  IF(D(J).LE.0.0)X(I)=0.0
  A(I)=B(I)+C(I)*D(I)
  E(I)=X(I)+F*G(I)
ENDDO
```

```
        Tuned
        -----
IF(D(J).LE.0.0)THEN
  DO I=1,N
    A(I)=B(I)+C(I)*D(I)
    X(I)=0.0
    E(I)=F*G(I)
  ENDDO
ELSE
  DO I=1,N
    A(I)=B(I)+C(I)*D(I)
    E(I)=X(I)+F*G(I)
  ENDDO
ENDIF
```

The compiler will recognize that the IF test is invariant but will not generate two versions of the loop as in the tuned example.

Boundary condition IF testing

Often, you want to do something different for just the first and/or last iteration of a loop. If the loop is performance-critical, then it is important to treat these special cases separately and have the main loop without an IF:

```
    Untuned                          Tuned
    -------                          -----
DO I=1,N                        A(1)=B(1)+C(1)*D(1)
  IF(I.EQ.1)THEN                X(1)=0.0
    X(I)=0.0                    E(1)=F*G(1)
  ELSEIF(I.EQ.N)THEN            DO I=2,N-1
    X(I)=1.0                      A(I)=B(I)+C(I)*D(I)
  ENDIF                           E(I)=X(I)+F*G(I)
  A(I)=B(I)+C(I)*D(I)           ENDDO
  E(I)=X(I)+F*G(I)              X(N)=1.0
ENDDO                           A(N)=B(N)+C(N)*D(N)
                                E(N)=1.0+F*G(N)
```

Repeated intrinsic function calculation

In this example, the untuned code calculates the values of SIN(X(J)) N times, whereas in the tuned code, they are calculated once and saved in a separate array.

```
    Untuned                          Tuned
    -------                          -----
DO I=1,N                        DIMENSION SINX(N)
  DO J=1,N                          .
    A(J,I)=B(J,I)*SIN(X(J))     DO J=1,N
  ENDDO                           SINX(J)=SIN(X(J))
ENDDO                           ENDDO
                                DO I=1,N
                                  DO J=1,N
                                    A(J,I)=B(J,I)*SINX(J)
                                  ENDDO
                                ENDDO
```

Calls to vector merge functions

Codes, typically ported from other systems such, as Cray vector processors, often make extensive use of the vector merge functions, CVMGM, CVMGN, CVMGP, CVMGT, and CVMGZ. They were used to avoid IF statements in loops preventing vectorization. On a non-vector architecture, this is unnecessary. They are supported by XL Fortran but the overhead of calling them is usually much greater than executing the equivalent conditional code. This is particularly true if, for example, CVMGT is called several times with the same logical condition, as in the following example:

```
    Untuned
    -------
DO I=1,N
  P(I)=CVMGT(A1(I),A2(I),D(I).LE.0.0)
```

```
  Q(I)=CVMGT(B1(I),B2(I),D(I).LE.0.0)
  R(I)=CVMGT(C1(I),C2(I),D(I).LE.0.0)
  S(I)=CVMGT(D1(I),D2(I),D(I).LE.0.0)
ENDDO

   Tuned
   -----
DO I=1,N
  IF(D(I).LE.0.0)THEN
    P(I)=A1(I)
    Q(I)=B1(I)
    R(I)=C1(I)
    S(I)=D1(I)
  ELSE
    P(I)=A2(I)
    Q(I)=B2(I)
    R(I)=C2(I)
    S(I)=D2(I)
  ENDIF
ENDDO
```

Replacing divides by reciprocal multiply

This optimization can sometimes be done automatically by the compiler by specifying at least -O3 optimization level together with -qfloat=hsflt. However, the hsflt option can be unsafe in some circumstances, see 8.2, “Recommended Compiler Options” on page 112.

Since divides are very costly, any loop that divides by the same value more than once can be easily optimized by taking the reciprocal of the value and then multiplying by the reciprocal, as in this example:

```
   Untuned                  Tuned
   -------                  -----
DO I=1,N                 DO I=1,N
  A(I)=B(I)/C(I)           OC=1.0/C(I)
  P(I)=Q(I)/C(I)           A(I)=B(I)*OC
ENDDO                      P(I)=Q(I)*OC
                         ENDDO
```

The following example shows that a similar trick can be done even when two (or more) different divisors are used:

```
   Untuned                  Tuned
   -------                  -----
DO I=1,N                 DO I=1,N
  A(I)=B(I)/C(I)           OCD=1.0/( C(I)*D(I) )
```

```
      P(I)=Q(I)/D(I)              A(I)=B(I)*OCD*D(I)
    ENDDO                         P(I)=Q(I)*OCD*C(I)
                                ENDDO
```

Here, two divides have been replaced by one divide and five multiplies - still a considerable saving in cycles.

8.4 Key Aspects of POWER3 (Model 260) Architecture

This section covers only those aspects of the architecture that are of the highest direct relevance to the performance of floating-point-intensive programs. More details on Model 260 architecture are given in Chapter 2, “The POWER3 Processor” on page 7.

- L1 data cache,
- L2 data cache,
- translation lookaside buffer (TLB), and
- the superscalar floating point units (FPUs).

Other aspects of the architecture, such as the instruction cache, can be significant for some programs but generally much less so than those considered here.

The way in which you can tune code to take best advantage of the architecture is the subject of the next section.

8.4.1 The POWER3 (Model 260) Level 1 Data Cache

Memory is buffered by a high speed data cache of 64 KB. Its structure and the effect of this on performance is considered in the following two subsections.

8.4.1.1 Structure of the L1 Data Cache

The structure of the Model 260 L1 cache is significantly different from (and, on the whole, better than) that of the POWER2 cache. There are three concepts which are key to understanding the cache:

- Cache lines

  Each line is 128 bytes long and is the basic unit of transfer between main memory and cache.

- Set-associativity

  This is one of the main POWER3/POWER2 differences: the POWER2 cache is 4-way set associative; the POWER3 cache is 128-way.

- Cache line prefetch

  This important feature of POWER3 is not present on POWER2.

These concepts will now be explained in detail.

Cache Lines
Conceptually, memory is sectioned into contiguous 128-byte lines, each one starting on a cache-line boundary whose hardware address is a multiple of 128. The cache is similarly sectioned and all data transfer between cache and memory is in units of these lines.

If, for example, a particular floating point number is required to be copied (loaded) into a floating point register so that computation may be done with it, then the whole cache line containing that number is transferred from memory to cache.

Set Associativity
The L1 data cache is mapped onto memory, as shown in Figure 21, which shows the L1 cache on POWER2, and Figure 22, which shows the same for POWER3. Each column in one of the diagrams is called a *congruence class,* and any particular line from memory may only be loaded into a cache line in the same congruence class and for POWER2 into one of only 4 locations; for POWER3 into one of 128 locations.

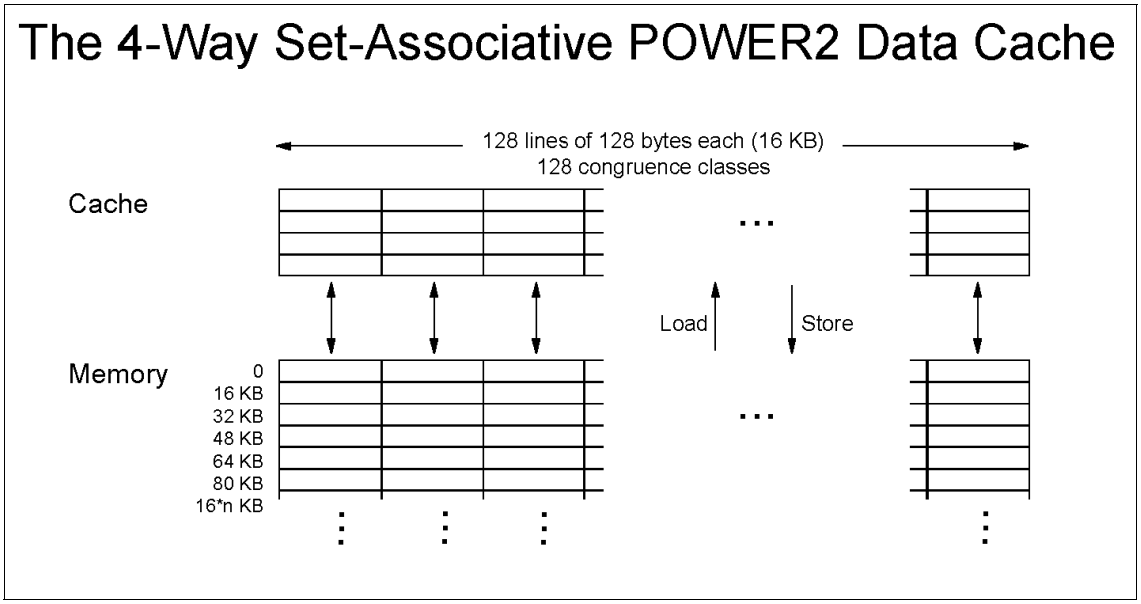


Figure 21. The 4-Way Set-Associative POWER2 Data Cache

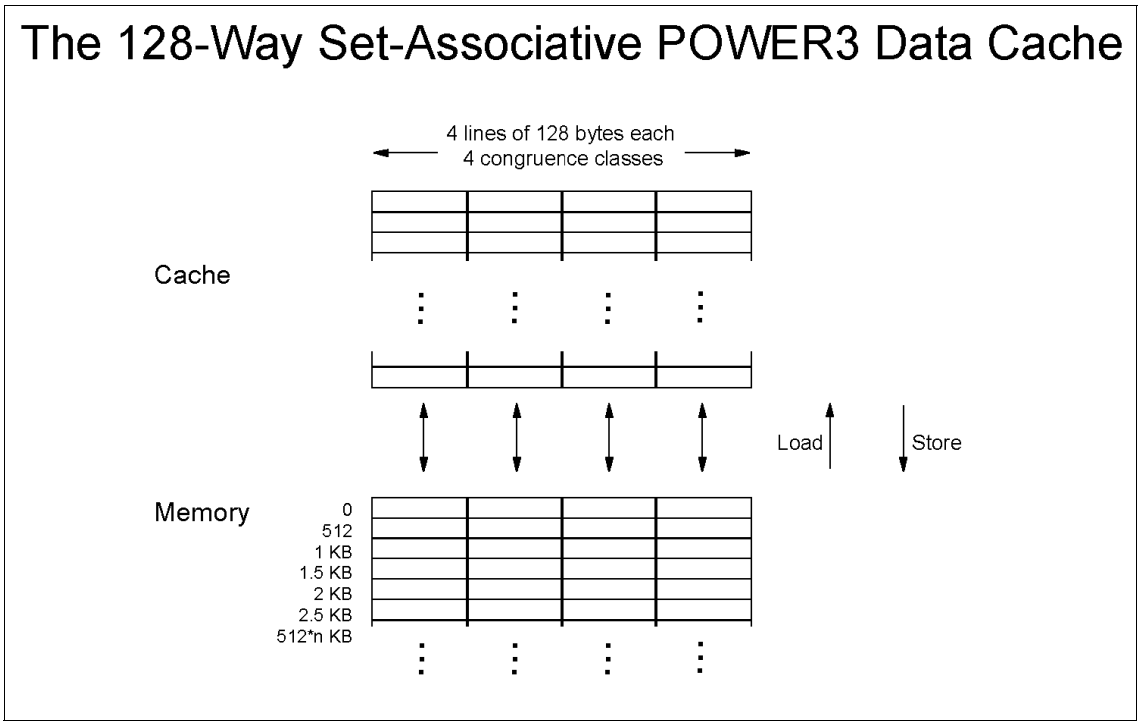


Figure 22. The 128-Way Set-Associative POWER3 Data Cache

Generally, when a new line from memory is loaded into the cache, existing data must be displaced. If the previous contents of the line have been modified, the line must be stored back into memory. The algorithm used by the hardware for selecting which cache line to use is an approximation of Least Recently Used on POWER2 and is round-robin on POWER3.

The set associative structure of the cache can lead to a reduction in its effective size. Suppose successive data elements are being processed that are regularly spaced in memory (that is with a constant *stride*). With the POWER2 cache, the worst case is when the stride is exactly 16 KB or a multiple of 16 KB. In this case, all elements will lie in the same congruence class and the effective cache size will be only *four* lines. This effect happens with strides that are a multiple of a power of 2.

The POWER3 cache, with its much greater degree of set associativity, is much less susceptible to this problem than the POWER2 cache. Strides of multiples of 1024 bytes will cause all the data to be in the same congruence class but will only cause a reduction in apparent cache size of a factor of 4.

Odd multiples of 512 will halve the effective size. This is minor compared with the possible reduction by a factor of 128 on POWER2. See also 7.4, “Large Stride” on page 102.

8.4.1.2 Effects of L1 Cache Misses on Performance

You can estimate the effects of cache misses on performance with the following approximate rules for POWER3 (Model 260):

- A load instruction (from memory to a floating point register) takes one cycle if the data is in the cache.

  (On a 200 MHz (cycles per second) Model 260, a cycle is 5 ns.)
- If the data is in L2 cache, it takes six or seven cycles.
- If the data is in memory only, it takes about 36 cycles. That is, the cost of a cache miss to memory is 35 additional cycles.
- Following the initial 35 cycle delay, forward sequentially accessed items in the same cache line may be loaded in a further one cycle each.
- The same timing applies to storing data from registers into memory. If the store is into a previously unreferenced line, the complete line must be fetched from memory first before the new value can be stored into it.
- If a cache line is overwritten by newly accessed data, then, if the data from the old line is needed again, it must be reloaded and another cache miss taken.

Cache Line Prefetch

Because of the relatively large number of cycles needed for a cache miss, POWER3 has a mechanism for mitigating the performance impact for sequentially accessed data. For up to four streams of data, the hardware attempts to detect sequential access and initiates the loading of subsequent lines in parallel, so they stream into the cache behind the first line without waiting for the miss to occur. The beneficial effects of this on performance are discussed in 7.3.1, “Copy” on page 95.

8.4.2 The POWER3 (Model 260) Level 2 Data Cache

On the Model 260, the L2 data cache is 4 MB in size. For numerically-intensive applications, it is likely to be of less importance than the L1 cache. The following points summarize the operation of the L2 cache:

- Data in the L1 cache may or may not also be in L2.
- Data loaded into L1 by the pre-fetch mechanism does not go into L2.
- Data loaded into L1 other than by pre-fetch (that is as a result of an L1 cache-miss) also goes into L2.

- An L1 cache miss costs only about six or seven cycles if the data is in the L2 cache compared with 35 cycles if it is not.

8.4.3 The Translation Lookaside Buffer (TLB)

Virtual storage constitutes the addressable memory space used by the AIX system. This linear contiguous address space is mapped, by a combination of hardware and software, onto the hardware memory (*real storage*) of the computer and onto paging spaces held on disk. If the amount of memory used by the system is greater than can be held in real storage, the paging mechanism of AIX will automatically cause transfers, as needed, between real storage and disk in units of 4 KB pages.

It is important to understand that the TLB has *nothing* to do with paging. As will be explained, TLB misses can and do occur with pages that are already in real storage.

For a 1 MB subset of pages in real storage, the translation lookaside buffer (TLB) holds the correspondence between virtual storage addresses and real storage addresses.

If the address of a page is held in the TLB, no additional delay occurs when data within the page is accessed. Otherwise, a TLB miss occurs. (An L1 cache miss may or may not occur at the same time.) The virtual/real address of the page is then resolved using the page and segment tables (held in real memory) and this is placed in the TLB, overwriting an existing entry on a least recently used basis.

The cost of a TLB miss varies between about 25 cycles if the relevant parts of the page and segment tables are in L2 cache, to possibly hundreds of cycles in unfavorable cases.

The POWER3 TLB has a total 256 entries, and therefore, addresses only 1 MB of memory. This is the same as on POWER2.

The TLB is 2-way set associative. Therefore, an application accessing data with a stride of exactly 512 KB (or a multiple) would see a TLB with only two entries. Arguably, such a stride would be even less likely to occur in practice than strides which can cause trouble with the POWER2 data cache.

8.4.4 The Superscalar Floating Point Units and Peak Megaflops

The peak rate of a single 200 MHz Model 260 processor is 800 MFLOPS (that is, four flops per machine cycle).

Approaching this rate in practice is only possible if delays due to the caches and the TLB have been eliminated. This section is therefore about what is normally called *in cache performance* but really should be *in L1 cache and TLB performance*. (It is possible to construct programs that operate in L1 cache but out of TLB.)

Fixed point (integer) arithmetic is done by separate fixed point units. Although some applications (such as signal processing) make extensive use of integer arithmetic, this is not considered in detail here.

8.4.4.1 FPU Performance Guide

The following key facts summarize the way the FPUs perform:

- A single Model 260 processor has two FPUs (connected to a single L1 cache) that can operate independently in parallel.
- The two FPUs see only floating point *registers*. There are 32 *architected* registers plus 24 *rename* registers that may substitute for an architected register through a hardware process known as *renaming*. These 56 registers serve both FPUs. They all have 64 bits.
- Floating point computation is carried out only with data in these registers.
- Data is copied into the registers from the L1 cache (loaded) and copied back to the L1 cache (stored) by two load/store units. (This is different from POWER2 architecture where floating point load/stores were done by the fixed point unit.)
- For in cache (and in TLB) data, a load or store of one floating point double precision (REAL*8) variable takes one cycle. (On POWER2, it was possible to load a quad-word (two adjacent double precision variables) in one cycle.)
- The load/store units operate independently, except that two stores cannot take place in one cycle. Two loads, or a load and a store, can take place in one cycle.
- Single precision (REAL*4) variables are loaded into separate registers (using only half their capacity) and each load takes one cycle as with double precision.
- The basic computational floating point instruction is a double precision multiply add, with variants multiply/subtract, negative multiply/add, and negative multiply/subtract. There are also single precision variants in POWER3 architecture (unlike POWER2).
- A single add, subtract, or multiply (not divide) is done using the same hardware as a multiply/add and takes the same amount of time. A

multiply/add counts as two floating point operations, so that, for example, a program doing only additions might run at half the megaflops rate of one doing alternate multiplies and adds.

- The assembler acronym for the double precision floating-point multiply/add is FMA. This term will be used extensively as a shorthand for any of the variants of this basic floating point instruction.
- The computational part of an FMA takes three or four cycles.
- The worst case would be a sequence of wholly dependent 4-cycle FMAs (where a result of one FMA is needed by the next) where only one of the FPUs would be active. This would run at the rate of one FMA per four cycles, as shown in the upper part of Figure 23. If there were two independent streams of dependent FMAs, this could use both FPUs.

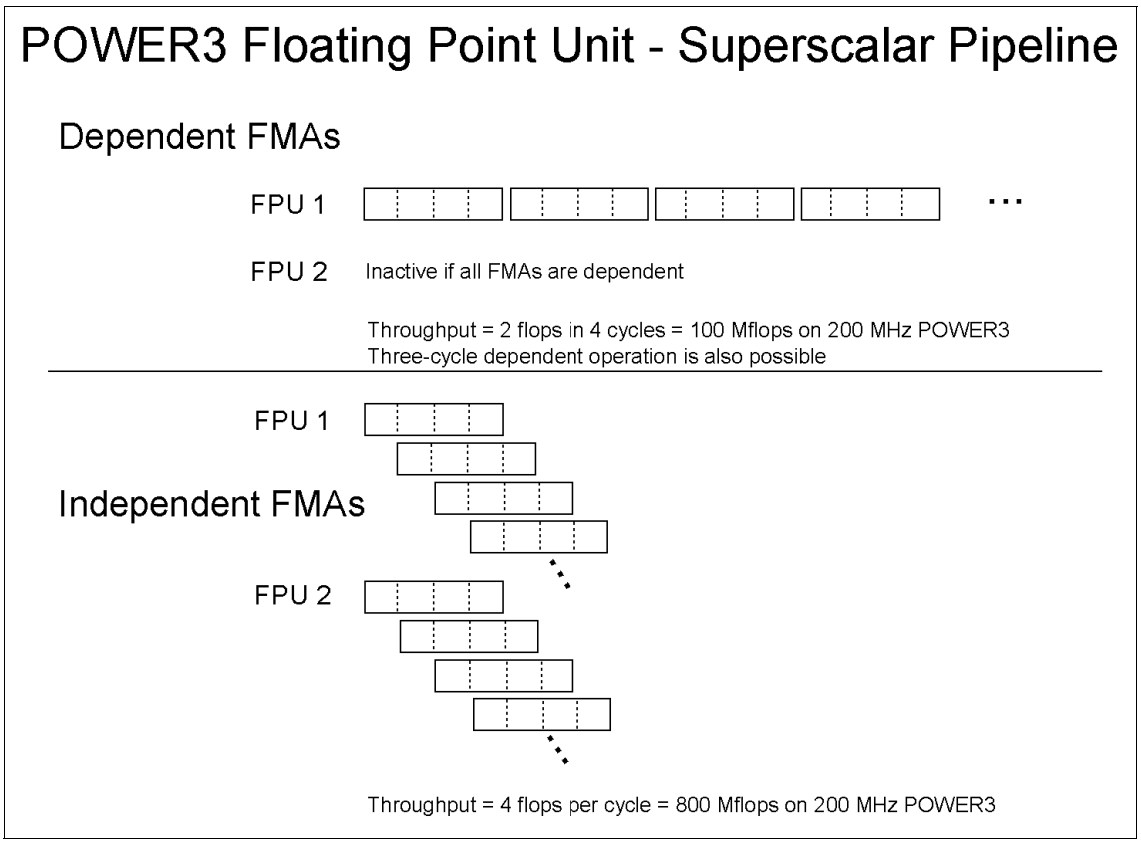


Figure 23. POWER3 Floating Point Unit - Superscalar Pipeline

- A sequence of *independent* FMAs, however, can be pipelined as shown in the lower part of Figure 23, and the throughput can then approach the peak rate of two FMAs per cycle.
- Divides are very costly and take about 18 cycles. Divides cannot be pipelined (either with another divide or with FMAs).
- A fundamental aspect of RISC architecture is that the functional units can run independently. Therefore, FMAs can run in parallel with load/stores and other functions.

8.4.4.2 Conditions for Approaching Peak Megaflops

When considering a numerically intensive loop, the following applies to the instruction stream within the loop:

- Operate solely within L1 cache and TLB.
- No divides (or square roots or function calls and so on).
- Multiplies must be paired with adds or subtracts so that each FMA is two flops.
- FMAs must be independent (and at least eight in number to keep two pipes of depth four going).
- The loop should be *FMA-bound*. That is, cycles needed for instructions other than FMAs (mainly load/stores) should be less than that needed for FMAs so that they can be overlapped with FMAs and effectively hidden. In principle, they could be equal to the FMA cycles, but, in practice, peak performance is approached most easily if they are less.

8.5 Tuning for Floating Point Performance on POWER3 (Model 260)

Tuning strategy can be summarized as follows and should be done in the following order:

1. *Avoid the negative.*

   Code so as to avoid cache and TLB misses.

2. *Exploit the positive.*

   Code so as to achieve pipelined FMA operation in the FPUs.

The techniques for achieving these two things are quite different and are discussed in the two sections which follow an introductory section on the automatic optimization obtainable from the compiler.

8.5.1 Letting the Compiler Do the Tuning

The ability of the compiler to optimize (or, effectively, *tune*) untuned code is improving with each new release, and it is becoming increasingly difficult to find *simple* examples of loops that require hand tuning to perform well.

Advice about which techniques should be done by hand is broadly:

1. If the compiler does it for you at -O3, do not bother.
2. If the compiler needs -O4 (or -qhot) to do it, probably it is worth doing yourself - for two reasons:
   - -qhot may slow down other more complex loops in the same routine.
   - The performance characteristics of the hand tuned code will be stable, that is, not dependent on the way in which the advanced capabilities of the compiler vary from release to release.
3. If the compiler will not do it at all, you've no option.

The remaining two sections include a series of case studies. The purpose of these is both to explain the principles behind tuning and to provide examples of how to go about it. In purely explanatory cases (where it is not worth hand tuning because the compiler does it well at -O3), the behavior of untuned and tuned code is generally illustrated by using the -O2 optimization level.

8.5.2 Getting and Understanding an Object Code Listing

Most tuning can be done without ever looking at an object code listing generated by the compiler but, often, it is essential for understanding why a particular tuning action does not seem to be working as expected.

Understanding just enough about object code to make sense of floating point-intensive loops is quite easy and well worth while. (It is, incidentally, possible for the compiler, to produce an assembler language source file by using the -S flag. This may then be edited and re-assembled by the compiler. No further discussion of this is included here.)

To generate a listing, compile with -qsource -qlist. The listing will then be found in the.lst file. If you compile at optimization level -O2, then the object code for the loop should map on to the Fortran source directly and be easily understandable. An extract from such a listing follows. At higher optimization levels, the listing will be more complex and difficult to understand.

