SGI Altix Applications Development and Optimization

Part No.: AAPPL-1.0-L2.4-S-SD-W
Release Date: August 1, 2003

RESTRICTION ON USE

U.S. GOVERNMENT RESTRICTED RIGHTS LEGEND

PART NUMBER

AAPPL-1.0-L2.4-S-SD-W, August 2003

RECORD OF REVISION

Revision 0.9, Version 2.4, May 2003.
Revision 1.0, Version 2.4, August 2003.

SGI TRADEMARKS

OTHER TRADEMARKS

Other brand or product names are the trademarks or registered trademarks of their respective holders.

CONTRIBUTIONS

This course is based on the Origin Applications Development and Optimization course developed by Merdi Rafiei, updated by Kim Snyder, further updated by Gerardo Cisneros. Altix-specific material derived from various sources, including presentations by John Baron, Tom Elken, Jean-Pierre Panziera, Arthur Raefsky and Brian Sumner.

FONT CONVENTIONS

The following font conventions are used throughout this manual:
The default font is used for regular text, keyboard keys (for example, Enter, Shift, Ctrl-c), program names, system names, and toolchest names.
Bold is used for button names (for example, **Apply** and **OK**), commands, menu names, icon names, and node names.
Emphasized is used for filenames, folders, and URLs.
The `fixed width` font is used for code, system output, and onscreen prompts.
`Fixed width italics` is used for variables.
`Fixed width bold` is used for user input.

Contents

Module 1

Overview

1.1 Module Objectives

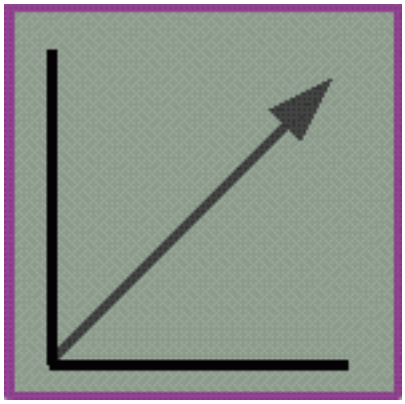
By the end of this module, you should be able to

- Compare and contrast examples of four major scalable hardware architectures
- Of these various architectures, compare and contrast
    - Scalability (interprocessor communication and I/O)
    - Operating system architectures
    - Language architectures
        - Ease of use and efficiency
    - I/O techniques
- Understand performance profiles
- Understand programming difficulties

1.2 Definition of Scalability

Scalable Computing

- Computational power that can grow over a large range of performance, while retaining compatibility



Scalability depends on the distance (time) between nodes

- Latency: Time to send first byte between nodes
- Short latency = tightly coupled system (capability solution)
- Long latency = loosely coupled system (capacity solution)

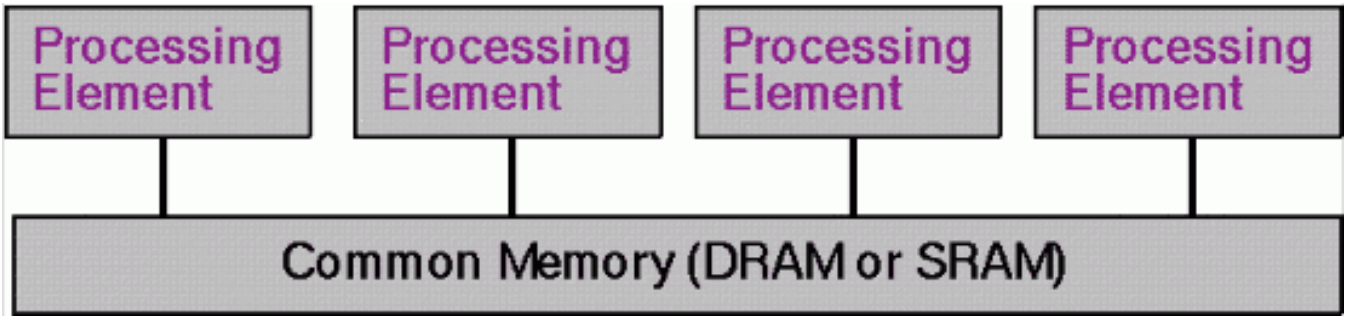
1.3 Scalability

Why is low-latency, high-bandwidth, interprocessor communication important?

- Low latency enables fine-grain parallelism
- Lower overhead = smaller segmentation
  - Split loops into small segments on multiple processors
    - Easier to find small parallel segments than large segments
  - Efficient small code segments allow for more parallelism in the code
    - More loops can be parallelized with a reasonable speedup
    - Especially important for automatic parallel code detection and generation, for example, with the automatic parallelization option in the Intel® compilers

1.4 Symmetric Multiprocessors (SMPs)

- Easiest parallel programming environment
  - All processors have equally fast (symmetric) access to memory
- Inexpensive to assemble
  - Examples: Multiprocessor workstations and PCs
- Limited scalability due to the memory access
  - Typically 2 to ∼100 processors



1.5 SMP Performance

- Latency to synchronize processors
  - Latency to communicate data among processors
    - **Fastest**: Shared register – approximately 10 nanoseconds (ns)
      - For example, CRAY T90™
    - **Fast**: Shared cache – approximately 300 ns
      - For example, Sun Ultra Enterprise 10000™ and many other servers and workstations
    - **Slowest**: Shared memory without shared cache 100s or 1,000s of ns
      - For example, multiprocessor PCs and many workstations

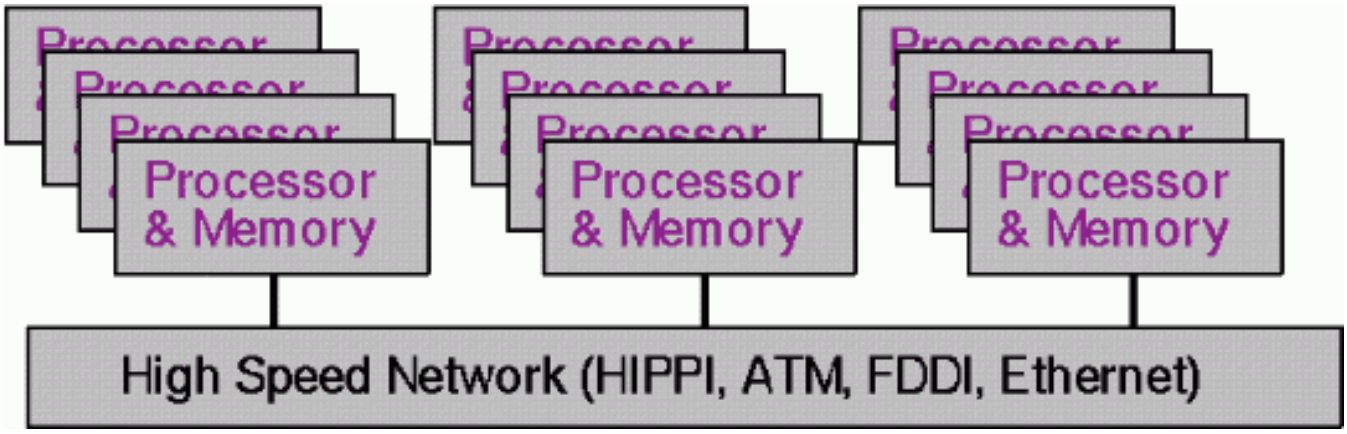
1.6 SMP Performance (continued)

Memory bandwidth

- Multiple processors exert heavy demands on memory bandwidth
- Bus-based systems may have low multiprocessor bandwidth
  - Examples: Most SMP workstations, servers and PCs
  - Fast execution in cache, but slow random access to memory
- Multiport memories can have very high bandwidths
  - Example: CRAY T90 — 880 gigabytes/second (GB/s)
  - 32 CPUs @ 440 MHz
  - 8 words per clock per CPU, 8 bytes per word = 64 bytes/clock
  - 32 CPUs * 440 MHz* 8 words * 8 bytes/word = 0.88 TB/s

1.7 Arrays (Clusters)

- Most popular, highly scalable architecture
  - Any networked computer can participate in clusters
  - Extreme example: PCs on the Internet used to crack encryption keys or find large prime numbers
- Popular technique for scaling beyond SMPs
- Easy to assemble, but often hard to use



1.8 Arrays (Clusters) (continued)

Cluster Issues

- Coarse-grain parallelism only
  - Requires thousands, millions, or billions of instructions between communication cycles
- No shared memory
  - Message passing required to span machines (capability)
  - Often used for multiple serial-job throughput (capacity)
- Long latencies
  - Typically 10s, 100s, or 1,000s of microseconds
  - Compare this with 10s to 1,000s of nanoseconds for SMPs
    - SMPs typically have 100 to 1000 times better latency than clusters

1.9 Arrays (Clusters) (continued)

Cluster Issues

- Multiple OS images
  - Hard to coordinate, schedule, and administer
  - Scheduling message-passing jobs can be difficult
- Distributed I/O
  - Hard to access remote peripherals
  - Often use NFS® or DFS for logically shared files