```
Extract from sample.lst generated with the command
xlf -c -qsource -qlist -O2 sample.f
---------------------------------
```

```
>>>>> SOURCE SECTION <<<<<
          1 |      SUBROUTINE SAMPLE(A,B,C,S,N)
          2 |      IMPLICIT REAL*8(A-H,O-Z)
          3 |      DIMENSION A(N),B(N),C(N)
          4 |      S=0.0D0
          5 |      DO I=1,N
          6 |        A(I)=X/C(I)
          7 |        S=S + A(I)*B(I)
          8 |      ENDDO
          9 |      END

>>>>> OBJECT SECTION <<<<<
            .
            .
 5|                              CL.0:
 6| 000030 lfdu     CC450008   1     LFDU     fp2,gr5=c(gr5,8)
 6| 000034 fdiv     FC411024  15     DFL      fp2=fp1,fp2,fcr
 7| 000038 lfdu     CC640008   0     LFDU     fp3,gr4=b(gr4,8)
 6| 00003C stfdu    DC430008   0     STFDU    gr3,a(gr3,8)=fp2
 7| 000040 fmadd    FC0200FA   3     FMA      fp0=fp0,fp2,fp3,fcr
 8| 000044 bc       4200FFEC   0     BCT      ctr=CL.0,
 7| 000048 stfd     D8060000   0     STFL     s(gr6,0)=fp0
```

The following points explain how to use the listing:

- The object code shown is just the subset of the complete listing that includes the DO-loop. To locate the loop:

  1. Note the numbers of the source statements that comprise the loop (5 - 8 in this case).
  2. Look for a BCT (branch on count) statement that loops back to a label (CL.0 in this case) where the numbers of the included object code statements is the same as in the Fortran source. Note that references to statements within the source loop are often found also outside the object code loop. These should be ignored: you need to find the BCT branching back to a label.

     Beware also that at -O3 and above, you may sometimes find a BCT at the end of a tidy-up loop as well as the main loop.

- The first statement in the loop is an LFDU statement. LFD means load float double. The U indicates that the address registers are automatically updated for the next loop iteration. This statement loads C(I) into fp2 (floating point register 2). You can tell it is an element of C that is being loaded from the c(gr5,8) part of the statement.

- The number to the left of the mnemonic (1 against the LFDU, 15 against the following DFL) is the compiler's estimate of the number of cycles needed for the statement (within a single FPU). A zero means that the instruction can be overlapped with previously listed instructions.
- Then comes a DFL (divide float). This computes X/C(I). Since X is a scalar, it is loaded once before the loop starts (into fp1) and does not appear inside the loop.
- Then comes another LFDU (loading B(I)), followed by an STFDU (store float double with update), which stores the result of X/C(I) into A(I).
- Then comes the FMA (floating point multiply/add) that computes S+A(I)*B(I). Note that S is neither loaded nor stored within the loop. It was loaded into fp0 before the loop and can be seen to be stored immediately after the final BCT of the loop.
- Note also that it was not necessary to load A(I) for this statement because the value was already present in fp2.

8.5.3 Tuning for the L1 Cache

Generally speaking, if you successfully tune so as to work in the L1 cache, you will also be working in the TLB. This is by no means always true but a discussion of tuning specifically for the TLB is beyond the scope of this publication.

There are two basic techniques for tuning for the L1 cache:

1. Stride minimization - stride 1 (sequential processing) being the goal, and
2. Blocking (also known as strip mining), whereby data is processed in blocks that fit in the cache.

Of these two, stride minimization is by far the most important.

8.5.3.1 Stride Minimization: Case Study T1

The *stride* of an array in a Fortran DO-loop refers to the way in which the referenced array elements are laid out in memory and is equal to the difference in address of successive elements. For indirectly addressed data, stride may be variable. For negatively incremented loops, stride is negative.

While stride can be measured in bytes, it is more usual to use units equal to the size of a data item. Thus, with double precision data, each element is 8 bytes long, so that a stride of 3 would be 24 bytes.

The following example illustrates stride. In understanding the examples with nested loops and multi-dimensional arrays, remember that:

1. It is the inner loop that determines the stride (assuming the compiler does not invert the loops).
2. Fortran arrays are laid out in memory in column major order, so that stride 1 processing is obtained by varying the *left* most subscript most rapidly.

```
DO I=1,N
   A(I).......     Stride 1
   A(I+4).....     Stride 1
   A(2*I).....     Stride 2

 DO I=1,N,3
   A(I).......     Stride 3

 DO I=N,1,-2
   A(2*I)......   Stride -4


 DIMENSION A(100,50)
      .
 DO I=1,L            Triply nested loop
   DO J=1,M
     DO K=1,N
       A(K,J)..    Stride 1
       A(J,K)..    Stride 100
       A(J,I)..    No stride. A(J,I) is a scalar (a single
                    value) in the inner loop since J and I
                    stay constant.
```

The effect of stride on performance is clearly shown by the following case study.

Case Study T1

```
       Untuned                        Tuned
       -------                        -----
DIMENSION A(N,N),B(N,N),C(N,N)   DIMENSION A(N,N),B(N,N),C(N,N)
     .                                .
DO I=1,N                         DO J=1,N
  DO J=1,N                         DO I=1,N
    C(I,J)=C(I,J)+A(I,J)*B(I,J)      C(I,J)=C(I,J)+A(I,J)*B(I,J)
  ENDDO                            ENDDO
ENDDO                            ENDDO

    Stride N on inner loop          Stride 1 on inner loop
    ----------------------          ----------------------
```

Table 17 shows the way in which performance varies as N is increased for compiler optimization levels 2, 3, with and without `-qhot`. The maximum theoretical in-cache performance for this loop is 200 MFLOPS.

Table 17. Case Study T1: Performance of Tuned and Untuned Code

N	20	50	100	150	200	300	500	1000
Untuned (MFLOPS)								
-O2	158	141	116	65.6	9.7	4.51	2.37	1.81
-O3	131	177	108	67.2	11.4	4.59	2.50	1.83
-O3 -qhot	154	147	146	143	117	103	88.6	80.3
-O2 -qhot	160	123	142	131	97.3	102	88.8	77.4
Tuned (MFLOPS)								
-O2	158	123	141	130	96.9	101	84.7	77.2
-O3	171	147	147	147	115	102	89.1	79.4
-O3 -qhot	154	147	143	150	119	102	88.6	80.2
-O2 -qhot	159	123	124	116	90.9	87.8	86.3	77.0

The timing program used to obtain these numbers executed this code within an outer iteration loop. Therefore, for small arrays (small values of N), these figures represent in-cache performance.

The following conclusions may be drawn:

1. The effect of executing with bad stride can be very serious.

   This is shown in the untuned -O2 and -O3 lines of the table where the performance with 1000x1000 arrays is 88 times worse than the in-cache performance. This is much worse than the 35x degradation that you would expect simply if every number referenced were a cache miss. TLB misses, in addition to cache misses, are the explanation.

2. Even with stride 1 processing, performance degrades (by somewhat more than a factor of 2) as the arrays become large.

3. As expected (since loop interchange is a function of -qhot), -O3 does not fix the problem.

4. With this simple example, -qhot (even when used only with -O2) fixes the problem essentially perfectly. (The bottom two lines for untuned code are more or less the same as for tuned code.)

Should you hand tune?

Yes.

Getting the DO-loops in the correct order so that the inner loop is stride 1 is such a basic easy tuning action that it really should be done. Not only does it mean that you are not forced to use -qhot (which might slow down other loops) but the code will also run much faster on other cache-based hardware platforms that do not boast such a powerful compiler as XL Fortran.

Of course, it is not always possible to structure the code so that all arrays are accessed stride 1. In that case, blocking may be essential to avoid serious performance problems.

8.5.3.2 Blocking: Case Study T2

The idea behind blocking is simple: process the data in small enough chunks that they fit in the cache.

There are two reasons for blocking large arrays:

1. If some arrays must be processed with a large stride.
2. If the data values are used many times.

   In this case, you want to do as many computations as possible with the data while it is in the cache and before it is flushed out by more recently accessed lines. This applies even if everything is stride 1, although the benefit here is limited to recovering the stride 1 degradation factor of about a factor of 2.

If neither of the above conditions is true, there is no point in blocking.

The following code illustrates the point:

```
All stride 1. No data re-use. No point in blocking.
---------------------------------------------------
   DO J=1,N
     DO I=1,N
       S = S + A(I,J)*B(I,J)
     ENDDO
   ENDDO

Unavoidable bad stride. No data re-use. (Case Study T2.)
```

```
-----------------------------------------------------------
      DO J=1,N
        DO I=1,N
          S = S + A(I,J)*B(J,I)
        ENDDO
      ENDDO

   All stride 1. Much data re-use. Matrix Multiply transpose.
   ----------------------------------------------------------
      DO I=1,N
        DO J=1,N
          DO K=1,N
            C(J,I) = C(J,I) + A(K,J)*B(K,I)
          ENDDO
        ENDDO
      ENDDO

   Unavoidable bad stride and much data re-use. Matrix Multiply.
   -------------------------------------------------------------
      DO I=1,N
        DO J=1,N
          DO K=1,N
            C(J,I) = C(J,I) + A(J,K)*B(K,I)
          ENDDO
        ENDDO
      ENDDO
```

The last two cases above are essentially the same as the first two except for the extra loop on the outside. This causes each data value to be used N times rather than once.

Matrix multiply is discussed in detail in 9.3, “Case Study: Matrix Multiplication” on page 151.

Case Study T2

```
Untuned code
------------
DO J=1,N
  DO I=1,N
    S = S + A(I,J)*B(J,I)
  ENDDO
ENDDO

Tuned (blocked) code
--------------------
DO JJ=1,N,NB
```

```
      DO II=1,N,NB
        DO J=JJ,MIN(N,JJ+NB-1)
          DO I=II,MIN(N,II+NB-1)
            S = S + A(I,J)*B(J,I)
          ENDDO
        ENDDO
      ENDDO
    ENDDO
```

Table 18 shows the performance of this code. The maximum theoretical in-cache performance of this loop is 400 MFLOPS on a Model 260.

Table 18. Case Study T2: Performance of Untuned and Tuned Code

N	40	80	320	800	1600
Untuned (MFLOPS)					
-O2	134	121	22.0	7.23	5.31
-O3	303	208	20.5	7.45	5.35
-O3 -qhot	333	174	64.3	50.8	46.4
-O2 -qhot	132	112	22.3	7.16	5.33
Tuned (MFLOPS)					
-O2	129	89.3	49.4	38.1	34.9
-O3	129	96.7	49.7	38.6	35.3
-O3 -qhot	303	182	65.8	48.6	48.6
-O2 -qhot	129	105	54.5	40.1	35.6

As before, the timing program used to obtain these numbers executed this code within an outer iteration loop. The small values of N, therefore, represent in-cache performance. They are included only to show the dramatic effect of the bad stride as N increases.

The following conclusions may be drawn. In all cases, *look only at the N=1600 column*, since the whole point of this loop is what happens with big matrices and hence big strides.

1. Again, the compiler is capable of tuning this loop, but this time, both -O3 and -qhot are needed (or -O4) before the untuned code will perform reasonably.
2. The tuned (blocked) code performs reasonably with all compiler options.

3. Even in the best case, the degradation compared with small matrices is substantial, the rate being about a factor of 6 below in-cache performance.

   This is a direct consequence of the fact that there is no data re-use. Before the loops are executed, *none* of the data is in cache or addressed by the TLB. Therefore, both cache and TLB misses are taken as the arrays are used. What the blocking does is to prevent these misses being unnecessarily taken multiple times.

With data re-use, blocking is considerably more successful, as in 9.3, “Case Study: Matrix Multiplication” on page 151. Here, rates in excess of 600 MFLOPS are achieved even for large matrices.

Should you hand-tune?

Probably.

If your code is performance-critical and is of the form that needs blocking, then either you or the compiler must block it. Check whether the compiler is doing it for you (that is, does -O4 give a substantial boost to the loop over -O3). If the compiler is doing it, well and good. However, make sure that -O4 is not slowing down other loops in the same routine.

But the real benefit of hand tuning is that you will have a much more stable situation since the performance of the code will not be dependent on a particular compiler option whose characteristics might change with future compiler releases.

8.5.4 Tuning for the CPU

This section discusses tuning code that is running in L1 data cache and in TLB. The examples given here should be regarded as kernels which, for in-cache performance would be embedded in outer loops in a real code. For example, Case Study T15 is a 2-D kernel which, when embedded in an outer loop, gives matrix multiply coding.

For present purposes, these kernels are simply imbedded in an outer iteration loop to show in-cache performance. All timings are for small matrices that fit in the cache.

8.5.4.1 Calculating Theoretical Performance for Simple Loops

For a simple nest of loops (not including function references), it is easy to calculate the theoretical maximum in-cache performance on a Model 260.

First, let us do it assuming that the compiler does no loop rearranging or unrolling - what happens in practice with -O2:

1. Look only at the inner loop.
2. Count the number of loads (array elements to the right of an equals sign) and stores (array elements to the left of an equals sign) in the loop. Count only array elements that *depend* on the DO-loop variable. Do *not* count scalars. For example,