1.10 Arrays (Clusters) (continued)

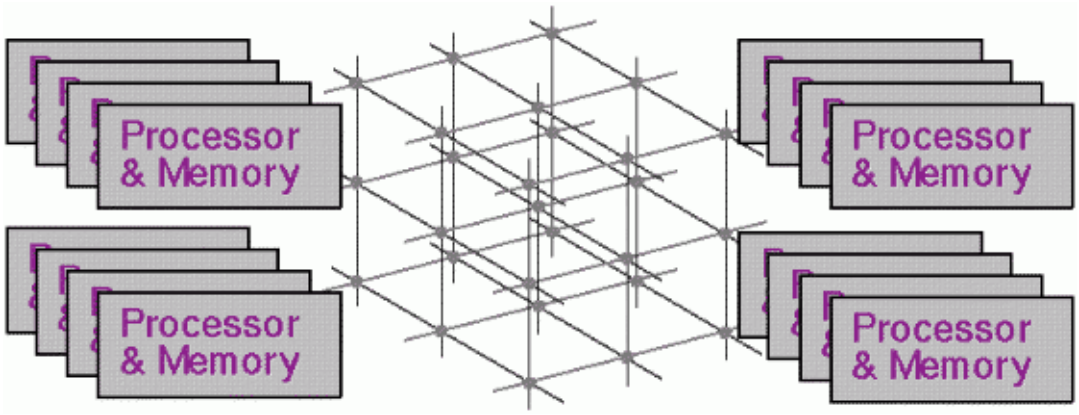
Cluster Benefits

- Easy to start small, then scale up hardware
  - Can add newer hardware to older hardware
- Nodes can be individual compute nodes
  - In addition to being members of a cluster
- High reliability
  - Parts of cluster can fail while others remain available
    - May lose access to some peripherals
- Excellent for throughput (multiple single-PE jobs)
  - A batch scheduler is typically used for scheduling multiple jobs
  - A system with a shared cache for data (e.g., a database stored in buffer cache) may still show advantages over a cluster in some throughput applications, if all jobs access same large datasets.

1.11 Massively Parallel Processors (MPPs)

Fast Distributed Memory

- Distributed memories scaling to terabytes
  - Similar to cluster programming primitives
  - Easier to program than clusters
    - Latency ∼1,000 times lower
    - Bandwidth 10s to 1,000s times faster



1.12 MPPs (continued)

Fast Distributed Memory

- Faster bandwidth
  - Typically gigabytes per second of bisection bandwidth
  - Often 100s of MB/s processor to processor
- Sometimes single-system images
  - Example: CRAY T3E™
- Scaling to thousands of processors
  - Scaling independent on low latency and high bandwidth
- Sometimes global I/O
  - Any processor can perform I/O on any device

1.13 MPP Features

- Large numbers of processors
  - 32 to 9,000+ processors
  - TeraFLOP/s of processing power
- Large memories
  - Up to terabytes of memory
- Large local memory bandwidth
  - Sum is hundreds or thousands of GB/s
- Huge I/O bandwidth
  - Up to 10s or 100s of GB/s

1.14 Scalable Symmetric Multiprocessors (S^2MPs)

- ccNUMA — Combining SMPs and MPPs
  - Cache-coherent (SMP-like)
  - Non-Uniform Memory Access (MPP-like)
- Logically programmable as an SMP
  - Every processor has fast and cache-coherent access to a common memory
  - SMP codes can run without modification
- MPP-like scalability
  - Scales to 1000s of processors

1.15 S^2MPs (continued)

ccNUMA - Scalability

- SMP codes tend to be written for 10s of processors
  - SMP codes access memory without regard for locality
  - SMP codes assume memory access speeds do not vary
  - This tends to limit the scalability of SMP codes
- S2MP can be programmed as an MPP
  - MPP programming techniques emphasize good locality of reference
  - Tends to improve scalability
- Combination: SMP-like code and high-level directives for locality
  - New area for language directives
  - Promise of scalability with ease of programming

1.16 S^2MPs (continued)

ccNUMA - Configuration

- OS: Single system image or multiple images
  - Can be configured as an SMP, MPP, or cluster
- I/O: Global device access (with single system image)
  - Any processor accessing any peripheral
  - Very fast random access from processors to disks

1.17 Scalable Operating Systems

- Single threaded kernels
  - Low-end UNIX® or PC OS
  - One system call processed at a time OK for uniprocessors or small SMPs
- Multithreaded kernels
  - High-end UNIX kernels
    - Multiple system calls executed simultaneously
    - Improves I/O dramatically on large SMPs
  - Note: OS design can have dramatic effects on system performance

1.18 Scalable Operating Systems (continued)

Arrayed (clustered) operating systems

- Full OS on each node of the cluster
  - Each node has its own system images on disk and in memory
  - Add global scheduling and administration layers
  - Add logically global I/O
    - Tie together with distributed filesystems (NFS, DFS, etc.)
    - Typically much slower I/O than on an SMP or MPP

1.19 Scalable Operating Systems (continued)

Distributed microkernels (Mach and Chorus)