```
DO I=1,N
  DO J=1,N
    A(J) = A(J) * ( DEF + B(I)*C(J) )
    X(I) = X(I) + PQR*C(J)
  ENDDO
ENDDO
```

   Count one store (for A(J)) and two loads (for A(J) and C(J)) for this loop. B(I) and X(I) are scalars since they do not depend on the inner loop index, J. DEF and PQR are explicit scalars. And C(J), although referenced twice, only needs to be loaded once. This loop, therefore, needs 3 load/stores.
3. Count the number of FMAs needed. Count one for each +* or -* pair you can find plus one for all unpaired + or - or *. The above loop would need 3 FMAs (there is one unpaired *).
4. Count the number of divides.
5. Count the number of flops (arithmetic operators ignoring brackets). There are 5 in the above loop.

Then:

- Cycles for loads/stores simply equals the number of load/stores.
- Cycles for computation equals the number of FMAs plus about 18 times the number of divides.

Since, in general, load/stores can be overlapped with computation, the cycles for the loop is whichever of these is greater. If cycles for load/stores is greater than for FMAs, the loop is load/store-bound; if cycles for FMAs is greater, it is FMA-bound; if they are equal, it is balanced.

There are two CPUs that operate independently at 200 MHz, so the peak theoretical megaflops rate is 400*F/C, where F is the number of flops in the loop, and C is the number of cycles you estimate.

8.5.4.2 Basic In-Cache Tuning Techniques: (Case Study T3)

In practice, the performance of the loop will depend also on how successful you are (or the compiler is) at arranging for sufficient independent FMAs to be present in the loop to allow the superscalar pipeline to operate. This will be illustrated with the following case study. The primary purpose of this is explanatory, so the -O2 compiler option will be used.

Case study T3

```
DO I=1,N
  S = S + W*A(I) + X*B(I) + Y*C(I) + Z*D(I)
ENDDO

Measured speed at 200 MHz with -O2 = 133 MFLOPS
```

Theoretically, the loop is balanced: 4 load/stores and 4 FMAs needed. Since all +* operators are paired, the peak theoretical speed is 800 MFLOPS on the 200 MHz Model 260 - compared with a measured value of 133 MFLOPS at -O2.

The reason is that, at -O2 (which implies -qstrict), the compiler observes a strict left to right computation ordering. Hence the FMA that computes "...+ X*B(I)" requires as input the result of "S + W*A(I)"; that is, it is dependent on it. All FMAs in the computation are dependent, and therefore, cannot be pipelined. This is even true from one iteration to the next since the expression uses the same reduction variable, S, as was computed in the previous iteration. Furthermore, because there are never any independent FMAs, only one FPU can be active. Dependent FMAs within a single FPU can execute one every three cycles (rather than the 4 normally assumed). Therefore, one iteration of the loop takes 12 cycles. Since there are 8 flops in the loop, the theoretical rate comes down to 200*8/12=133.33 MFLOPS - as measured.

So it is necessary to introduce independent FMAs.

```
      T3A                        T3B
      ---                        ---
DO I=1,N                   DO I=1,N
    S = S + W*A(I)            S1 = S1 + W*A(I)
    S = S + X*B(I)            S2 = S2 + X*B(I)
    S = S + Y*C(I)            S3 = S3 + Y*C(I)
    S = S + Z*D(I)            S4 = S4 + Z*D(I)
ENDDO                      ENDDO
                              S = S1 + S2 + S3 + S4

Measured at -O2:  133 MFLOPS          526 MFLOPS
```

Note, first, that T3A fails to remove the dependency, since each statement in the loop uses the same reduction variable, so that it still goes at 133 MFLOPS. The T3B case, however, makes the 4 FMAs in the loop independent and gives an immediate 4-fold performance increase (at -O2). (Note that 526 MFLOPS is almost exactly 2/3 of the 800 MFLOPS peak - too exact to be a coincidence - but also too difficult to explain in detail.)

But clearly, more independent FMAs are need to keep both FPU pipelines busy. So, let's *unroll* the loop - to depth 4, say.

```
                T3C
                ---
      IODD = MOD(N,4)
      DO I=1,IODD
        S00 = S00 + W*A(I) + X*B(I) + Y*C(I) + Z*D(I)
      ENDDO
      DO I = IODD+1, N, 4
        S10 = S10 + W*A(I)
        S20 = S20 + X*B(I)
        S30 = S30 + Y*C(I)
        S40 = S40 + Z*D(I)
        S11 = S11 + W*A(I+1)
        S21 = S21 + X*B(I+1)
        S31 = S31 + Y*C(I+1)
        S41 = S41 + Z*D(I+1)
        S12 = S12 + W*A(I+2)
        S22 = S22 + X*B(I+2)
        S32 = S32 + Y*C(I+2)
        S42 = S42 + Z*D(I+2)
        S13 = S13 + W*A(I+3)
        S23 = S23 + X*B(I+3)
        S33 = S33 + Y*C(I+3)
        S43 = S43 + Z*D(I+3)
      ENDDO
      S = S10 + S20 + S30 + S40 + S11 + S21 + S31 + S41 +
     &    S12 + S22 + S32 + S42 + S13 + S23 + S33 + S43

  Measured at -O2: 714 MFLOPS
```

As can be seen, the process of unrolling to depth N involves processing the loop in batches of N iterations with an extra small loop at the start (or end) to tidy up odd iterations. Unrolling is artificial and messy. It is vital for maximum megaflops - but, fortunately, at -O3, the compiler is excellent at doing it for you.

Possible reasons why T3C is still slightly short of 800 MFLOPS are:

- Interference between loads and FMAs (the loop is balanced: it is easier to approach the peak if a loop is FMA-bound), or
- Possible interleaving conflicts on the loads.
- Overhead for tidy-up loop and summation of S.

Whatever the reason, this result reflects a general observation that it seems more difficult to approach the peak on POWER3 than it is on P2SC. Although some kernels do go in excess of 790 MFLOPS, generally, most loops struggle to exceed 700.

How does the compiler do?

Table 19 shows the performance with various compiler options and code versions:

Table 19. Performance of Case Study T3

Coding	MFLOPS at -O2	MFLOPS at -O3	MFLOPS at -O4
T3	133	133	606
T3B	526	400	714
T3C	714	714	625

Note that on the hand-tuned code, T3C, -O4 causes a slow down.

Should you hand-tune?

Well, on the original code, even -O4 only gives you 606 MFLOPS. It is disappointing that the easy hand-coding in T3B does not work well with -O3 since -O3 is usually good at unrolling. The problem here is that -O3 does not introduce additional reduction variables so that dependencies remain.

The most stable situation, as before, is to do it yourself and use -O3.

8.5.4.3 Making a Load/Store-Bound Loop FMA-Bound

First, for illustration, here is a loop that is firmly load/store-bound and *cannot* be made FMA-bound:

```
DO I = 1,N
  A(I) = B(I) + C(I)*D(I)
ENDDO
```

This needs four cycles for the load/stores. There is only one FMA; so even if this operates dependently at four cycles, it should overlap with the load/stores. Unrolling and so on might reduce the cycles needed for the

FMAs but nothing can reduce the four cycles for the load/stores. Therefore, with two flops in the loop, the peak theoretical performance is 400*2/4 = 200 MFLOPS.

Table 20 shows the measured performance. And, as can be seen, no amount of unrolling that -O4 might do can speed this up.

Table 20. Performance of Load/Store Bound Loop.

MFLOPS at -O2	MFLOPS at -O3	MFLOPS at -O4
196	196	196

Case Study T4

However, some nests of loops can be transformed so as to change the load/store - FMA balance. The following example, T3, is one that the compiler will not do, even at -O4.

```
          T4
          --
      DO I = 1,N
        DO J = 1,N
          Y(I) = Y(I) + X(J)*A(J,I)
        ENDDO
      ENDDO

Measured MFLOPS on 200 MHz at -O2      -O3      -O4
                               138      328      385
```

Note that the loop is already well-structured in that the inner loop both has stride 1 and is a sum-reduction (Y(I) is a scalar).

The theoretical calculation for the inner loop gives 400 MFLOPS (2 loads for X(J) and A(J,I) and 1 FMA - loop is load/store-bound). The -O4 option nearly achieves this and -O3 also does quite well. -O2 suffers, as before, from each iteration of J being dependent on the previous one through the scalar value Y(I).

Reversing the order of the loops would make matters worse. Then we would have 3 load/stores instead of 2. Also A(J,I) would be accessed with bad stride. For in-cache operation, this might not be a problem, but could become a serious problem if the arrays became larger.

The trick to make the loop less load/store-bound is to keep the order of the loops the same but to unroll the *outer* loop. (Tidy-up code is omitted.)

```
          T4A
          ---
      DO I = 1, N, 8
        DO J = 1, N
          Y(I  ) = Y(I  ) + X(J)*A(J,I  )
          Y(I+1) = Y(I+1) + X(J)*A(J,I+1)
          Y(I+2) = Y(I+2) + X(J)*A(J,I+2)
          Y(I+3) = Y(I+3) + X(J)*A(J,I+3)
          Y(I+4) = Y(I+4) + X(J)*A(J,I+4)
          Y(I+5) = Y(I+5) + X(J)*A(J,I+5)
          Y(I+6) = Y(I+6) + X(J)*A(J,I+6)
          Y(I+7) = Y(I+7) + X(J)*A(J,I+7)
        ENDDO
      ENDDO

Measured MFLOPS on 200 MHz at -O2      -O3       -O4
                              149      157       606
```

The point is that X(J) is now re-used seven times in the loop. Each iteration of the loop needs nine load/stores and eight FMAs for 16 flops. It is still load/store-bound but only just. Unrolling more would help slightly - it would asymptotically become balanced as the unrolling depth increases. For depth 8, as shown, theoretical performance is 400*16/9 = 711 MFLOPS.

The measured performance is shown below the code listing. With -O4, it's reasonably close to the theoretical value - but why does -O2 only go at 149 MFLOPS? There are eight independent FMAs in the loop; so there seems no reason for this. To find out why, it is necessary to study the assembler listing obtained with -qsource -qlist (from the V5.1.1 compiler at -O2):

```
       Extract from T4A.lst file, XLFV5.5 at -O2 level
       -----------------------------------------------
16|                            CL.6:
17| 000128 lfdu     CC950008   1     LFDU     fp4,gr21=c(gr21,8)
18| 00012C lfdu     CC760008   1     LFDU     fp3,gr22=c(gr22,8)
17| 000130 lfdu     CC340008   1     LFDU     fp1,gr20=b(gr20,8)
18| 000134 lfd      C8430000   1     LFL      fp2=a(gr3,0)
19| 000138 lfdu     CCD70008   1     LFDU     fp6,gr23=c(gr23,8)
20| 00013C lfdu     CD380008   1     LFDU     fp9,gr24=c(gr24,8)
19| 000140 lfd      C8A30008   1     LFL      fp5=a(gr3,8)
20| 000144 lfd      C9030010   1     LFL      fp8=a(gr3,16)
18| 000148 fmadd    FC4110FA   1     FMA      fp2=fp2,fp1,fp3,fcr
21| 00014C lfd      C8630018   1     LFL      fp3=a(gr3,24)
17| 000150 fmadd    FC01013A   1     FMA      fp0=fp0,fp1,fp4,fcr
21| 000154 lfdu     CC990008   1     LFDU     fp4,gr25=c(gr25,8)
23| 000158 lfdu     CD5B0008   1     LFDU     fp10,gr27=c(gr27,8)
22| 00015C lfdu     CCFA0008   1     LFDU     fp7,gr26=c(gr26,8)
```

```
20| 000160 fmadd    FD01427A   1     FMA      fp8=fp8,fp1,fp9,fcr
23| 000164 lfd      C9230028   1     LFL      fp9=a(gr3,40)
19| 000168 fmadd    FCA129BA   1     FMA      fp5=fp5,fp1,fp6,fcr
22| 00016C lfd      C8C30020   1     LFL      fp6=a(gr3,32)
21| 000170 fmadd    FC61193A   1     FMA      fp3=fp3,fp1,fp4,fcr
18| 000174 stfd     D8430000   1     STFL     a(gr3,0)=fp2
17| 000178 stfd     D803FFF8   1     STFL     a(gr3,-8)=fp0
24| 00017C lfdu     CC9C0008   1     LFDU     fp4,gr28=c(gr28,8)
22| 000180 fmadd    FC4131FA   1     FMA      fp2=fp6,fp1,fp7,fcr
19| 000184 stfd     D8A30008   1     STFL     a(gr3,8)=fp5
23| 000188 fmadd    FCC14ABA   1     FMA      fp6=fp9,fp1,fp10,fcr
24| 00018C lfd      C8A30030   1     LFL      fp5=a(gr3,48)
20| 000190 stfd     D9030010   1     STFL     a(gr3,16)=fp8
21| 000194 stfd     D8630018   1     STFL     a(gr3,24)=fp3
24| 000198 fmadd    FC21293A   1     FMA      fp1=fp5,fp1,fp4,fcr
23| 00019C stfd     D8C30028   1     STFL     a(gr3,40)=fp6
22| 0001A0 stfd     D8430020   1     STFL     a(gr3,32)=fp2
24| 0001A4 stfd     D8230030   1     STFL     a(gr3,48)=fp1
25| 0001A8 bc       4200FF80   0     BCT      ctr=CL.6,
```

The loop contains eight FMAs as expected. However, instead of nine loads and zero stores, there are 16 loads and eight stores. Closer inspection reveals that the eight elements of the array Y are all being stored unnecessarily - and then seven of them are being reloaded unnecessarily. When there was only one statement involving the scalar Y(I), the compiler recognized it as a scalar. Now, however, at -O2, the compiler is assuming that the Y values used by the statements may depend on the values calculated in other statements. At -O4, compiler logic is invoked that works out that this is not so. Inspection of the assembler code for -O4 reveals that the unnecessary stores have been eliminated.

To produce hand-tuned code that will perform well at -O2 requires the introduction of explicit temporary scalars:

```
          T4B
          ---
DO I = 1, N, 8
  S0 = Y(I  )
  S1 = Y(I+1)
  S2 = Y(I+2)
  S3 = Y(I+3)
  S4 = Y(I+4)
  S5 = Y(I+5)
  S6 = Y(I+6)
  S7 = Y(I+7)
  DO J = 1, N
```

```
        S0 = S0 + X(J)*A(J,I  )
        S1 = S1 + X(J)*A(J,I+1)
        S2 = S2 + X(J)*A(J,I+2)
        S3 = S3 + X(J)*A(J,I+3)
        S4 = S4 + X(J)*A(J,I+4)
        S5 = S5 + X(J)*A(J,I+5)
        S6 = S6 + X(J)*A(J,I+6)
        S7 = S7 + X(J)*A(J,I+7)
      ENDDO
      A(I  ) = S0
      A(I+1) = S1
      A(I+2) = S2
      A(I+3) = S3
      A(I+4) = S4
      A(I+5) = S5
      A(I+6) = S6
      A(I+7) = S7

    ENDDO
```

Measured MFLOPS on 200 MHz at	-O2	-O3	-O4
	645	606	588

This example illustrates a good general rule:

- To help the compiler, it is a good idea to *replace* scalar array elements or expressions with explicitly coded scalars. Note, however, that there must not be too many of them. The compiler will try to allocate a hardware register to each scalar in a loop. There are 32 architected registers and, if the compiler runs out, it will *spill* the registers to memory with a serious performance impact. The general advice to keep loops small and simple applies here.

Now -O2 is going at 645 MFLOPS - as close as we are likely to get to the theoretical limit of 711 MFLOPS. Note again that, with this hand-tuned loop, -O4 slows it down.

MxN Unrolling - Matrix Multiply

```
    DO I=1,N
      DO J=1,N
        DO K=1,N
          S = S + A(J,K)*B(K,I)
        ENDDO
      ENDDO
    ENDDO
```

The example coding shows the heart of matrix multiply coding where the scalar result element, C(I,J) has already been replaced with a temporary scalar. Code setting C(I,J) to S and back again has been omitted for clarity.

The problem with this is that the inner loop is load/store bound. There are two loads and only one FMA.

A key technique to making the loop FMA-bound instead of load/store-bound is to unroll both of the *outer* loops to relatively small depths. The following code, for the sake of illustration, shows 3x2 unrolling:

```
DO I=1,N,3
  DO J=1,N,2
    DO K=1,N
      S00 = S00 + A(J  ,K)*B(K,I  )
      S01 = S01 + A(J  ,K)*B(K,I+1)
      S02 = S02 + A(J  ,K)*B(K,I+2)
      S10 = S10 + A(J+1,K)*B(K,I  )
      S11 = S11 + A(J+1,K)*B(K,I+1)
      S12 = S12 + A(J+1,K)*B(K,I+2)
    ENDDO
  ENDDO
ENDDO
```

Note the re-use of elements of A and B. There are now five loads in the loop and six FMAs - it has become FMA-bound.

Generally, for MxN unrolling, there are (M+N) loads and (M*N) FMAs. So 2x2 unrolling is balanced and anything more is FMA-bound. To drive both of the FPU pipelines, needs 8 FMAs in the loop, so 2x4 or 3x3 would be needed. In practice, make it as high as possible (the more FMA-bound the better) consistent with not causing register spill. For this loop, 4x4 is the usual limit. See 9.3, “Case Study: Matrix Multiplication” on page 151, for more details on matrix multiply.

8.6 Some Comments on Parallel Coding for Model 260

The Model 260 is available as a 2-way SMP. Much of the discussion in this section, however, is equally applicable to SMPs other than Model 260, such as the previous PowerPC SMP models based on the 604e chip, and to possible follow-on POWER3 products with more than two processors.

To date, almost all RS/6000 parallelization work in the scientific and technical computing area has been done for the SP, because the best performance for such applications has been obtainable from POWER2 processors, and these

have not been available as SMPs. Model 260 is the first SMP processor with leading edge floating point performance, and this considerably widens the options available for parallelization.

In particular, the XL Fortran Version 5 compiler allows automatic and semi-automatic parallelization at the loop level.This is described in detail in Chapter 4, “Using the SMP Feature of XL Fortran” on page 29. With some programs, good speedups close to 2 can be obtained. However, for many, the percentage of CPU time spent in parallelizable loops is considerably less than 100 percent and speedups are disappointing.

With the distributed memory SP, parallelization at the loop level is not usually practicable because of the message passing overhead across the SP switch. To perform successfully across distributed memory nodes requires parallelization at a high level, using either explicitly coded message passing or the IBM XL Fortran HPF Compiler.

Usually, this high-level parallelization is far more thorough-going and effective than any loop-level parallelization can be. Therefore, if the work has already been done to create an SP version, it probably makes sense to run this on the 2-way Model 260 rather than use the compiler to generate loop-level SMP parallel code. There are two options to consider:

1. If coded using MPI, run it unchanged in one of the ways discussed in Chapter 6, “Message Passing Interface” on page 81.
2. Keep the same structure as in the MPI code, but, instead of running two separate processes under the control of PE, run two pthreads. This would require some recoding, but, since the parallel logic would remain the same, it may well be straightforward.

   MPI calls would be replaced with explicitly coded memory to memory copies plus thread-to-thread synchronization. It may also be possible to avoid the overhead of memory to memory copy if the logic is such that, with the addition of synchronization coding only, the threads can work with the same data areas.

The question of running an existing MPI program across multiple SMP nodes in an SP is discussed in Chapter 6, “Message Passing Interface” on page 81.

Chapter 9. Throughput Measurements

This section takes a look at the throughput obtained by running two copies of a program simultaneously, compared to a single copy of the program by itself. No parallel programming is involved.

First, a copy program, which access storage very heavily, is examined and then some more realistic user programs.

9.1 Copy Program

Figure 24 on page 148 and Figure 25 on page 148 show the aggregate rates for untuned and tuned copy program respectively, running a single copy, and then two copies simultaneously.

For lengths of less than 32 KB, where the data is in the L1 cache, the aggregate rate for two copies is almost exactly twice that for one copy.

For lengths of less than 2 MB, where the data is in the L2 cache, the aggregate rate for two copies is close to twice that for one copy.

For very long lengths, the aggregate rate for two tuned copies is only marginally improved over that for one tuned copy. This is because a single tuned copy by itself uses almost all of the memory bandwidth.

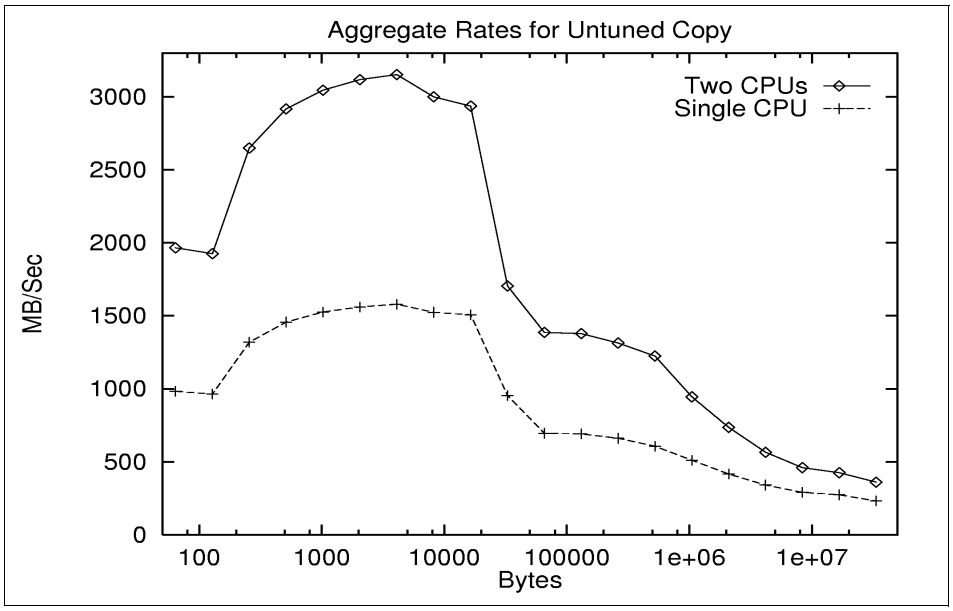


Figure 24. Aggregate Rates for Untuned Copy

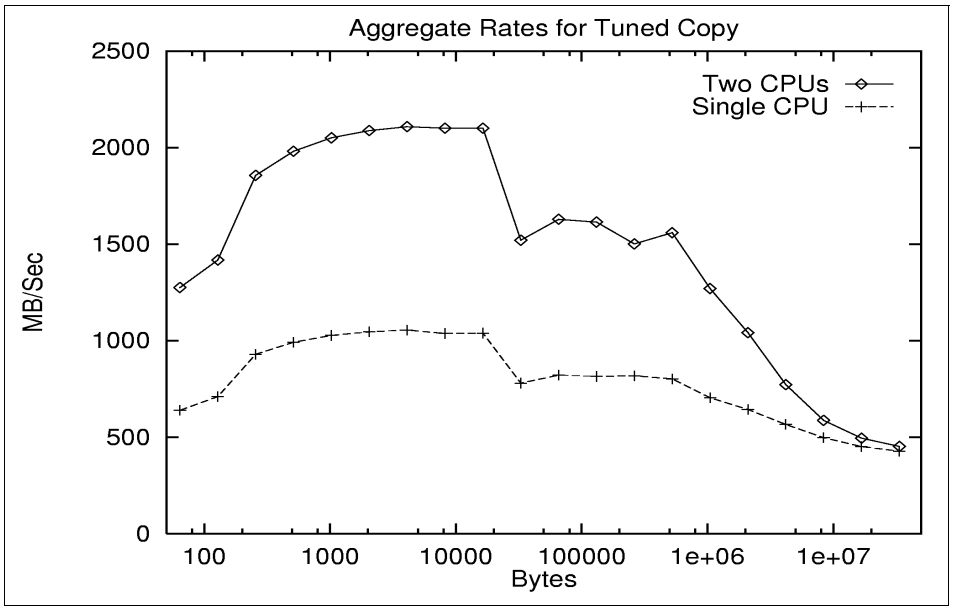


Figure 25. Aggregate Rates for Tuned Copy

The ratios of copy rates for one and two processors are shown in Table 21.

Table 21. Summary of Copy Rates

Program	Rate (MB/s)		Throughput Ratio
	Single Program	Two Programs	
16 KB Copy	1464	2936	2.00
16 KB Tuned Copy	786	1569	2.00
1 MB Copy	510	945	1.85
1 MB Tuned Copy	704	1270	1.80
8 MB Copy	291	460	1.58
8 MB Tuned Copy	497 s	587 s	1.18

The low copy rate for 8 MB represents an extreme situation which will rarely occur in practice. The throughput ratios for real programs are much better than this, and are described in the next section.

9.2 User Programs

Table 22 lists the throughput ratios for a number of real user programs.

Table 22. Real User Programs

Program	Elapsed Time (sec)		Throughput Ratio
	Single Program	Two Programs	
1. Omega-X	298	307	1.94
2. Pre-Stack Migration	184	184	2.00
3. Weather Forecast	1248	647	1.93
4. Oil Reservoir Simulator	471	480	1.96
5. RADIOSS	3476	3520	1.97
6. sPPM	99.2	101.8	1.95

Program Notes:

1. Seismic Omega-X: Uses tridiagonal complex arithmetic solver, with about 30 MB of data.

2. Pre-Stack Migration: Seismic industry program. It uses floating point to integer conversion to index into arrays. The data in this example occupied less 1 MB.
3. Weather Forecast: Details of this program are given in 10.3, “Weather Forecast Code” on page 159. The working set size was about 120 MB.
4. Oil Reservoir Simulator: Details of this program are given in 10.2, “Oil Reservoir Simulator” on page 159. The working set size was about 150 MB.
5. RADIOSS: This is a crash analysis code, with more details in 10.5, “Crash Worthiness Analysis: RADIOSS” on page 163. The problem had over 60,000 elements.
6. sPPM: ASCI benchmark program. Solves a 3D gas dynamics problem using a simplified version of PPM (Piecewise Parabolic Method). The working set size was 236 MB. For more details of the application, see:

   `www.llnl.gov/asci_benchmarks/asci/limited/ppm/sppm_readme.html`

The fact that the throughput ratio for two processors is so close to 2 for all of the above programs, even those with large working sets, is a tribute to the cache design and memory access techniques implemented in the Model 260.

9.3 Case Study: Matrix Multiplication

This section analyzes various implementations of a fundamental building-block used in dense linear algebra and elsewhere, the Level 3 BLAS routine DGEMM. DGEMM implements the following matrix multiply-and-update operation:

C = alpha*op(A)*op(B) + beta*C

where op(A) can be either A or A^T, the transpose of A. The full calling sequence for DGEMM is as follows:

```
DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

where:

`TRANSA,TRANSB`	`CHARACTER` variables with values ’`No Transpose`’ or ’`Transpose`’ (only the first character is significant and need be supplied).
`M,N,K`	The dimensions of the matrices. If `TRANSA,TRANSB` = ’`N`’, then the matrices are of dimensions `A(M,K)`, `B(K,N)`, and `C(M,N)`.
`ALPHA,BETA`	The scalar constants.
`A,B,C`	The arrays to be multiplied.

LDA, LDB, LDC — The leading dimensions of the matrices as they are declared in the calling (sub)-program.

While few *real* applications are as simple as DGEMM, with such easily understood data access patterns, an understanding of how to achieve good performance with DGEMM can be very useful when going on to tune more realistic applications. Moreover, DGEMM operations are widespread throughout dense linear algebra, such as matrix factorizations and eigenvalue problems. For example, the public-domain LAPACK library contains a wide range of routines that have been restructured where possible as blocked algorithms, so that tuned Level 3 BLAS routines, such as DGEMM, can be used to achieve good performance on a range of different architectures.

9.3.1 The Computational Kernel

To illustrate the coding required to achieve close to optimal performance on POWER3, consider the simple multiply-and-update operation:

$$C = C + A^T * B$$

(A^T is used so that operations at the inner-most level are all stride one.)

This can be implemented by the following code fragment:

```
DO I = 1, M
   DO J = 1, N
      DO L = 1, K
         C(I,J) = C(I,J) + A(L,I)*B(L,J)
      END DO
   END DO
END DO
```

Although the nesting order of the loops can be changed, the so-called *DOT* formulation (where the innermost loop is a dot-product) was used since this maps best to the POWER architecture's FMA instructions.

The speed of the innermost loop is limited by the requirement to load both A(L,I) and B(L,J) in order to perform the single FMA operation. On POWER1, with a single floating-point unit, near optimal performance could be achieved by unrolling the two outermost loops to depth two, as implemented in the following fragment (where all *tidy-up* code has been omitted for simplicity):

```
DO I = 1, M, 2
   DO J = 1, N, 2
      T11 = ZERO
      T21 = ZERO
```

```
      T12 = ZERO
      T22 = ZERO
      DO L = 1, K
         T11 = T11 + A(L,I  )*B(L,J  )
         T21 = T21 + A(L,I+1)*B(L,J  )
         T12 = T12 + A(L,I  )*B(L,J+1)
         T22 = T22 + A(L,I+1)*B(L,J+1)
      END DO
      C(I  ,J  ) = C(I  ,J  ) + T11
      C(I+1,J  ) = C(I+1,J  ) + T21
      C(I  ,J+1) = C(I  ,J+1) + T12
      C(I+1,J+1) = C(I+1,J+1) + T22
   END DO
END DO
```

With this code, each loaded element of A and B has been used twice, so that each FMA operation only requires a single load, which can be overlapped with the FMA. This code is *optimal* for POWER1, in the sense that the ratio of loads to FMA operations is precisely what the hardware supports.

For POWER2 and POWER3, with dual floating-point units, the same theoretical 1:1 ratio of loads to FMA instructions is still supported by the hardware. In practice, however, it is necessary to unroll further to facilitate the overlap of FMA operations and loads. The following code fragment allows the floating-point units to operate at peak performance for these architectures, while overlapping the loads:

```
DO I = 1, M, 4
   DO J = 1, N, 4
      T11 = ZERO
      T21 = ZERO
      ...
      ...
      T34 = ZERO
      T44 = ZERO
      DO L = 1, K
         T11 = T11 + A(L,I  )*B(L,J  )
         T21 = T21 + A(L,I+1)*B(L,J  )
         ...
         ...
         T34 = T34 + A(L,I+2)*B(L,J+3)
         T44 = T44 + A(L,I+3)*B(L,J+3)
      END DO
      C(I  ,J  ) = C(I  ,J  ) + T11
      C(I+1,J  ) = C(I+1,J  ) + T21
      ...
      ...
```

```
        C(I+2,J+3) = C(I+2,J+3) + T34
        C(I+3,J+3) = C(I+3,J+3) + T44
      END DO
    END DO
```

On a single 200 MHz POWER3 processor, in the case where all the matrices fit into cache, this code performs at close to 750 MFLOPS, out of a peak of 800 MFLOPS.

Note that the XL Fortran compiler is capable of performing many loop transformations, such as loop unrolling, in order to improve performance. If the above code is compiled with -O3 -qarch=pwr3, the compiler further unrolls the innermost loop to a depth of two. This actually reduces performance, as the code runs out of floating-point registers and needs temporary storage for register contents. On the other hand, if the code is compiled with -O2 -qarch=pwr3, it appears that, although the *optimal* numbers of FMA and LOAD instructions are generated for the innermost loop, the order of the instructions produced makes it more difficult for the processor to overlap the loads with the FMA instructions.

9.3.2 Single Processor Implementation of DGEMM

This section shows how the code fragment above may be extended to be a full implementation of DGEMM. In the discussion above, it is assumed that all the data fits into the Level 1 cache. In practice, for large matrices, this won't be the case, and it is necessary to divide the matrices into blocks, as shown in Figure 26 on page 154, and then to arrange to perform as many operations as possible on the blocks currently residing in cache, before they are flushed from cache as newer blocks are loaded.

In Figure 26, the block of C is updated with the multiplication of the blocks of A and B. There are many such updates involved in completing the matrix multiplication, and there is some choice in the order in which the updates are carried out. In the implementation described here, if the matrix were to be partitioned into 2-by-2 blocks, the block multiplication

```
C11|C12   C11|C12   A11|A12   B11|B12
--- --- = --- --- + --- --- * --- ---
C21|C22   C21|C22   A21|A22   B21|B22
```

would be carried out in the following order:

```
C11 = C11 + A11 * B11
C21 = C21 + A21 * B11
C11 = C11 + A12 * B21
C21 = C21 + A22 * B21
C12 = C12 + A11 * B12
C22 = C22 + A21 * B12
C12 = C12 + A12 * B22
C22 = C22 + A22 * B22
```

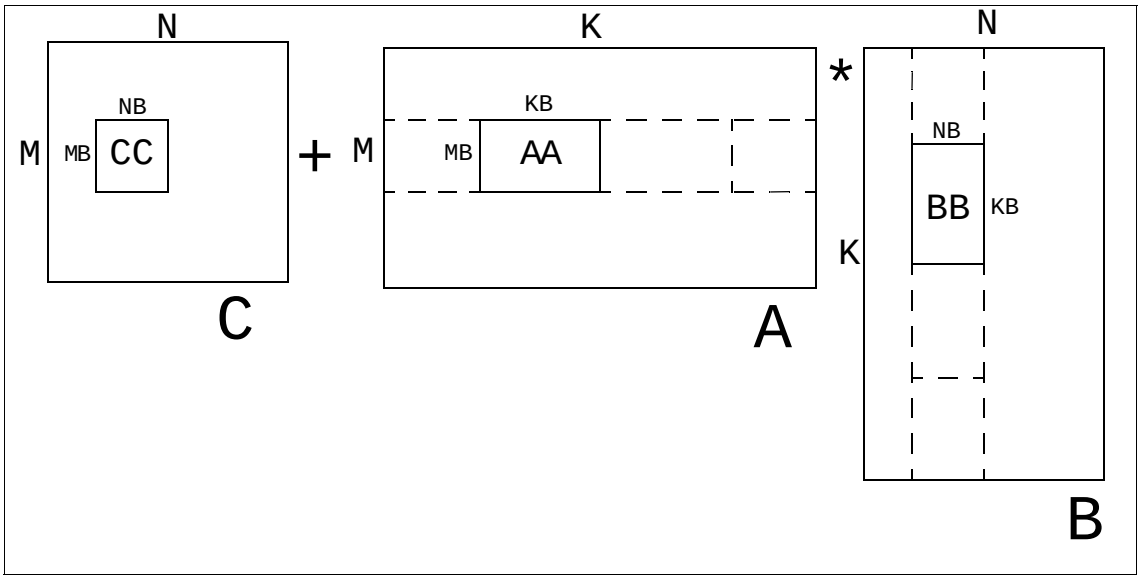


Figure 26. Block Matrix Multiplication

In order to implement the full DGEMM specification, arrange for the multiplication by the constants ALPHA and BETA. The algorithm uses the following block structure when TRANSB='No Transpose':

```
DO J = 1, N, NB
   DO L = 1, K, KB
      DO I = 1, M, MB
         TEMP1 = ALPHA*AA or ALPHA*AA^T
         CC = BETA*CC + TEMP1^T*BB
      END DO
   END DO
END DO
```

and the following when TRANSB='Transpose' where the DDOT kernel would access B along its rows with non-unit stride:

```
DO J = 1, N, NB
   DO L = 1, K, KB
      TEMP2 = ALPHA*BB^T
      DO I = 1, M, MB
         TEMP1 = AA or AA^T
         CC = TEMP1^T*TEMP2 + BETA*CC
      END DO
   END DO
END DO
```

In fact, the implementation of DGEMM described above is available in the public-domain. It is the work of Kagstrom and Ling from Umea University, Sweden, and is currently available from Netlib at the following URL:

```
http://www.netlib.org/blas/gemm_based/ssgemmbased.tgz
```

The performance of any blocked algorithm will clearly vary according to the choice of the size of the blocks. The best choice of blocksize will also depend upon the size of the problem and, to some extent, on the *shape* of the matrices. The only change made to public-domain code was to change the blocksize to have MB=32 and KB=NB=100. This is certainly not optimal, but gave good performance for a wide range of matrix dimensions.

In Figure 27 on page 157, the performance of this code when multiplying square matrices of increasing size, so that M = N = K is shown. The performance of the code described above (labelled Fortran) is compared against the implementation of DGEMM in the ESSL POWER3-enhanced library, and performs nearly as well for square matrices. It should be noted, however, that ESSL is likely to perform better across a wider range of matrix dimensions than the Fortran version described here, for example, on rectangular matrices where one dimension is relatively small.

9.3.3 Automatically Parallelized DGEMM

Compiling the Fortran version described above with the flag -qsmp doesn't yield any benefit, since the compiler is only able to parallelize the main nest of loops on the innermost DO-loop, which removes all the benefits of the four-by-four unrolling technique, and significantly reduces performance. However, in the block structure described above, it is easy to see that independent blocks of the matrix C are being updated in the `DO I = 1, M, MB` loop, and these may be performed in parallel. The code has been modified to use the INDEPENDENT compiler directive as follows:

```
      DO J = 1, N, NB
         DO L = 1, K, KB
*SMP$       INDEPENDENT
            DO I = 1, M, MB
               ...
               (multiply the blocks)
               ...
            END DO
         END DO
      END DO
```

The performance of this code on a machine with two 200 MHz processors (labelled "Fortran -qsmp") is compared against the single processor performance, and also against the ESSL SMP library. As may be seen from Figure 27, this code performs at very close to twice the speed of the single processor version.

Notice

The ESSL used in this publication is an early beta of a POWER3-enhanced library, please refer to Appendix D, "Special Notices" on page 199 regarding the performance numbers.

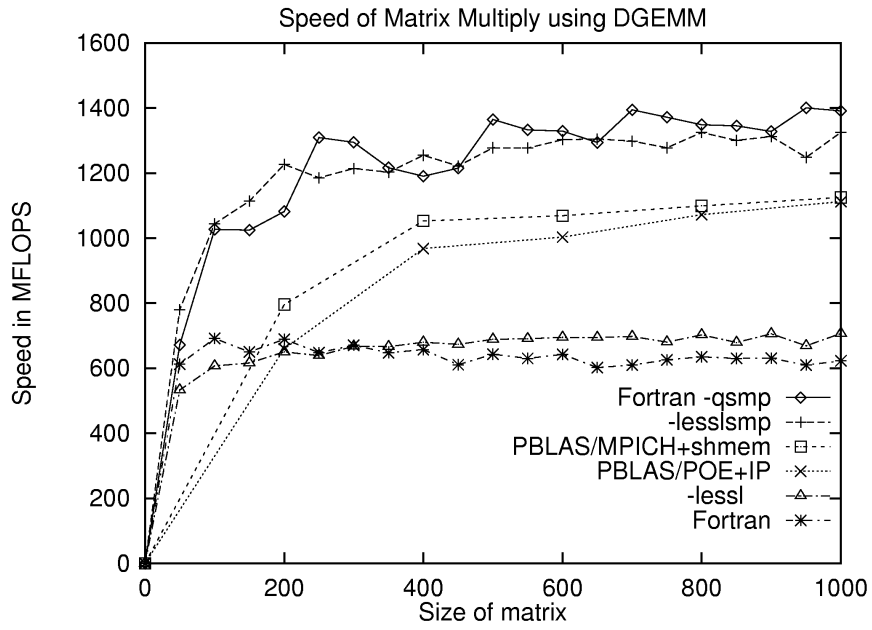


Figure 27. Performance of DGEMM

9.3.4 MPI Implementations

The performance of a distributed memory version of matrix-multiply running on the two-processor SMP machine was also examined. The routine PDGEMM from the PBLAS (Parallel BLAS) library distributed with the SCALAPACK (Scalable LAPACK) library was used. This routine uses a block-cyclic distribution of the matrices, with communications performed by the BLACS (Basic Linear Algebra Communications Subprograms), which are in turn implemented in terms of MPI calls. Within processes, blocks are multiplied using the usual DGEMM routine, and here the beta ESSL POWER3 enhanced library is used. There are a number of ways in which the matrices can be blocked. Simply use 64-by-64 blocks, since at this matrix size the performance of the ESSL routine DGEMM is already close to its peak.

The two implementations of MPI, described in Chapter 6, “Message Passing Interface” on page 81, are used. The version in POE, which currently needs to use the IP loopback interface, and the public domain version MPICH, built to use shared memory.

Figure 27 on page 157 includes timings for the PBLAS routine using MPICH (shown as PBLAS/MPICH+shmem), and the same routine using POE with IP (shown as PBLAS/POE+IP). As may be expected, the overhead of communicating matrix blocks, even on the same SMP machine, means that the MPI code is slower than the true SMP versions. But it is interesting to note that the version using IP is only slightly slower than the shared-memory version, showing that the ratio of computation to communication is relatively high in PDGEMM.

Chapter 10. Kernels, Codes, and Benchmarks

In this chapter, a discussion of common kernels, codes, and benchmarks, and how they relate to POWER3, is given.

10.1 GAMESS

The General Atomic and Molecular Electronic Structure System (GAMESS) is a general *ab initio* quantum chemistry package. This program is maintained by the members of the Gordon research group at Iowa State University.

Briefly, GAMESS can compute wavefunctions ranging from RHF, ROHF, UHF, GVB, and MCSCF, with CI and MP2 energy corrections available for some of these. Analytic gradients are available for these SCF functions, for automatic geometry optimization, transition state searches, or reaction path following. Computation of the energy Hessian permits prediction of vibrational frequencies. A variety of molecular properties, ranging from simple dipole moments to frequency dependent hyperpolarizabilities may be computed. Many basis sets are stored internally, and together with effective core potentials, all elements up to Radon may be included in molecules. Several graphics programs are available for viewing the final results. Many of the computational functions can be performed using direct techniques or in parallel on appropriate hardware.

Because GAMESS is distributed freely, the development of the code benefits from contributions from many collaborators located around the world, in academia, government laboratories, and industry. Among the many features of GAMESS, the most exciting are its enhanced performance due to a fully parallel implementation and the advantage it takes of modern graphics methods. As the POWER3 nodes for the IBM RS/6000 SP were not available at the time of this publication, this parallel feature was not tested.

A detailed description of the program is available in the following journal article:

General Atomic and Molecular Electronic Structure System, M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, J.H.Jensen, S.Koseki, N.Matsunaga, K.A.Nguyen, S.Su, T.L.Windus, M.Dupuis, J.A.Montgomery J. Comput. Chem., 14, 1347-63(1993).

The homepage of GAMESS is located at:

`http://www.msg.ameslab.gov/GAMESS/GAMESS.html`

The compiler options for the runs were:

```
xlf_r -qarch=x -O3 -qalias=noaryovrlp:nointptr:std
```

where `x` is either `pwr3` or `pwr2` depending on the platform.

The timing results are seen in the Table 23. Only the CPU time (user time is reported. The wall clock time is much higher because of I/O. The speedup being greater then megahertz scaleup is attributed to the extra load/store unit in the POWER3 processor.

Table 23. GAMESS Runs in Seconds

Dataset	P2SC-160	Model 260	Speedup
C4H4 FULLNR MCSCF	75.74	60.47	1.25
C4H6 GVB hessian	215.32	201.81	1.07
Thymine RHF gradient	336.74	270.28	1.25
Ti2H8 MP2 energy	322.58	240.92	1.39

10.2 Oil Reservoir Simulator

A number of measurements were made with an Oil Reservoir simulator, which demonstrated the performance of the Model 260 relative to the Model 590, the effect of Fortran V5 compared to V4, and the throughput capability when using two processors on the 260.

The program was compiled with -O2, and without either -qarch=pwr2 or -qarch=pwr3 so that the same executable could run on both the Model 590 and the Model 260.

Measurements were made for 20 time steps for a two-phase problem with 64 KB grid blocks. The working set size was 150 MB.

Table 24. Times for Oil Reservoir Simulator Code

System	Compiler Level	Time (secs)	Ratios
590	V4.1.0	1310	
260 (1 prog)	V4.1.0	471	260/590 = 2.78
260 (1 prog)	V5.1.1	471	V5/V4 = 1.00
260 (2 progs)	V5.1.1	480	Throughput = 1.96

The performance of the Model 260 compared to the Model 590 is not quite in the ratio of the cycle times (5 nanoseconds compared to 15 nanoseconds), but is very much in line with expectations considering the similarity of memory access times.

Fortran V5 and V4 give the same results. This is probably because this code has been highly tuned in the past, and not so much was left for the compiler to tune.

The throughput ratio indicates that there is very little memory interference between the two programs. Apparently, a lot of the data references are to the L1 and L2 caches.

10.3 Weather Forecast Code

A finite different weather forecast program was run on the Model 260 to produce a six hour forecast. The forecast included six hours of data assimilation prior to the starting the forecast. The forecast was for a local area. The the horizontal grid size was 128 by 128, and there were 19 vertical levels.

The model was set up to run with four processes using MPI. It was run with MPI in IP mode using loopback, and so automatically used both processors of the Model 260. To get a throughput comparison of one processor compared to two, the program was bound to one processor using the `bindprocessor` command immediately after the program started.

For comparison purposes, the model was also run on a 120 MHz POWER2 four node SP, a 160 MHz POWER2 single node, and with both Fortran V4 and Fortran V5. Results are shown in Table 25. The code was compiled with -O3, and -qarch as specified in the Table.

The total working set size of all 4 processes was about 120 MB.

Table 25. Times for Weather Forecast Code

System	Procs	Fortran Level	Fortran Opt	Time (secs)	Ratios
260 pwr3	1	V5.1.1	pwr3	968	
260 pwr3	2	V5.1.1	pwr3	506	`Throughput = 1.91` `pwr3/nopwr = 1.20`
160 MHz pwr2	1	V4.1.0	nopwr	1510	
260 pwr3	1	V4.1.0	noprw	1268	`260/160MHz = 1.19`

System	Procs	Fortran Level	Fortran Opt	Time (secs)	Ratios
260 pwr3	2	V4.1.0	nopwr	667	
260 pwr3	2	V5.1.1	noprw	606	V5/V4 = 1.10
120 MHz pwr2	4	V4.1.0	pwr2	480	

The following conclusions can be drawn:

- As with the oil reservoir simulator, the throughput of two processors compared to one is greater than 1.9.
- The Model 260 is 19 percent faster on this code than the 160 MHz POWER2. This is a little less than the increase in MHz (160 to 200).
- One Model 260 (with two processors) is almost equivalent to a four node 120 MHz POWER2 SP.
- Fortran V5 gives a 10 percent speedup over Fortran V4. Since this code has not been hand tuned, the compiler has a lot of opportunity to improve the performance.
- The -qarch=pwr3 option gives a 20 percent speedup. Since the code is mostly single precision, the new single precision floating point operations are extremely useful.

10.4 Computational Fluid Dynamics: FIRE

FIRE is a comprehensive computational fluid dynamics (CFD) analysis product developed by AVL List GmbH, Austria. It provides solutions to a wide variety of fluid flow and combustion applications, for example in the automotive, biomedical, aerospace, electronics, and chemical industries. FIRE applies to internal and external flow, gases, or liquids, steady state or transient. The code is able to handle time-dependent boundary conditions and moving geometries. Pre- and post-processors and solution modules are tightly integrated. The flow solver is based on a finite volume differential scheme using a modified variant of the SIMPLE algorithm. For further details, refer to the AVL's Web page:

`http://firewww.avl.co.at`

FIRE represents a class of CFD codes which are also well tuned for vector computers. The performance of such CFD codes depends strongly on memory bandwidth. The FIRE *Kernel* Benchmarks (Serial Linear Equation Solver Benchmarks) are designed by AVL for performance comparison of

different hardware platforms. Benchmark results for a lot of platforms are available through URL `http://firewww.avl.co.at/html/346.htm`. Because POWER2 machines offer high memory bandwidth, POWER2 and succeeding P2SC machines have turned out to be excellent RISC platforms for this code. The POWER3 workstation Model 260 is expected to deliver a similar performance with respect to FIRE as the fastest P2SC model, since the memory subsystem sustains a similar performance level. As pointed out in Chapter 7, “Performance and Tuning Analysis” on page 87, the L1 cache bandwidth is smaller compared to P2SC, whereas the prefetching mechanism provides a higher memory bandwidth when working out-of-cache.

In the following, results for the FIRE kernel benchmarks and a complete FIRE benchmark based on an industrial relevant input deck are presented. For this purpose, the kernel benchmark code and FIRE version 70b (patchlevel 2) has been compiled using XLF 5.1.1.0. The results for P2SC were obtained on AIX 4.2.1 using XLF 5.1.0.2.

The kernel benchmark test cases are outlined in Table 26 on page 163. In order to reach the expected performance level, the XLF flags, -qarch=pwr3 -qtune=pwr3 -O3 -qhot, are required. As verified for a single case, the -qhot flag speeds up the computation by about 15 percent. Additional flags, such as -qalias=noaryovrlp:nointptr:std -qfloat=hsflt:fold, improve the performance in this case just by less than 2 percent. For comparison, P2SC results are given in Table 27.

Table 26. FIRE Kernel Benchmark Cases

Test Case	Number of Cells
TJUNC (t-junction)	13,845
SWIRL (helical intake port)	47,312
PENT (square duct)	108,000
COJACK (water cooling jacket)	318,044
WING (airfoil)	864,000

Table 27. FIRE Kernel Benchmark Results

Execution Times [sec]	P2SC 160 MHz sequential	P2SC 160 MHz MPI: 1-way	POWER3 sequential	POWER3 SMP: 2-way
TJUNC	1.4		0.97	0.65
SWIRL	4.6	4.9	4.7	2.3
PENT	16.9	18.2	18.7	10.6

Execution Times [sec]	P2SC 160 MHz sequential	P2SC 160 MHz MPI: 1-way	POWER3 sequential	POWER3 SMP: 2-way
COJACK	61.0	64.0	67.3	45.7
WING		125.6	136.9	102.3

The following code section shows a loop which consumes a major part of the CPU time of the kernel benchmark. The loop counter ranges from one to the number of grid cells.

```
      DO 4 NC=NINTCI,NINTCF
        DIREC2(NC)=BP(NC)*DIREC1(NC)
     X            -BS(NC)*DIREC1(LCC(1,NC))
     X            -BW(NC)*DIREC1(LCC(4,NC))
     X            -BL(NC)*DIREC1(LCC(5,NC))
     X            -BN(NC)*DIREC1(LCC(3,NC))
     X            -BE(NC)*DIREC1(LCC(2,NC))
     X            -BH(NC)*DIREC1(LCC(6,NC))
    4 CONTINUE
```

The performance is determined by about three loads and one indirect addressing per multiply-subtract operation. The compiler performs a level two unrolling, limited by the number of available registers. No obvious possibilities of Fortran tuning have been found. Increasing the formal array dimensions, by a small offset, such as seven, does increase the overall performance of the kernel benchmark by few percent (not reported in Table 27). The SMP performance is clearly limited by the bandwidth of the shared memory bus. For this application, a distributed memory approach using independent processors delivers a better (almost linear) speedup, as proven for P2SC.

Next, an industrial application of 3D incompressible flow simulation with about 540000 grid cells was examined for 150 time steps. The corresponding benchmark results are shown in Table 28. Due to numerical round off errors, the accumulated sum of iterations is1566 for P2SC and 1570 for POWER3. The benchmark results support the expectation that FIRE will show a similar performance on a 200 MHz POWER3 system (1 CPU) as on P2SC.

Table 28. FIRE Benchmark Results

P2SC Model 397 (160 MHz)	POWER3 Model 260 (200 MHz)
real 23988 user 23295 system 19	real 23907 user 23879 system 9

Take Note

This benchmark was performed with a pre-GA POWER3 version of FIRE on pre-GA hardware for the purpose of an early performance evaluation. This does not imply availability of this product nor support by AVL. The performance may change as the application software and the compiler develop. For the latest Kernel Benchmark results, see the URL mentioned in this section.

10.5 Crash Worthiness Analysis: RADIOSS

Crashworthiness analysis and sheet metal forming codes represent a class of finite element codes based on explicit time step integration schemes. Compared to implicit codes they require significantly less memory. In general, crash applications are not I/O bound, as large implicit applications often are. Especially running in parallel, a significant part of the working set of a processor might fit into a large second level cache. In case of single precision analysis, the PowerPC 604e based RS/6000 Model F50 (332 MHz) has shown for several benchmark cases a similar performance level per CPU as P2SC based machines, which are usually much stronger in numerical intensive computing. Thus, POWER3 is expected to improve crash analysis performance compared to P2SC.

RADIOSS CRASH is a crashworthiness analysis code developed by Mecalog, France. RADIOSS has been widely validated and is used worldwide by automotive companies and their suppliers to study the crashworthiness behavior of their new products. For further details, refer to the RADIOSS web page `http://www.radioss.com`.

In order to get an early performance comparison between P2SC and POWER3, two industrial test cases, as described in Table 29, have been studied for 5000 cycles (timesteps). For a complete simulation, typically more than 100,000 cycles are necessary. Large input decks exceed 150000 elements. RADIOSS version 3.1n was compiled using XLF 3.2.5 for P2SC

and XLF 5.1.0.0 for POWER3 and PowerPC 604e, respectively. For POWER3 and PowerPC 604e, the flag -qstrict was applied.

Table 29. RADIOSS Benchmark Test Cases

case 1	Front impact of a car on a rigid wall: 66,288 3D 4-node elements
case 2	Side impact of a car with a solid barrier: 2 dummies included 47,928 3D 4-node shell elements 6,325 8-node brick solid elements

The benchmark results are shown in Table 30. As RADIOSS is using 64 bit floating-point arithmetic, the 32 bit PowerPC 604e platform is outperformed by P2SC. Probably due the introduction of a 4 MB L2 cache and due to the advanced chip logic, such as branch prediction and out-of-order execution, the POWER3 platform offers a solid speedup of more than 30 percent compared to P2SC. The throughput of a two processor system Model 260 is almost perfect for this benchmark, taking into account that the I/O was performed sequentially. With help of a second disk, the I/O performance could be improved, too.

Table 30. RADIOSS Benchmark Results

case	time [sec]	PowerPC 604e Model F50 (332 MHz)	P2SC Model 397 (160 MHz)	POWER3 Model 260 (200 MHz)	
				1 job	throughput (2 jobs, 1 disk only)
1	real user system		5089 5071 1	3476 3475 1	3499 / 3540 3484 / 3523 1 / 1
2	real user system	6678 6622	5410 5391 1	3561 3560 1	

Take Note

These benchmarks were performed with a pre-GA POWER3 version of RADIOSS on pre-GA hardware for the purpose of an early performance evaluation. This does not imply availability of this product nor support by Mecalog. The performance may change as the application software and the compiler develop.

10.6 Finite Difference Kernel

In the finite difference numerical methods, frequently it requires to compute the partial derivatives and the grid-point average of the field variable. The example shown below illustrates the tuning method when using this type of computational kernel. The example is the SUBROUTINE RESID of the mg (multigrid) program from NAS Parallel Benchmarks (NPB).

The sample code distributed in NAS NPB 1.0 requires 18 floating adds and 3 floating-point multiply-add instructions.

```
      do 600 k =2,n-1
      do 600 j =2,n-1
      do 600 i =2,n-1
        r(i,j,k)=v(i,j,k)
     >     -a0*( u(i,  j,  k  ) )
     >     -a2*( u(i-1,j-1,k  ) + u(i,  j-1,k  )
     >        +  u(i-1,j,  k  ) + u(i,  j,  k  )
     >        +  u(i,  j-1,k-1) + u(i,  j,  k-1)
     >        +  u(i,  j-1,k  ) + u(i,  j,  k  )
     >        +  u(i-1,j,  k-1) + u(i-1,j,  k  )
     >        +  u(i,  j,  k-1) + u(i,  j,  k  ))
     >     -a3*( u(i-1,j-1,k-1) + u(i,  j-1,k-1)
     >        +  u(i-1,j,  k-1) + u(i,  j,  k-1)
     >        +  u(i-1,j-1,k  ) + u(i,  j-1,k  )
     >        +  u(i-1,j,  k  ) + u(i,  j,  k  ))
  600 continue
```

The total number of arithmetical operations can be reduced by pre-computing two grid point averages of u field and storing them in two scratch vectors, namely u1 and u2. The following code from NAS NPB 2.3 requires nine floating point adds and three floating point multiply-add instructions.

```
      do i3=2,n-1
      do i2=2,n-1
        do i1=1,n
          u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
     >           + u(i1,i2,i3-1) + u(i1,i2,i3+1)
          u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
     >           + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
        enddo
        do i1=2,n-1
          r(i1,i2,i3) = v(i1,i2,i3)
     >                - a0 * u(i1,i2,i3)
     >                - a2 * ( u2(i1) + u1(i1-1) + u1(i1+1) )
     >                - a3 * ( u2(i1-1) + u2(i1+1) )
        enddo
```

```
      enddo
      enddo
```

The preceding code works well in a vector computer. However, due to the presence of two additional temporary vectors, the performance gain of the this code is not proportional to the reduction of the number of arithmetic operations on a cache based superscalar computer. On the Model 260, the speedup factor is about 1.13.