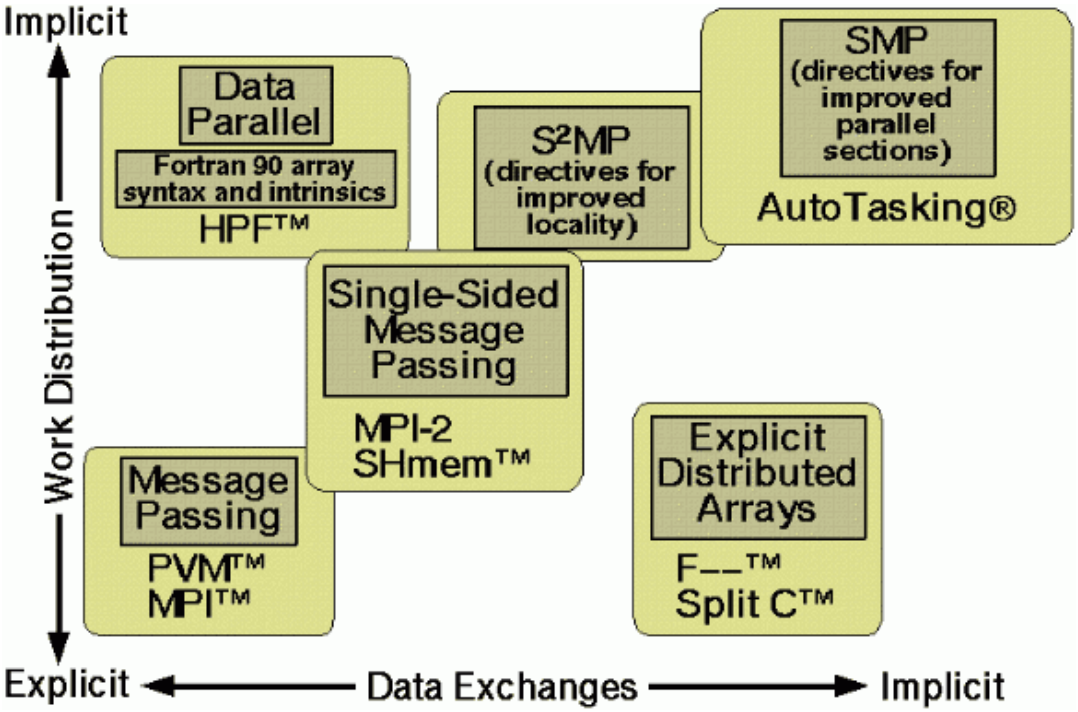
- Small kernel on each node
  - Handles local functions locally
  - Local memory and processor scheduling
- Asks for help from system nodes for global functions
  - I/O is usually a global function
- Single system image
  - Highly scalable
- Examples:
  - CRAY T3E uses a distributed Chorus™ kernel
  - Intel® Paragon® uses a distributed Mach™ kernel

1.20 Scalable Application Development

- Fortran
- C & C++
- SMP multitasking (multithreading)
- Message passing
- Single-sided message passing
- Explicit-distributed languages
- Implicit-distributed languages
- Explicit process control—POSIX threads

1.21 Scalable Application Development (continued)

Parallel Programming Models



1.22 Scalable Application Development (continued)

Shmem is single-sided message passing ported from the CRAY T3D/T3E

- Read or write any word on any process' memory at any time
  - Close to hardware (1 to 2 μs) latency on fast MPPs and S^2MPs
- Callable from Fortran, C, and C++

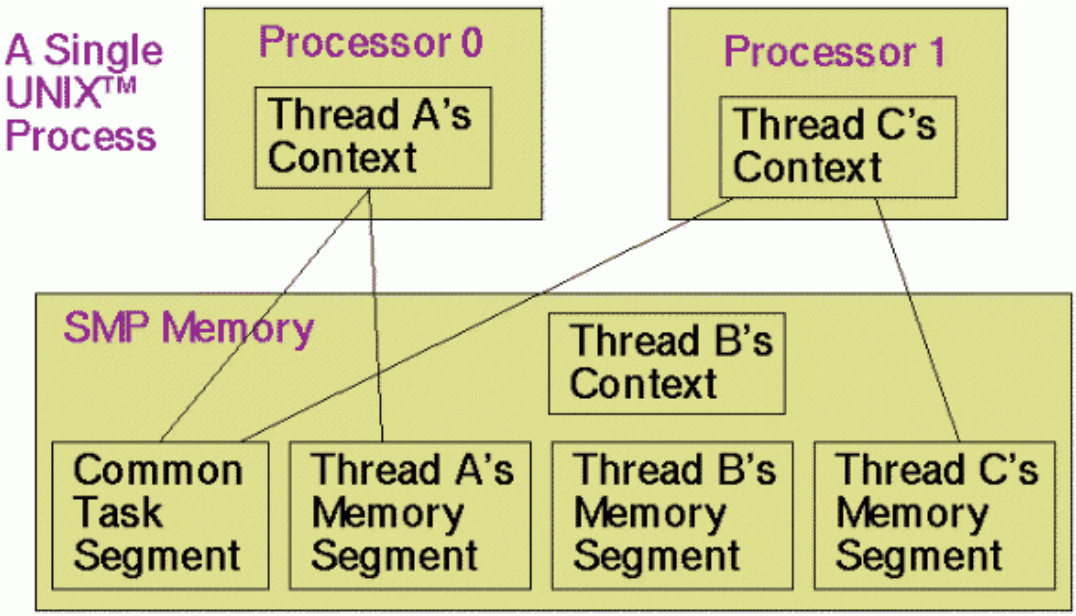
1.23 Scalable Application Development (continued)

SMP multitasking (multithreading)