For Model 260, this computational kernel can be coded as follows. The two scratch vectors are replaced by six scalar temporaries. u1m, u1i, and u1p represent u1(i-1), u1(i), and u1(i+1) respectively, similarly for u2. In the loop u1m and u1i are iteratively replaced, and u1p is recomputed. The compiler will recognize that this is a *predictive commoning* construct. Thus, it eliminates the need for load/store of vector temporaries as shown in the previous example code.

```
  do k=2,n-1
  do j=2,n-1
    u1m = u(1,j-1,k) + u(1,j+1,k)
 >      + u(1,j,k-1) + u(1,j,k+1)
    u2m = u(1,j-1,k-1) + u(1,j+1,k-1)
 >      + u(1,j-1,k+1) + u(1,j+1,k+1)
    u1i = u(2,j-1,k) + u(2,j+1,k)
 >      + u(2,j,k-1) + u(2,j,k+1)
    u2i = u(2,j-1,k-1) + u(2,j+1,k-1)
 >      + u(2,j-1,k+1) + u(2,j+1,k+1)
    do i=2,n-1
      u1p = u(i+1,j-1,k) + u(i+1,j+1,k)
 >        + u(i+1,j,k-1) + u(i+1,j,k+1)
      u2p = u(i+1,j-1,k-1) + u(i+1,j+1,k-1)
 >        + u(i+1,j-1,k+1) + u(i+1,j+1,k+1)
      r(i,j,k) = v(i,j,k)
 >             - a0 * u(i,j,k)
 >             - a2 * ( u2i + u1m + u1p )
 >             - a3 * ( u2m + u2p )
       u1m = u1i
       u2m = u2i
       u1i = u1p
       u2i = u2p
     enddo
 enddo
 enddo
```

The speedup factor of the above code over the original code (NPB 1.0) is about 1.56.

10.7 Iterative Eigenvalues Solver

This customer benchmark program is an engineering analysis for ship building applications. The Jacob iterative method of computing eigenvalues for dense matrices dominate the CPU utilitization of this benchmark setup. SUBROUTINE JACOBI consumes more than 99% CPU time. The following code fragment shows the key part of this subroutine.

```
      SUBROUTINE JACOBI( N, RTOL, NSMAX, IFPR, IOUT)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      PARAMETER(NNSVAB=840,NNELEM=144,MMDDOF=42)
      COMMON /P2/X(NNSVAB,NNSVAB),B(NNSVAB,NNSVAB)
      COMMON /P6/A(NNSVAB,NNSVAB)
      COMMON /P8/EIGV(NNSVAB)
C
      ........<set up>............
      DO 300 NSWEEP = 1, NSMAX
        EPS=(0.01**NSWEEP)**2
        DO 230 J = 1, NR
        DO 210 K = J+1, N
          EPTOLA = (A(J,K)*A(J,K)) / (A(J,J)*A(K,K))
          EPTOLB = (B(J,K)*B(J,K)) / (B(J,J)*B(K,K))
          IF (EPTOLA.LT.EPS .AND. EPTOLB.LT.EPS) GO TO 210
          AKK = A(K,K)*B(J,K) - B(K,K)*A(J,K)
          AJJ = A(J,J)*B(J,K) - B(J,J)*A(J,K)
          AB  = A(J,J)*B(K,K) - B(J,J)*A(K,K)
          CHECK = (AB*AB+4.*AKK*AJJ) / 4.
          IF (CHECK .LT. 0.) STOP 'CHECK'
          SQCH = SQRT(CHECK)
          D1 = 0.5*AB + SQCH
          D2 = 0.5*AB - SQCH
          DEN = D1
          IF (ABS(D2) .GT. ABS(D1)) DEN = D2
          IF ( DEN .EQ. 0.) THEN
            CA = 0.D0
            CG = -A(J,K)/A(K,K)
          ELSE
            CA = AKK/DEN
            CG = -AJJ/DEN
          END IF
          IF (J.GT.2) THEN
            IF (J-1 .GE. 0) THEN
              DO 120 I = 1, J-1
                AJ = A(I,J) BJ = B(I,J)
                AK = A(I,K)
                BK = B(I,K)
                A(I,J) = AJ + CG*AK
                B(I,J) = BJ + CG*BK
                A(I,K) = AK + CA*AJ
                B(I,K) = BK + CA*BJ
  120         CONTINUE
            END IF
            IF (K+1 .LE. N) THEN
              DO 150 I = K+1, N
                AJ = A(J,I)
                BJ = B(J,I)
                AK = A(K,I)
                BK = B(K,I)
                A(J,I) = AJ + CG*AK
                B(J,I) = BJ + CG*BK
```

```
              A(K,I) = AK + CA*AJ
              B(K,I) = BK + CA*BJ
150         CONTINUE
          END IF
          IF (J+1 .LE. K-1) THEN
            DO 180 I = JP1, KM1
              AJ = A(J,I)
              BJ = B(J,I)
              AK = A(I,K)
              BK = B(I,K)
              A(J,I) = AJ + CG*AK
              B(J,I) = BJ + CG*BK
              A(I,K) = AK + CA*AJ
              B(I,K) = BK + CA*BJ
180         CONTINUE
          END IF
        END IF
        AK = A(K,K)
        BK = B(K,K)
        A(K,K) = AK + 2.D0*CA*A(J,K) + CA*CA*A(J,J)
        B(K,K) = BK + 2.D0*CA*B(J,K) + CA*CA*B(J,J)
        A(J,J) = A(J,J) + 2.D0*CG*A(J,K) + CG*CG*AK
        B(J,J) = B(J,J) + 2.D0*CG*B(J,K) + CG*CG*BK
        A(J,K) = 0.D0
        B(J,K) = 0.D0
        DO 200 I = 1, N
          XJ = X(I,J)
          XK = X(I,K)
          X(I,J) = XJ + CG*XK
          X(I,K) = XK + CA*XJ
200     CONTINUE
210   CONTINUE
230   CONTINUE
      DO 250 I = 1,N
        EIGV(I) = A(I,I)/B(I,I)
250   CONTINUE
       .....<check for convergence>.............
300 CONTINUE
331 CONTINUE    ! converged
    ........<clean up>.........

    RETURN
    END
```

The A and B matrices are positive definite symmetric. For this benchmark, N=798. (arrays are declared as 840x840), and NSWEEP=20. The inner most DO loops 120, 150, and 180 are the hot spots of this program. Notice that these loops are processing the data on the upper triangle of matrix A and B. Loop 120 access A and B with unit stride, loop 150 with stride of 840, and loop 180 with unit stride of 840. A stride of 840 elements is accessing data in memory every 6720 bytes interval. The frequently used programming techniques of loop interchange (nested loops), matrix transposition, and cache blocking are not applicable here.

The tuning method for this code is to take advantage that A and B are the symmetric matrices. Loop 150 can be easily modified to a unit stride data access loop by working on the lower triangle of the matrices. To make loop

180 with a unit stride access, the method is to compute this loop upon the completion of loop 120 and 150 for each column of J (completion of K loop, DO 210). The modified will require to store CG and CA values, copy one column of data from lower triangle to upper triangle for every J iteration, and copy the upper triangle to the lower triangle for each completion of sweep (DO 300). The essence of the tuned code is shown below.

```
      DO 300 NSWEEP = 1, NSMAX
        EPS=(0.01**NSWEEP)**2
        DO 230 J = 1, NR
          DO 210 K = J+1, N
            IDX(K) = 0         !!!!
            IF (A(K,J)*A(K,J).LT.A(J,J)*A(K,K)*EPS .AND.
     &          B(K,J)*B(K,J).LT.B(J,J)*B(K,K)*EPS GOTO 210
            IDX(K) = 1         !!!!
            ..... < compute CA and CG >.....
            CAX(K) = CA    !!!
            CGX(K) = CG    !!!
            IF (N .GT. 2) THEN
CIBM ...    DO THIS LOOP AT UPPER TRIANGLE
              IF (J-1 .GE. 0) THEN
                DO 120 I = 1, J-1
                  AJ = A(I,J)
                  BJ = B(I,J)
                  AK = A(I,K)
                  BK = B(I,K)
                  A(I,J) = AJ + CG*AK
                  B(I,J) = BJ + CG*BK
                  A(I,K) = AK + CA*AJ
                  B(I,K) = BK + CA*BJ
                  A(I,J) = AJ + CG*AK
                  B(I,J) = BJ + CG*BK
                  A(I,K) = AK + CA*AJ
                  B(I,K) = BK + CA*BJ
  120           CONTINUE
              END IF
CIBM ................. DO THIS LOOP AT LOWER TRIANGLE
              IF (K+1 .LE. N) THEN
                DO 150 I = K+1, N
                  AJ = A(I,J)
                  BJ = B(I,J)
                  AK = A(I,K)
                  BK = B(I,K)
                  A(I,J) = AJ + CG*AK
                  B(I,J) = BJ + CG*BK
                  A(I,K) = AK + CA*AJ
                  B(I,K) = BK + CA*BJ
  150           CONTINUE
              END IF
C-------
            END IF
            AK = A(K,K)
            BK = B(K,K)
            A(K,K) = AK + 2.D0*CA*A(K,J) + CA*CA*A(J,J)
            B(K,K) = BK + 2.D0*CA*B(K,J) + CA*CA*B(J,J)
            A(J,J) = A(J,J) + 2.D0*CG*A(K,J) + CG*CG*AK
            B(J,J) = B(J,J) + 2.D0*CG*B(K,J) + CG*CG*BK
            A(K,J) = 0.D0
            B(K,J) = 0.D0
C
            DO 200 I = 1, N
```

```
              XJ = X(I,J)
              XK = X(I,K)
              X(I,J) = XJ + CG*XK
              X(I,K) = XK + CA*XJ
  200       CONTINUE
C
  210     CONTINUE
CIBM.............. DO LOOP 180 HERE
DO K = J+1,N
            ATMP = A(K,j)
            DO 180 I = K+1,N
                BJ = BTMP
                ATMP   = AJ + CGX(I) * A(I,K)
                BTMP   = BJ + CGX(I) * B(I,K)
                A(I,K) = A(I,K) + CAX(I)*AJ
                B(I,K) = B(I,K) + CAX(I)*BJ
  180       CONTINUE
            A(K,J) = ATMP
            B(K,J) = BTMP
          ENDDO
CIBM  --------  COPY ONE COLUMN TO UPPER TRIANGLE
          DO I = J+1,N
            A(J,I) = A(I,J)
            B(J,I) = B(I,J)
          ENDDO

  230   CONTINUE
C
CIBM ----  COMPLETE A SWEEP...COPY UPPER TRIANGLE TO LOWER
        DO J = 1,N-1
        DO K = J+1,N
          A(K,J) = A(J,K)
          B(K,J) = B(J,K)
        ENDDO
        ENDDO
c
        DO 220 I = 1,N
          EIGV(I) = A(I,I) / B(I,I)
  220   CONTINUE
C ...............
C
  300 CONTINUE
```

The performance of original and tuned for NSWEEP=20, and N=798 is shown in Table 31. The timing for tuned code includes the unrolling of loop 120, 150, and 180 manually (not shown in the above listed code).

Table 31. CPU Time for SUBROUTINE JACOBI, (in Seconds)

	M590 66.5Mhz	P2SC 160Mhz	POWER3 200Mhz
	AIX 3, XLF 3.2.2	AIX 4.3, XLF 5.1.1	AIX 4.3, XLF 5.1.1
Original	4408.8	3637.8	2016.7
Tuned	576.4	298.8	285.7

Appendix A. Industry Standard Benchmarks

In this appendix, the performance of the Model 260 using four widely quoted industry standard benchmarks will be presented.

Take Note

The performance numbers of the Model 260 shown in this Appendix are the results of pre-GA systems. All Model 260 data shown are estimated values and presented for illustrative purpose only. The official performance data of the Model 260 will be submitted by IBM to the organizations responsible for these benchmarks.

A.1 LINPACK Benchmark

Web site: `http://www.netlib.org`

The LINPACK benchmark measures the performance of a computer system in solving a system of linear equations. No code modification is allowed for the matrix order n=100 case. The performance of this case heavily depends on the ability of FORTRAN compiler and preprocessor making the high order transformation to generate a BLAS 3 code. The n=1000 case (referred to as TPP, Toward Peak Performance) allows for replacing LU routines. Thus, the efficient coding will implement a cache blocking technique. The importance of high sustained memory bandwidth in today's application programs is not presented in this benchmark. Table 32 shows the results of Model 260.

Table 32. LINPACK Performance

		P2SC(160Mhz)	Model 260, 1 CPU
LINPACK DP MFLOPS, n=100		311.9	233.1
LINPACK TPP	MFLOPS	528.0	642.0
	% of peak	82.5	80.2

The LINPACK DP, n=100 performance differences between P2SC (160Mhz) and Model 260 is primarily due to the size of L1 cache. The required active data needed for LINPACK DP, n=100 benchmark is about 90 KB. This fits in L1 data cache of P2SC processor, but not in Model 260.

A.2 SPEC95

Web site: `http://www.specbench.org`

System Performance Evaluation Cooperative (SPEC) benchmark suite consists of 10 FORRAN 77 floating-point programs (SPECfp95) and eight C-language integer programs (SPECint95). Since the modification of these benchmark programs are not allowed, the compiler optimization and FORTRAN preprocessor capability play an important role on floating-point benchmark. The memory requirements of the benchmark programs is rather small compared to the actual application programs running today.

Table 33. SPEC95 Performance

	P2SC (160Mhz)	Model 260
SPECfp_base95	23.6	27.6
SPCEfp95	26.6	30.1
SPECint_base95	7.77	12.5
SPECint95	8.62	13.2

A.3 STREAM

Web site: `http://www.cs.virginia.edu/stream`

The CPU speed growth rate has been outpacing memory speed growth in the last decade. In the same time period, the problem size of application programs grew rapidly. The sustained memory bandwidth becomes an important factor of program performance. STREAM is a simple, synthetic benchmark that measures the sustainable memory bandwidth. The sustained memory bandwidth takes into account of memory latency in addition to the actual data transfer on the memory bus or switch.

Table 34. Sustained MB/s Memory Bandwidth Measured by STREAM

			P2SC(160Mhz)	Model 260
Name	Kernel	bytes/iteration	MB/s	MB/s
Copy	a(i)=b(i)	16	779.2	941.8
Scale	a(i)=q*b(i)	16	775.5	985.1
Sum	a(i)=b(i)+c(i)	24	883.9	1096.3
Triad	a(i)=b(i)+q*c(i)	24	881.2	1102.8

The STREAM definition of bytes/iteration is based on the memory being referenced in a kernel. Typically, for a cache based microprocessor, it will require to transfer a(i) array in these kernel twice (cache load and store back) on the memory bus or switch. Whereas, the STREAM benchmark counts only once.

A.4 NAS NPB 1.0

Web site: `http://www.nas.nasa.gov//NAS/NPB`

NAS (Numerical Aerodynamic Simulation) Parallel Benchmarks NPB 1.0 consists of eight programs. The first five (EP, FT, IS, MG, and CG) are kernel benchmarks with simple data structure. The simulated application benchmarks, which compute the numerical solution to the nonlinear partial differential equations, are LU (direct LU decomposition solver), SP (scalar pentadiagonal) and BT (block tridiagonal). NPB 1.0 is a *pencil and paper* benchmark. The primary objective is the parallel performance. In early NAS publications, it also includes the serial one CPU data. Table 35 shows the single CPU performance of LU, SP, and BT on Model 260 as compared to IBM RS/6000 SP Wide-node2 (77 MHz), data published in "NAS Parallel Benchmark Results 12-95", by S. Saini and David Bailey, Report NAS-95-02, December 1995. (Available from the URL listed at the beginning of this section).

Table 35. NAS NPB 1.0 (LU, SP, BT) Single CPU Performance, Time in Seconds

	SP Wide-node2 (77Mhz)		Model 260	
	Class A	Class B	Class A	Class B
LU	501.5	2066.6	235.8	980.3
SP	711.8	3087.0	422.0	1760.7
BT	1130.7	4775.7	654.2	2762.0

Appendix B. Enabling Vector Codes to POWER3

This appendix gives a brief discussion of the performance considerations when porting vector codes to the POWER3 superscalar architecture.

In general, a good vector program will perform well on POWER3 when compiled by XL Fortran Version 5.1.1. For optimal performance on POWER3, consider the following discussion.

B.1 Data Access

Most of vector codes are programmed for interleaving memory without cache. Vector load/store operations on a vector computer take place directly from memory to vector registers. POWER3 uses a cache-based memory subsystem. As discussed previously, the stride used is important for high performance on POWER3. Typically, large non-unit stride data access can be remedied by recoding with the following methods:

- Interchange loops
- Transpose matrix and arrays
- Cache blocking

In a vector computer, many codes introduce vector temporaries to facilitate vectorization. This is not necessary on POWER3. If it is possible, scratch scalar temporaries is preferred to avoid non-necessary load/stores.

B.2 Data Dependency and Recursive Code

Vector computers and RISC superscalar computers are both of pipeling machines. The vector computer pipelines the data stream, each vector instruction operates on data contained in the vector registers consisting of n elements, typical of above 64. The key performance factor of a vector machine is how DO loop can be vectorized. A loop with recursive data access cannot be vectorized (data depedence is allowed). The performance of vector and non-vector loop can be as much as a factor of ten. The vectorization consideration more often than not over shadows the reduction of arithmetic operations in the loops, for example, cyclic reduction algorithms and red-black ordering of Gauss-Seidel iteration method.

POWER3, which is a RISC superscalar computer, pipelines the instruction stream. Each instruction operates on a set of floating-point registers of one element. The performance penalty due to dependency and recursion in the loops is less critical than on a vector computer. At most, the penalty is

instruction latency is 3-4 cycles in POWER3. Nevertheless, this data dependency can be overcome with simple loop unrolling (xlf version 5.1.1 does a good job of this). This will result in a sufficiently large instruction stream which allows pipeline executions.

B.3 Vector Length

For good performance, the vector computer benefits from a long vector. The peak performance of a loop is achieved when the vector length is in the order of 1000. Thus, many vector codes that compute 3D problems by using 1D arrays need to lengthen the vector length. By doing this, the code can perform additional computation to check boundary conditions and indexing vectors to address the 1D array for 3D space. Vector length consideration is significantly less important on POWER3. (Remember, it pipelines the instructions rather than data.) A straight forward coding practice is to declare explicitly the dimension in 3D and to code the computations with three nested loops.