- Implicit work and data distribution
- Multiple logical (and physical) processors sharing a single symmetric address space in memory
    - Requires logically shared memory
- Schedule logical processors to tasks as needed
- Efficiently mix many applications (serial and parallel) with high processor and memory efficiency
- Automatic parallelism with directives to assist auto-parallelism
- Directives added to languages
    - Fortran 77, Fortran 90, C, and C++ directives
    - OpenMP or vendor syntax
- Scalability varies depending on the application

1.24 Scalable Application Development (continued)

SMP multitasking (multithreading)



1.25 Scalable Application Development (continued)

Message passing

- Most popular parallel development style for independent software vendors (ISVs)
  - Portability is key feature for ISVs
- Harder to program than SMP parallelism
  - Coarser grain parallelism
  - Much more user code than with SMP directives
- MPI
  - De facto standard for message passing programming
- PVM
  - Works on heterogeneous (multivendor) environments

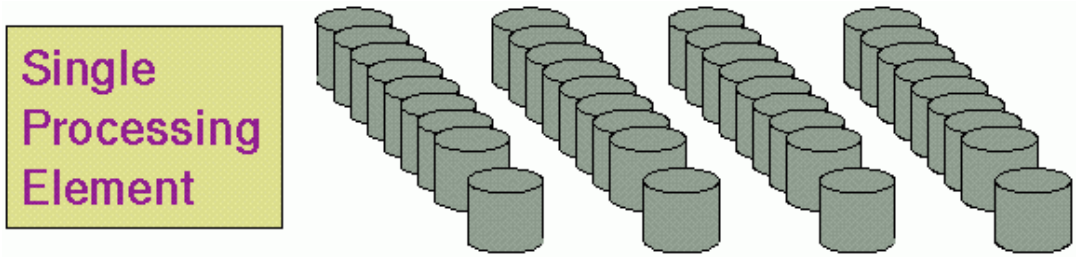
1.26 Scalable Application Development (continued)

Implicit distribution languages

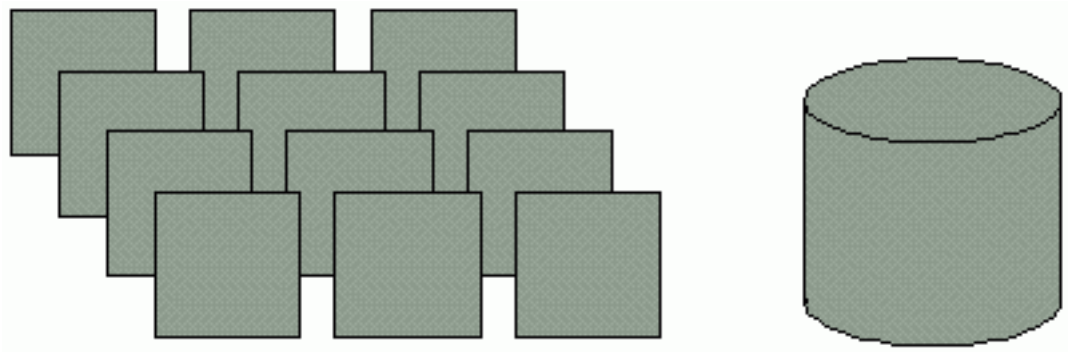
- HPF™: High Performance Fortran™
  - Designed to provide SMP-like programming on physically distributed memories
  - Programmer supplies directives to lay out memory
  - Programmer organizes work into data-parallel constructs
  - Compiler distributes data, distributes work, and takes care of interprocessor communication implicitly
  - Tends to have high overhead for communication
  - Available on most parallel machines
  - HPF is not an ISO/ANSI standard