B.4 Conditional Processing

A vector computer handles conditional processing (IF-THEN-ELSE) by computing both *true* and *false* branches, then selects the result based on a vector mask register which contains the value of the IF condition. POWER3 takes the branch upon the conditional test and execute the instruction within the target branch. The following code fragment illustrates the difference between vector computer and POWER3 code.

Code A	Code B

eps = 1.d0-100
DO i = 1,n
.....
  umax =...
  umin=...
  xx = sqrt(max(0.,x-a))
  u=(b+prmin-plmin) / (eps+c-sgn*xx)
  xx=sqrt(max(0.,x+a))
  z=(b-prmin+plmin) / (eps+c-sgn*xx)
  IF (umax.LT.umin) u=z
.....
ENDDO
	eps=1.d0-100
	DO i =1,n
	.....
	  umax=...
	  umin=..
	  IF (umax.LT.umin) THEN
	  xx=sqrt(max(0.,x+a)
	  u=(b-prmin+plmin)
	  ELSE
	  xx=sqrt(max(0.,x-a)
	  u=(b+prmin-plmin)
	  ENDIF
	  u = u / (eps+c-sgn*xx)
	.....
	ENDDO

For a vector computer, the performance difference of Code A and Code B is very little. Code B should be used for POWER3.

The Gather/Scatter coding technique is commonly employed in the vector programs (to minimize the arithmetical operations in the IF clause). In general, POWER3 programming will also benefit from this coding method.

Many old Cray vector programs may contain the vector merge intrinsic functions, namely CVMxx. XL FORTRAN supports these functions. For performance considerations, these functions should be replaced with the equivalent conditional codes.

Appendix C. Threads

Thread support, added to AIX in Version 4, divides program-execution control into two elements:

- A process is a collection of physical resources required to run the program, such as memory and access to files.
- A thread is the execution state of an instance of the program, such as the current contents of the instruction-address register and the general-purpose registers. Each thread runs within the context of a given process and uses that process's resources. Multiple threads can run within a single process, sharing its resources.

The following sections will give a short introduction to threads in general and specific AIX thread implementation details. The last sections shows a program example using both directives and the Fortran POSIX thread interface. For a more complete description about the AIX thread implementation please see:

- *AIX Performance Tuning Guide,* SR28-5930
- *AIX General Programming Concepts: Writing and Debugging Programs,* SC23-2205

C.1 Symmetric Multiprocessing (SMP) Concepts and Architecture

As with any change that increases the complexity of the system, the use of multiple processors generates design considerations that must be addressed for satisfactory operation and performance. The additional complexity gives more scope for hardware/software trade-offs and requires closer hardware/software design coordination than in uniprocessor systems. The different combinations of design responses and trade-offs give rise to a wide variety of multiprocessor system architectures.

The major design considerations are:

- Symmetrical verses Asymmetrical Multiprocessors

  In an asymmetrical multiprocessor system, the processors are assigned different roles. One processor may handle I/O, while others execute user programs, and so forth. Some of the advantages and disadvantages of this approach are:

  - By restricting certain operations to a single processor, some forms of data serialization and cache coherency problems (see the following

discussion) can be reduced or avoided. Some parts of the software may be able to operate as though they were running in a uniprocessor.

- In some situations, I/O-operation or application-program processing may be faster because it does not have to contend with other parts of the operating system or the workload for access to a processor. In other situations, I/O-operation or application-program processing can be slowed because not all of the processors are available to handle peak loads.
- The existence of a single processor handling specific work creates a unique point of failure for the system as a whole.

In a symmetric multiprocessor system, all of the processors are essentially identical and perform identical functions:

- All of the processors work with the same virtual and real address spaces.
- Any processor is capable of running any thread in the system.
- Any processor can handle any external interrupt. (Each processor handles the internal interrupts generated by the instruction stream it is executing.)
- Any processor can initiate an I/O operation.

This interchangeability means that all of the processors are potentially available to handle whatever needs to be done next. The cost of this flexibility is primarily borne by the hardware and software designers, although symmetry also makes the limits on the multiprocessability of the workload more noticeable.

The RS/6000 family contains, and AIX Version 4 supports, only symmetric multiprocessors.

C.2 Thread Implementation Model

There are various thread implementation models. One model is the *library-thread model*. In such a model, the threads of a process are not visible to the operating system kernel, and the threads are not kernel scheduled entities. The process is the only kernel scheduled entity. The process is scheduled onto the processor by the kernel according to the scheduling attributes of the process. The threads are scheduled onto the single kernel scheduled entity (the process) by the run-time library according to the scheduling attributes of the threads. This is the model of threads provided on AIX Version 3.

At the other end of the spectrum is the *kernel-thread model*. In this model, all threads are visible to the operating system kernel. Thus, all threads are kernel scheduled entities, and all threads can concurrently execute. The threads are scheduled onto processors by the kernel according to the scheduling attributes of the threads. This model is the model provided in AIX Version 4.1 and AIX Version 4.2.

AIX Version 4.3 uses a hybrid model that offers the speed of library threads and the concurrency of kernel threads. In hybrid models, a process has a varying number of kernel scheduled entities associated with it. It also has a potentially much larger number of library threads associated with it. Some library threads may be bound to kernel scheduled entities, while the other library threads are multiplexed onto the remaining kernel scheduled entities. For this reason, a hybrid model is referred to as a *M:N model*. In this model, the process can have multiple concurrently executing threads; specifically, it can have as many concurrently executing threads as it has kernel scheduled entities.

In order to make the switch to thread programming easier, AIX introduced threads API models based on preliminary drafts of the now-official IEE POSIX standard. AIX 4.3 is the first release to conform fully to the IEEE POSIX standard for threads APIs, IEEE POSIX 1003.1-1996.

C.3 Understanding Threads

A thread is an independent flow of control that operates within the same address space as other independent flows of controls within a process. In previous versions of AIX, and in most of UNIX systems, thread and process characteristics are grouped into a single entity called a process. In other operating systems, threads are sometimes called *lightweight processes*, or the meaning of the word thread is sometimes slightly different.

C.3.1 Threads and Processes

In traditional single-threaded process systems, a process has a set of properties. In multi-threaded systems, these properties are divided between processes and threads.

C.3.1.1 Process Properties

A process in a multi-threaded system is the changeable entity. It must be considered as an execution frame. It has all traditional process attributes, such as:

- Process ID, process group ID, user ID, and group ID

- Environment
- Working directory

A process also provides a common address space and common system resources:

- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory)

C.3.1.2 Thread Properties

A thread is the schedulable entity. It has only those properties that are required to ensure its independent flow of control. These include the following properties:

- Stack
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Some thread-specific data

An example of thread-specific data is the error indicator, errno. In multi-threaded systems, errno is no longer a global variable, but usually, a subroutine returning a thread-specific errno value. Some other systems may provide other implementations of errno.

Threads within a process must not be considered as a group of processes. All threads share the same address space. This means that two pointers having the same value in two threads refer to the same data. Also, if any thread changes one of the shared system resources, all threads within the process are affected. For example, if a thread closes a file, the file is closed for all threads.

C.3.1.3 The Initial Thread

When a process is created, one thread is automatically created. This thread is called the initial thread. It ensures the compatibility between the old processes with a unique implicit thread and the new multi-threaded processes. The initial thread has some special properties, not visible to the programmer, that ensure binary compatibility between the old single-threaded programs and the multi-threaded operating system. It is also the initial thread that executes the main routine in multi-threaded programs.

C.3.2 Threads Implementation

A thread is the schedulable entity, meaning that the system scheduler handles threads. These threads, known by the system scheduler, are strongly implementation-dependent. To facilitate the writing of portable programs, libraries provide another kind of thread.

C.3.2.1 Kernel Threads

A kernel thread is a kernel entity, like processes and interrupt handlers; it is the entity handled by the system scheduler. A kernel thread runs within a process, but can be referenced by any other thread in the system. The programmer has no direct control over these threads, unless writing kernel extensions or device drivers. See *AIX Kernel Extensions and Device Support Programming Concepts, SC23-2207* for more information about kernel programming.

C.3.2.2 User Threads

A user thread is an entity used by programmers to handle multiple flows of controls within a program. The API for handling user threads is provided by a library, the threads library. A user thread only exists within a process; a user thread in process A cannot reference a user thread in process B. The library uses a proprietary interface to handle kernel threads for executing user threads. The user threads API, unlike the kernel threads interface, is part of a portable programming model. Thus, a multi-threaded program developed on an AIX system can easily be ported to other systems.

On other systems, user threads are simply called threads, and lightweight process refers to kernel threads.

C.3.3 Thread Scheduling

In previous versions of AIX, the CPU scheduler dispatched processes. In AIX Version 4, the scheduler dispatches threads. In the SMP environment, the availability of thread support makes it easier and less expensive to implement SMP-exploiting applications. Forking multiple processes to create multiple flows of control is cumbersome and expensive, since each process has its own set of memory resources and requires considerable system processing to set up. Creating multiple threads within a single process requires less processing and uses less memory.

Thread support exists at two levels:

libpthreads.a support in the application program environment

kernel thread support

C.3.3.1 Default Scheduler Processing of Migrated Workloads

The new division between processes and threads is invisible to existing programs. In fact, workloads migrated directly from earlier releases of AIX create processes as before. Each new process is created with a single thread (the initial thread) that contends for the CPU with the threads of other processes. The default attributes of the initial thread, in conjunction with the new scheduler algorithms, minimize changes in system dynamics for unchanged workloads.

Priorities can be manipulated with the nice and renice commands and the setpri and setpriority system calls, as before. The scheduler allows a given thread to run for at most one time slice (normally 10ms) before forcing it to yield to the next dispatchable thread of the same or higher priority.

C.3.3.2 Scheduling Algorithm Variables

Several variables affect the scheduling of threads. Some are unique to thread support; others are elaborations of process-scheduling considerations:

- Priority:

  A thread's priority value is the basic indicator of its precedence in the contention for processor time.

- Scheduler run queue position:

  A thread's position in the scheduler's queue of dispatchable threads reflects a number of preceding conditions.

- Scheduling policy:

  This thread attribute determines what happens to a running thread at the end of the time slice.

- Contention scope:

  A thread's contention scope determines whether it competes only with the other threads within its process or with all threads in the system. A pthread created with process contention scope is scheduled by the library, while those created with system scope are scheduled by the kernel. The library scheduler uses a pool of kernels threads to schedule pthreads with process scope.

  Generally, pthreads should be created with system scope, if they are performing I/O. Process scope is useful, when there is a lot of intra-process synchronizations. Contention scope is a libpthreads.a concept.

Notice

The default for the contention scoop on all IBM RS/6000 machines is processor scope. There are two ways to change this behavior:

1. Use of pthread_attr_setscope() in the code.
2. Set the environment variable AIXTHREAD_SCOPE to S.

- Processor affinity:

  The degree to which affinity is enforced affects performance.

The combinations of these considerations can seem complex, but there are essentially three distinct approaches from which to choose in managing a given process:

- Default:

  The process has one thread, whose priority varies with CPU consumption and whose scheduling policy, SCHED_OTHER, is comparable to the AIX Version 3 algorithm.

- Process-level control:

  The process can have one or more threads, but the scheduling policy of those threads is left as the default SCHED_OTHER, which permits the use of the existing AIX Version 3 methods of controlling nice values and fixed priorities. All of these methods affect all of the threads in the process identically. If `setpri()` is used, the scheduling policy of all of the threads in the process is set to SCHED_RR.

- Thread-level control:

  The process can have one or more threads. The scheduling policy of these threads is set to SCHED_RR or SCHED_FIFO, as appropriate. The priority of each thread is fixed and is manipulated with thread-level subroutines.

C.3.3.3 Scheduling Environment Variables

Within the libpthreads.a framework, a series of tuning knobs have been provided that may impact the performance of the application. When using XL Fortran, most of the following environment variables can and should be controlled by the environment variable XLSMPOPTS, described in section 4.8, “XLSMPOPTS Environment Variable” on page 57. A case where you

cannot use XLSMPOPTS would be to change the default of the contention scope. The environment variables are:

- SPINLOOPTIME=n, where n is the number of times to retry a busy lock before yielding to another processor. n must be a positive value.
- YIELDLOOPTIME=n, where n is the number of times to yield the processor before blocking on a busy lock. n must be a positive value. The processor is yielded to another kernel thread, assuming there is another runnable one with sufficient priority.
- AIXTHREAD_SCOPE={P|S}, where P signifies process based contention scope and S signifies system based contention scope. Either P or S should be specified. The braces are provided for syntactic reasons only. The use of this environment variable impacts only those threads created with the default attribute. The default attribute is employed when the attr parameter to pthread_create is NULL.

The following environment variables impact the scheduling of pthreads created with process based contention scope:

- AIXTHREAD_MNRATIO=p:k, where k is the number of kernel threads that should be employed to handle p runnable pthreads. This environment variable controls the scaling factor of the library. This ratio is used when creating and terminating pthreads.
- AIXTHREAD_SLPRATIO=k:p, where k is the number of kernel threads that should be held in reserve for p sleeping pthreads. In general, fewer kernel threads are required to support sleeping pthreads, since they are generally woken one at a time when processing locks and/or events. This conserves kernel resources.
- AIXTHREAD_MINKTHREADS=n, where n is the minimum number of kernel threads that should be used. The library scheduler will not reclaim kernel threads below this figure. A kernel thread may be reclaimed at virtually any point. Generally, a kernel thread is targeted as a result of a pthread terminating.

C.3.4 Thread Models and Virtual Processors

User threads are mapped to kernel threads by the threads library. The way this mapping is done is called the thread model. There are three possible thread models, corresponding to three different ways to map user threads to kernel threads:

- M:1 model
- 1:1 model

- M:N model.

The mapping of user threads to kernel threads is done using virtual processors. A virtual processor (VP) is a library entity that is usually implicit. For a user thread, the virtual processor behaves as a CPU for a kernel thread. In the library, the virtual processor is a kernel thread or a structure bound to a kernel thread.

In the M:1 model, all user threads are mapped to one kernel thread; all user threads run on one VP. The mapping is handled by a library scheduler. All user threads programming facilities are completely handled by the library. This model can be used on any system, especially on traditional single-threaded systems. Figure 28 illustrates this model.

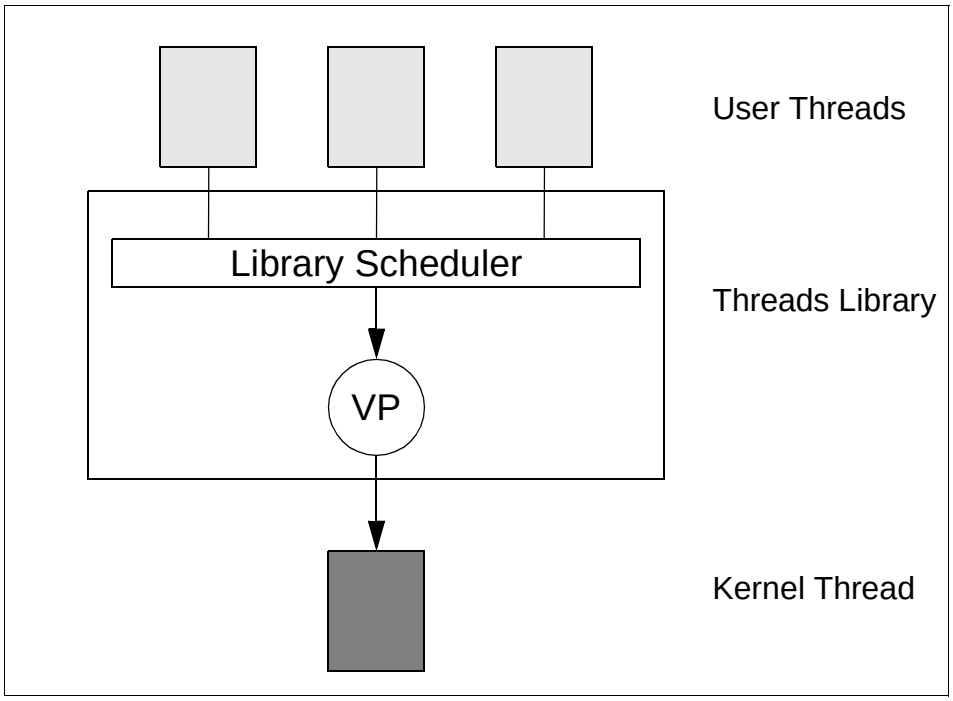


Figure 28. M:1 Threads Model

In the 1:1 model, each user thread is mapped to one kernel thread; each user thread runs on one VP. Most of the user threads programming facilities are directly handled by the kernel threads. Figure 29 illustrates this model.

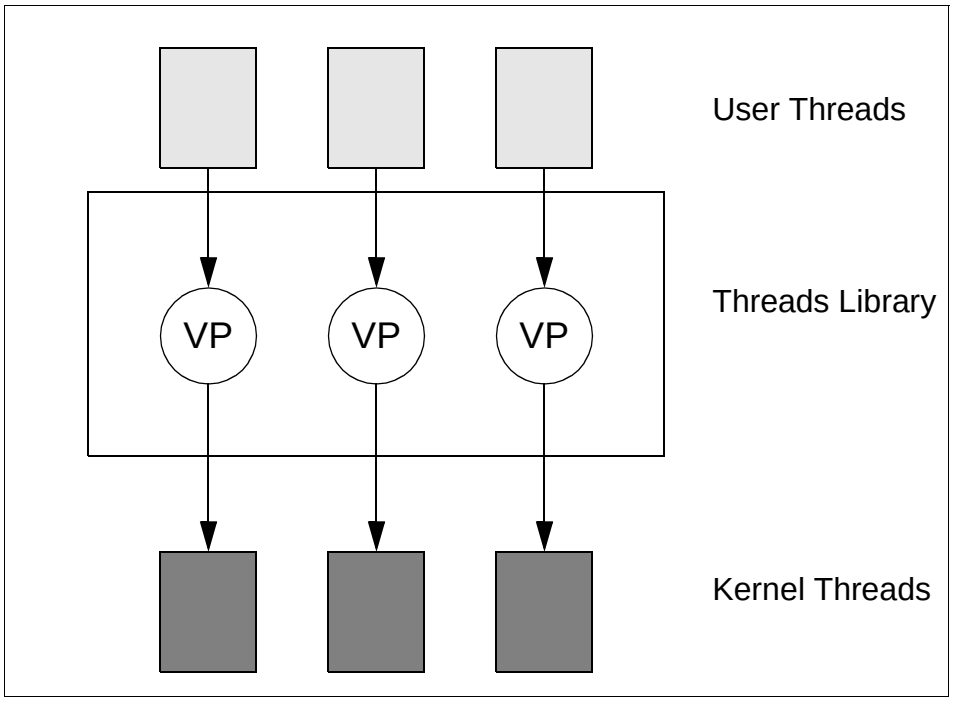


Figure 29. 1:1 Threads Model

In the M:N model, all user threads are mapped to a pool of kernel threads; all user threads run on a pool of virtual processors. A user thread may be bound to a specific VP, as in the 1:1 model. All unbound user threads share the remaining VPs. This is the most efficient and most complex thread model; the user threads programming facilities are shared between the threads library and the kernel threads. Figure 30 illustrates this model.

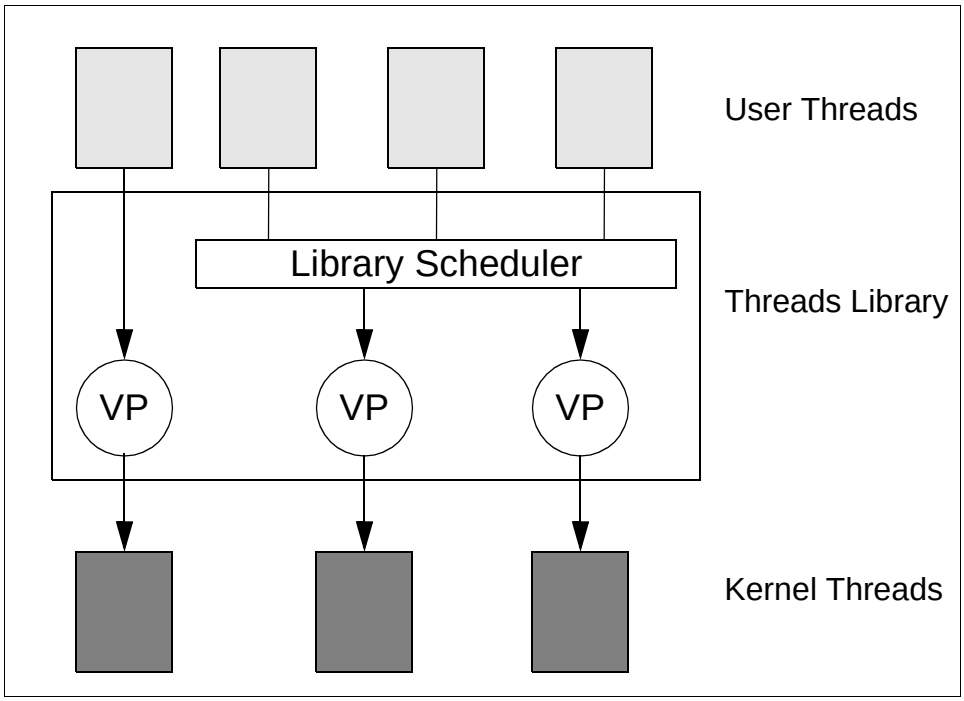


Figure 30. M:N Threads Model

C.3.5 Contention Scope and Concurrency Level

The contention scope of a user thread defines how it is mapped to a kernel thread. There are two possible contention scopes:

- *System contention scope*, sometimes called global contention scope

  A system contention scope user thread is a user thread that is directly mapped to one kernel thread. All user threads in a 1:1 thread model have system contention scope.

- *Process contention scope*, sometimes called local contention scope.

  A process contention scope user thread is a user thread that shares a kernel thread with other (process contention scope) user threads in the process. All user threads in a M:1 thread model have process contention scope.

In an M:N thread model, user threads can have either system or process contention scope. In the previous figure, for example, the user thread on the left side has system contention scope; the other ones all have process

contention scope. Therefore, an M:N thread model is often referred as a mixed-scope model.

The concurrency level is a property of M:N threads libraries. It defines the number of VPs used to run the process contention scope user threads. This number cannot exceed the number of process contention scope user threads, and is usually dynamically set by the threads library. The system also sets a limit to the number of available kernel threads.

C.3.6 libpthreads.a POSIX Threads Library

AIX provides a threads library, called libpthreads.a, based on the POSIX 1003.1c industry standard for a portable user threads API. Any program written for use with a POSIX thread library can easily be ported for use with another POSIX threads library; only the performance and very few subroutines of the threads library are implementation-dependent. For this reason, multi-threaded programs written for this version of AIX will work on any future version of AIX.

The XL Fortran V5 provides an interface to the pthread library called f_pthread. It is implemented as a Fortran 90 module.The naming convention is to use the prefix f_ before the corresponding AIX pthread routine name or type definition name. For more information about the f_pthread implementation, see the 4.10, “OpenMP Porting” on page 65 and the *XL Fortran Language Reference,* SC09-2607.

C.3.7 libpthreads_compat.a POSIX Draft 7 Threads Library

AIX provides binary compatibility for existing multi-threads applications that were coded to Draft 7 of the POSIX thread standard. These applications will run without re-linking.

The libpthreads_compat.a library is actually provided for program development. AIX Version 4.3 provides program support for both Draft7 of the POSIX Thread Standard and Xopen Version 5 Standard, which includes the final POSIX 1003.1c Pthread Standard.

Take Note

IBM’s default value for the detached state of a thread is PTHREAD_CREATE_DETACHED (not joinable). This is different than other platforms, such as SUN and SGI.

C.4 A Simple Thread Program

The following sections will compare the directive and POSIX thread approach. The example code, which is made thread-enabled, is a reduction of an array:

```
PROGRAM SEQUENTIAL
IMPLICIT NONE
INTEGER, PARAMETER :: IMAX=10000000
INTEGER(4) :: i
REAL(8) :: MYSUM
REAL(8), DIMENSION(IMAX) :: A

CALL INIT(A,IMAX)

MYSUM=0.D0
DO I=1,IMAX
MYSUM=MYSUM+A(I)
END DO

PRINT *,'SUM=',MYSUM
END PROGRAM
```

The subroutine `INIT` sets the array `A` to `A(I)=SIN(3.1415*I/N)`.

C.4.1 Using SMP Directives

This is a very simple test case and the compiler does not need any directives to parallelize it. By compiling with

```
xlf90_r -qsmp -O3 -qarch=pwr3 -qreport=smplist simple.f
```

the compiler produces a parallelization report for the main loop showing the basic idea of the parallelization:

```
mysum = 0d0
ScRed_12 = mysum
ScRed_13 = dble(0)
ScRed_14 = dble(0)
ScRed_15 = dble(0)
C 1585-501  Original Source Line 11
PARALLEL do i=1,10000000,4
         ScRed_12 = ScRed_12 + a(i)
         ScRed_13 = ScRed_13 + a(i + 1)
         ScRed_14 = ScRed_14 + a(i + 2)
         ScRed_15 = ScRed_15 + a(i + 3)
       end do
mysum = ScRed_12 + ScRed_13 + ScRed_14 + ScRed_15
```

As you see, the compiler recognizes the parallelism in the loop, unrolls the loop four times, introduces four temporary variables, and finally does the reduction. Please note that the above listing is only a report and does not necessarily show how the compiler finally optimizes the code. A deeper analysis and the use of the compiler flag -qlist shows that the compiler finally unrolled the loop 16 times.

For some programs, it make sense to compile with -qsmp=noauto telling the compiler not to parallelize loops without directives. This could be useful if your code contains a lot of loops, where the compiler makes the wrong decision to parallelize them. In this case, you have to tell the compiler which loop to parallelize. In the above example code, you could add one of the following directives:

- !SMP$ PARALLEL DO REDUCTION(MYSUM)
- !SMP$ INDEPENDENT
- !SMP$ ASSERT(NODEPS)

before the loop. Compiling this with:

```
xlf90_r -qsmp=noauto -O3 -qarch=pwr3 -qreport=smplist simple.f
```

produces the same parallelization as above.

C.4.2 Using the Fortran PThread Module

When you want to use pthreads in Fortran, IBM provides an interface to the pthread library, as described in section 4.9, “OpenMP Porting Considerations” on page 58. As an example using this module, the sum program from the last section is programmed using Fortran pthreads. The example is written so that it easy to understand; it is not optimized in any way. The program discussed is located near the end of this section.

Line	Comment
1-5	All global variables are put into a module.
9	To use the Fortran 90 interface to the POSIX library you have to use this module.
14-15	These are the thread structures needed in this case.
22-23	Here the attribute for the threads are initialized. The standard values are used but for the detachstate, which is set to `PTHREAD_CREATE_UNDETACHED` in order to be able to synchronize the threads.
26-31	The loop indices of each thread are calculated.

33-38 The threads are started. Each thread gets three arguments:

- A number to identify itself. The thread id `thread(i)` could also be used together with POSIX functions `f_pthread_equal()` and `f_pthread_self()`.
- The start and end index of the loop.

40-42 The synchronization. After this loop is finished, all threads but the initial one are gone.

64 This line could be a potential performance problem as it is likely to introduce *false sharing* between the threads.

False sharing occurs as two threads running on two different CPUs use two different data that are located in the same cache line. If one threads writes to his data, which is located in its cache, the whole cache line has to be transferred to the second CPU, even if the second thread doesn't use this data.

There are several solutions to this problem:

- Remove the array and do the update directly on the variable ENDSUM. This has to be done in a CRITICAL SECTION which also introduces a performance problem.
- Pad the array. That way each element is in its own cache line. This can be done with a TYPE statement, but will increase the memory use of the program.

The loop calculating the sum is moved into the subroutine TSUM, which is started by the F_PTHREAD_CREATE() call.

When linking a thread enabled program, you should use the xlf90_r (or xlf_r) call to the linker, in order to make sure that you link with the thread safe system libraries. Please note that the program now has over 60 lines of code compared to 16 lines in the version using Fortran directives.

```
+1  MODULE TDATA
+2   INTEGER, PARAMETER :: NRT=2, IMAX=10000000
+3   REAL(8), DIMENSION(IMAX) :: A
+4   REAL(8) :: MYSUM(NRT)
+5  END MODULE TDATA
+6
+7
+8  PROGRAM SEQUENTIAL
+9  USE F_PTHREAD
+10 USE TDATA
+11 IMPLICIT NONE
+12 INTEGER(4) :: I, IERR
```

```
+13 ! USED BY THE THREADS :
+14 TYPE(F_PTHREAD_T) :: THREAD(NRT)
+15 TYPE(F_PTHREAD_ATTR_T) :: ATTR
+16 INTEGER(4) :: ARG(3, NRT), IBEG(NRT), IEND(NRT)
+17 REAL(8) :: ENDSUM
+18 EXTERNAL TSUM
+19
+20 CALL INIT(A,IMAX)
+21
+22 IERR = F_PTHREAD_ATTR_INIT(ATTR)
+23 IERR = F_PTHREAD_ATTR_SETDETACHSTATE(ATTR,
   PTHREAD_CREATE_UNDETACHED)
+24
+25 ! SET UP THE LOOP COUNTERS
+26 IBEG(1)=1
+27 DO I=1,NRT-1
+28 BEG(I+1)=IBEG(I)+(IMAX/NRT)
+29 IEND(I)=IBEG(I+1)-1
+30 END DO
+31 IEND(NRT)=IMAX
+32
+33 DO I=1,NRT
+34 ARG(1,I)=I
+35 ARG(2,I)=IBEG(I)
+36 ARG(3,I)=IEND(I)
+37 IERR=F_PTHREAD_CREATE(THREAD(I),ATTR,FLAG_DEFAULT,TSUM,ARG(1,I))
+38 END DO
+39
+40 ! WAIT FOR THE THREADS TO FINISH
+41 DO I=1,NRT
+42 IERR = F_PTHREAD_JOIN(THREAD(I))
+43 END DO
+44
+45 ! BUILD THE SUM
+46 ENDSUM=0.D0
+47 DO I=1,NRT
+48 ENDSUM=ENDSUM+MYSUM(I)
+49 END DO
+50
+51 PRINT *,'SUM=',ENDSUM
+52 END PROGRAM
+53
+54 SUBROUTINE TSUM(ARG)
+55 USE TDATA
+56 REAL(8),AUTOMATIC :: PART_SUM
+57 INTEGER(4) :: I, ARG(3)
+58
```

```
+59 PART_SUM=0.D0
+60 DO I=ARG(2),ARG(3)
+61 PART_SUM=PART_SUM+A(I)
+62 END DO
+63 MYSUM(ARG(1))=PART_SUM
+64 END SUBROUTINE
```

C.4.3 Conclusions

This simple example shows that you should use the compiler directives if possible. There are several advantages to do so:

- The code is easier to program.
- Directives are less error prone than direct pthread programming.
- The code is easier to read.
- As the directives are hidden in comments, the code can still be compiled and run on a system that doesn't have a thread library.
- You can change the scheduling of a loop, without changing the code.

The advantages of the pthread library are mainly performance and the ability to parallel the code on a higher level than loops. Please note that the Fortran pthread interface it is not an industry standard, even if it is as close to the POSIX standard as a Fortran module can be.

As mentioned in section 4.9, "OpenMP Porting Considerations" on page 58, you can mix both f_pthreads and directives, so you are not forced to use one of the alternatives.

Appendix D. Special Notices

This publication is intended for developers of numerically intensive code for the RISC System/6000, for business partners and sales specialists wanting supporting metrics of the POWER3 performance potentials, and for technical specialists who require detailed product information to help demonstrate IBM's industry leading technology. See the PUBLICATIONS section of the IBM Programming Announcement for Fortran Version 5.1.1 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have

been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX®	AIX/6000®
DB2®	Power PC 603®
Power PC 604®	PowerPC 601®
PowerPC 603®	PowerPC 601e®
POWER2 Architecture	POWER3 Architecture
RISC System/6000®	RS/6000®

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

Appendix E. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

E.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 207.

- *AIX Version 4.3 Differences Guide,* SG24-2014
- *AIX 64-Bit Performance in Focus,* SG24-5103
- *RS/6000 Models E30, F40, F50, and H50 Handbook,* SG24-5143

E.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

E.3 Other Publications

These publications are also relevant as further information sources:

- *RISC System/6000 Technology*, SA23-2619
- *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, SA23-2737
- *XL Fortran for AIX Language Reference Version 5 Release 1*, SC09-2607

- *XL Fortran for AIX User's Guide Version 5 Release 1*, SC09-2606
- D. Bacon, S. Graham, and O. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, Vol. 26, 1994
- Y. Aoyama, "*RS/6000 Program Tuning Vol. 3: SMP Fortran*" (1998, in Japanese) (contact nakanoj@jp.ibm.com)
- D. Kulkarni, S. Tandri, L. Martin, N. Copty, R. Silvera, X. Tian, X. Xue, and J. Wang, "XL Fortran Compiler for IBM SMP Systems," *AIXpert Magazine*, December 1997
- *Optimization and Tuning Guide for Fortran, C, and C++"*, SC09-1705
- *General Atomic and Molecular Electronic Structure System*, M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, J.H.Jensen, S.Koseki, N.Matsunaga, K.A.Nguyen, S.Su, T.L.Windus, M.Dupuis, J.A.Montgomery J. Comput. Chem., 14, 1347-63(1993).
- *AIX Performance Tuning Guide*, SR28-5930
- *AIX General Programming Concepts: Writing and Debugging Programs*, SC23-2205

E.4 Information Available on the Internet

The following information is available on-line.

- http://www.openmp.org/
- http://www.netlib.org/blas/gemm_based/ssgemmbased.tgz
- http://www.doe.org
- http://www.llnl.gov/asci/
- http://www.netlib.org/scalapack/
- http://www.rs6000.ibm.com/software/Apps/essl.html
- http://www.rs6000.ibm.com/software/sp_products/esslpara.html
- http://www.rs6000.ibm.com/resource/aix_resource/sp_books/
- http://www.software.ibm.com/ad/fortran/xlfortran/cray.htm
- http://www.rs6000.ibm.com/software/sp_products/performance/
- http://www.rs6000.ibm.com/resource/technology/MASS/
- http://www.llnl.gov/asci_benchmarks/asci/limited/ppm/sppm_readme.html
- http://www.netlib.org/blas/gemm_based/ssgemmbased.tgz
- http://www.msg.ameslab.gov/GAMESS/GAMESS.html

- http://firewww.avl.co.at
- http://www.radioss.com
- http://www.netlib.org
- http://www.specbench.org
- http://www.cs.virginia.edu/stream
- http://www.nas.nasa.gov//NAS/NPB
- http://firewww.arl.co.at/html/346.htm

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `http://www.redbooks.ibm.com/`.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/`

- **PUBORDER** – to order hardcopies in the United States
- **Tools Disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLCAT REDPRINT
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BokkManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type the following command:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

- **REDBOOKS Category on INEWS**
- **Online** – send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** – send orders to:

	IBMMAIL	**Internet**
In United States	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** – send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax** – send orders to:

United States (toll free)	1-800-445-9269
Canada	1-800-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **408 256 5422 (Outside USA)** – ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name | Last name

Company

Address

City | Postal code | Country

Telephone number | Telefax number | VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date | Card issued to | Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of Abbreviations

API	Application Program Interface
ASCI	Accelerated Strategic Computing Initiative
BCT	Branch on Count
BHT	Branch History Table
BLAS	Basic Linear Algebra Subroutines
BLACS	Basic Linear Algebra Communications Subroutines
BT	Block Tridiagonal
BTAC	Branch Target Address Cache
CCR	Condition-Code Register
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
DASD	Direct Access Storage Device
DFL	Divide Float
DIMM	Dual Inline Memory Modules
DOE	Department of Energy
ESSL	Engineering and Scientific Subroutine Library
FMA	Floating-point Multiply-Add
FPR	Floating-Point Register
FPU	Floating Point Unit
GAMESS	General Atomic and Molecular Electronic Structure System
GPR	General-Purpose Register
ITSO	International Technical Support Organization
LFD	Load Float Double
LLNL	Lawrence Livermore National Laboratory
LRU	Least Recently Used
MASS	Mathematical Acceleration Subsystem
MFLOPS	Millions of Floating-Point Operations per Second
MPI	Message Passing Interface
MTU	Maximum Transmission Unit
NUS	Numerical Aerodynamic Simulation
NWP	Numerical Weather Prediction
PBLAS	Parallel Basic Linear Algebra Subroutines
PPM	Piecewise Parabolic Method
P2SC	POWER2 Single/Super Chip
RISC	Reduced Instruction-Set Computer
RSC	RISC Single Chip
SPEC	System Performance Evaluation Cooperative
SOI	Silicon-on-Insulator
SMP	Symmetric Multiprocessing

SP	IBM RS/6000 Scalable POWERparallel Systems
STFDU	Store Float Double with Update
TLB	Translation Lookaside Buffer
TPP	Toward Peak Performance
VP	Virtual Processor
XLF	XL Fortran
VMM	Virtual Memory Manager

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

ITSO Redbook Evaluation

RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide
SG24-5155-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
_ **Customer** _ **Business Partner** _ **Solution Developer** _ **IBM employee**
_ **None of the above**

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

_____

_____

_____

_____

What other redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

_____

_____

_____

_____

SG24-5155-00
Printed in the U.S.A.