1.27 Scalable I/O – Points of View

Parallel I/O devices with serial I/O code



Parallel I/O code with serial I/O devices

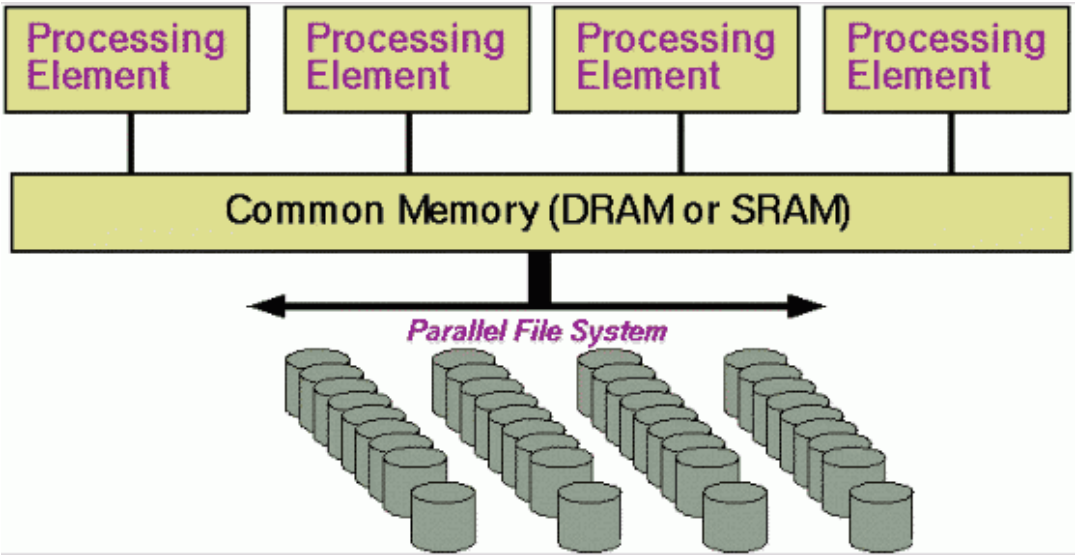


- Combine these techniques
  - Parallel I/O code with parallel I/O devices

1.28 Scalable I/O - Parallel I/O Programming

Parallel I/O on SMPs

- Global I/O easy
    - Any process can access any disk directly



1.29 Summary

SMPs: Most popular parallel platforms

- Easiest to program
- Easy global I/O
- Least expensive parallel solution
  - Sometimes difficult to scale applications
- Fastest SMP execution environment:
  - SMP programming with language directives
  - Message passing also fast, and faster than on clusters
- Fastest SMP OS: Multithreaded kernels

1.30 Summary (continued)

Clusters: Popular but Difficult Scalable Systems

- Good throughput engines (if access to shared data is “cheap”)
- Difficult distributed-parallel programming
- Slow, difficult global I/O
- Easy to reconfigure
- Good resiliency
- Only one parallel execution model: Message passing
- Rare to see (real) single system images

1.31 Summary (continued)

MPPs: Most scalable parallel platforms

- Scalability to teraFLOP/s already demonstrated
- Easier to program than clusters (better latency)
- Harder to program than SMPs (incoherent memory)
- Huge distributed memories (terabytes)
- Some have good global I/O, others not
- Most have single system image OS
- Highest performance language: gets and puts
- Expensive — losing market share

1.32 Summary (continued)

S^2MPs: SMP-like MPPs

- Theoretical scalability to teraFLOP/s
  - 100s of GFLOP/s (sustained) demonstrated to date
- Logically symmetric memory (SMP-like)
- Physically distributed memory (MPP-like)
- SMP ease of programming (cache-coherent memory)
- Message passing scalability
- Huge distributed memories (terabytes)
- Good global I/O
- Single system image

Module 2

Altix Compiling Environment

2.1 Module Objectives

After completing the module, you will be able to

- Recognize the Altix compiler flow
- Compile programs with standard options
- Create and use static and dynamic shared libraries
- Use some object file analyzers

2.2 C/C++ and Fortran Compilers

- Intel® Compilers
  - 7.0 Compilers shipped November 2002
  - 7.1 Compilers shipped March 2003
  - Fortran supports OpenMP 2.0
  - C/C++ compatible with gcc and C99 standard (subset)
- GNU Fortran and C
  - Enable easy migration from 32-bit platforms to Altix
  - Included in the standard Linux distribution
- ORC (Open Research Compiler) Fortran and C
  - Based on SGI Pro64
  - No OpenMP support
  - Available at *http://ipf-orc.sourceforge.net/*

2.3 Compiling a Program

- Compile line:

```
ecc [ option(s) ] filename.{c|C|cc|cpp|cxx|i}
efc [ option(s) ] filename.{f|for|ftn|f90|fpp}
gcc|g++ [ option(s) ] filename.{c|C|cc|cxx|m|i|ii}
g77 [ option(s) ] filename.{f|for|F|fpp}
```

- Filename requires the appropriate extension

```
% ecc main.c

% efc main.f

% g77 main.f[or]

% g++ main.C
```

2.4 Common Compiler Options

`-o <`*`file_name`*`>`	Renames the output *`file_name`*
`-g`	Produces additional symbol information for debugging
`-O` or `-O{0,1,2,3}`	Invokes the optimizer at different levels
`-I<`*`dir_name`*`>`	Look for include files in *`dir_name`*
`-c`	Compiles without invoking the linker (produces a `.o` file only)

2.5 Floating-point Underflow

Many processors do not handle denormalized arithmetic (for gradual underflow) in hardware. Whether environments support gradual underflow is very implementation dependent, and may lead to differences in numerical results.

- The Intel® compilers provide the `-ftz` option to force flushing denormalized results to zero

Notes:

Frequent gradual underflow arithmetic in a program does cause it to run very slowly, consuming large quantities of system time, which can be determined with `time`. In this case, it is best to trace the source of the underflows and fix the code, as gradual underflow is usually the source of reduced accuracy anyway.

2.6 Storage Classes and Virtual Addresses

A variable can have one of different storage classes; these are usually stored in different areas of the virtual address space of programs, and the different storage classes cause differences in behavior (particularly in some contexts in parallel programs).

automatic variables are local to each invocation of a block (e.g., when a function is called), and are discarded upon exit from the block. They are stored on the *stack*, a region at high virtual addresses that grows towards low addresses. As blocks are entered and exited in Last-in, First-out fashion, memory from the stack is always freed at the "top" of the stack (i.e., at low addresses).
Examples are variables defined within a function/procedure body without qualifiers (both in C and in Fortran).

static variables are local to a block or group of blocks, but retain their values upon reentry to a block, i.e., they cannot be discarded when control has left the block.
They are frequently stored just like external variables, with the differenc that the compiler knows they can only be referred to in the proper context called *lexical scope* or *static extent*.
Examples are variables with a `static` storage class specifier in C, a `SAVE` attribute in Fortran, and named common blocks in Fortran.

external variables exist and retain their values for the life of the entire program. They are stored on the *heap*, a region of space at low addresses in virtual space that grows to larger addresses.
Examples are variables defined out of the scope of a block in C, unnamed common in Fortran, and (obviously) C variables declared or defined with the `external` storage class qualifier.

2.7 Storage Classes and Virtual Addresses (continued)

dynamic variables are allocated during execution of the program through explicit mechanisms (e.g., `ALLOCATE` in Fortran or `malloc()` in C). They are stored on the heap, and a reference to their location is stored in another variable (which has a non-dynamic storage class).

The heap thus contains variables that can either never be reclaimed, be reclaimed explicitly (e.g.with a free() in C) or implicitly (e.g., unsaved Fortran allocatable arrays).

volatile variables are variables whose content may change by intervention of something outside the scope of the currently executing block of code. As a result, the compiler is forced to evaluate expressions involving volatile variables each time they appear in the code.

Expressions using volatile variables act as full memory barriers; a compiler is not allowed to move memory references across these when optimizing the code (and *a fortiori* cannot delay storing the variable into memory by keeping its "present" value cached in a register.

This property can be used to good effect for implicit synchronisation using variables between threads or processes in parallel programs, where the compiler might otherwise "optimize away" the synchronization operations.

Notes:

The environment on most machines has stringent limits for the size of the stack at runtime, and programs that try to allocate very large automatic variable space may not run successfully and are usually terminated with a core dump because of a SIGBUS (bus error) or SIGSEGV (segmentation violation) signal if no precautions are taken.

Locations on the stack are constantly reused, and as a result automatic variables, which are not initialized, will have unspecified values that vary from call to call.

Static variable space on the heap is initialized to zero.

In the Fortran compiler, it is possible to force all automatic storage class variables to behave as static storage class variables with the `-save` option, something used often for programs that assume zero-initialized variables.

2.8 Modules

`module` is a user interface that provides for the dynamic modification of a user's environment, i.e., users do not have to modify their `PATH` and other environment variables by hand to access the compilers, loader, libraries, and utilities. If enabled, modules can be used on the SGI Altix Series to customize the compiling environment. To access the software on the SGI Altix Series, do the following (typically `MODULESHOME` will be */opt/modules/x.y.z*, where *x.y.z* is the modules package version):

- C shell initialization (in *.cshrc*):

```
source ${MODULESHOME}/init/csh
module load intel-compilers-latest mpt-1.7-1rel
module load scsl-1.4.1-1
```

- Bourne shell initialization (in *.profile*):

```
.  ${MODULESHOME}/init/sh
module load intel-compilers-latest mpt-1.7-1rel
module load scsl-1.4.1-1
```

- Korn shell initialization (in *.profile*):

```
.  ${MODULESHOME}/init/ksh
module load intel-compilers-latest mpt-1.7-1rel
module load scsl-1.4.1-1
```

Notes:

If modules is not available on your system, its installation and use is highly recommended. The latest release may be found at *http://sourceforge.net/projects/modules/*. It is also included in SGI ProPack 2.3 and later.

2.9 Modules (continued)

- To view which modules are available on your system (any shell):

```
% module avail

--------- /sw/com/modulefiles ---------
SCCS                        ivision.R
admin                       ivision.lnk
. . .
capd                        mpt-1.7-1rel
. . .
epic.5.1                    scsl-1.4.1-1
. . .
intel-compilers-latest      transcript.4.0
. . .
```

- To load modules into your environment (any shell):

```
% module load intel-compilers-latest mpt-1.7-1rel
% module load scsl-1.4.1-1
```

- To list which modules are in your environment (any shell):

```
% module list

Currently Loaded Modulefiles:
    1) intel-compilers-latest    3) scsl-1.4.1-1
    2) mpt-1.7-1rel
```

- See **`man module`** for more options

2.10 Libraries

- Libraries are files that contain one or more object (*.o*) files
- Libraries are used to
  - Protect a company's investment in software development by allowing it to ship only object code to customers and developers
  - Simplify local software development by "hiding" compilation detail
- In UNIX, libraries are sometimes called *archives*

2.11 Static Versus Dynamic Libraries

- Static library
  - Calls to library components are satisfied at link time by copying text from the library into the executable
- Dynamic library
  - As the program starts, all needed libraries are linked into the program
  - When loaded into memory, the library can be accessed by multiple programs
  - Dynamic libraries are formed by creating a Dynamically Shared Object (DSO) file

2.12 Creating Static Libraries

- Use the archiver command, ar(1)
  `ar options [posObject] libName [object1...objectN]`
- Common archiver options

`-d`	Deletes specified object
`-m`	Moves specified object to the end of the archive
`-q`	Appends specified object to the end of the archive
`-r`	Replaces an earlier version of the object in the archive
`-t`	Lists the table of contents of the archive
`-x`	Extracts a file from the archive

2.13 Examples Using ar(1)

- Create a library with three object files:

```
% ar -q libutil.a object1.o object2.o object3.o
```

- List the contents of the archive:

```
% ar -t libutil.a
object1.o
object2.o
object3.o
```

- Add a file to the archive:

```
% ar -q libutil.a object4.o
```

- Replace an object with a newer version:

```
% ar -r libutil.a object4.o
```

- Delete an object from the archive:

```
% ar -d libutil.a object4.o
```

2.14 Using Static Libraries

- To use a static library, include the library on the compile line:

  ```
  % gcc -o myprog myprog.c func1.o libutil.a
  ```

- If the library is named *lib<name>.a* and it is not in a standard library directory, use the `-L<dir>` and `-l<name>` options:

  ```
  % gcc -o myprog myprog.c func1.o -L./libs -lutil
  ```

  - In the above example, if both a dynamic and static libraries exist in the same directory, the dynamic library is chosen first

- To use the static version of standard libraries, use the full path name of the library or the `-static` option:

  ```
  % gcc myprog.c /usr/lib/libm.a
  ```

  or

  ```
  % gcc myprog.c -static -lm
  ```

2.15 Creating Dynamic Libraries

- To create a dynamic library with a series of object files:

  % **ld -shared object1.o object2.o -o libops.so**

- To create a DSO from an existing static library:

  % **ld -shared --whole-archive libutil.a -o libutil.so**

2.16 Using Dynamic Libraries

- To use a dynamic library, include the library on the compiler line:

```
% gcc -o myprog myprog.c func1.o libops.so
% gcc -o myprog myprog.c func1.o -L./libs -lops
% gcc myprog.c -lm
```

- When using `-l<string>` and, within a directory, both *lib<string>.a* and *lib<string>.so* exist, the DSO library is used
- If your dynamic library is not in the standard directories, the run-time linker `ld.so` cannot find it unless you
  - Use the `-rpath <directory>` option during linking:

    ```
    % gcc -o myprog myprog.c -Wl,-rpath -Wl,./libs -L./libs
    -lops
    ```

    or

  - Set the LD_LIBRARY_PATH environment variable before running the executable:

    ```
    % setenv LD_LIBRARY_PATH ./libs
    % myprog
    ```

2.17 C/C++ Libraries

Class libraries included with the Intel compiler

- `libguide.a, libguide.so`
  - for support of OpenMP-based program
- `libsvml.a`
  - short vector math library
- `libirc.a`
  - Intel® support for PGO (profile-guided optimization) and CPU dispatch
- `libimf.a, libimf.so`
  - Intel® math library
- `libcprts.a, libcprts.so`
  - Dinkumware C++ library
- `libunwind.a, libunwind.so`
  - Unwinder library
- `libcxa.a, libcxa.so`
  - Intel® runtime support for C++ features

2.18 Getting Information About Object Files and Libraries

Command	Purpose
`file`	Lists the general properties of the file
`size`	Lists the size of each section of the object file
`readelf`	Lists the contents of an ELF object file
`ldd`	Lists shared library dependencies
`nm`	Lists the symbol table information
`dis`	Disassembles the source code
`strip`	Removes the symbol table and relocation bits from an object file
`c++filt`	Demangles names for C++

2.19 Listing File Properties and File Size

- Use `file`(1) for information about object files and executables

```
% file main
main:  ELF 64-bit LSB executable, IA-64, version 1,
dynamically linked (uses shared libs), not stripped
```

2.20 Estimating Memory Requirements of a Program

- `size`(1) reports the size of a program
- Reported size is the minimum space required

```
$ size main
   text        data     bss     dec     hex filename
   3254         788      80    4122    101a main
$ size -A main
main  :
section              size                  addr
.interp                24   4611686018427388360
.note.ABI-tag          32   4611686018427388384
.hash                 144   4611686018427388416
.dynsym               408   4611686018427388560
.dynstr               244   4611686018427388968
.gnu.version           34   4611686018427389212
.gnu.version_r         64   4611686018427389248
.rela.dyn              96   4611686018427389312
.rela.IA_64.pltoff    192   4611686018427389408
.init                 272   4611686018427389600
.plt                  448   4611686018427389888
.text                1024   4611686018427390336
.fini                 112   4611686018427391360
.rodata                 8   4611686018427391472
.opd                   48   4611686018427391488
.IA_64.unwind_info     56   4611686018427391536
.IA_64.unwind          48   4611686018427391592
.data                   4   6917529027641085592
.ctors                 16   6917529027641085600
.dtors                 16   6917529027641085616
.got                  128   6917529027641085632
.dynamic              464   6917529027641085760
.sdata                 32   6917529027641086224
.IA_64.pltoff         128   6917529027641086256
.sbss                   8   6917529027641086384
.bss                   72   6917529027641086392
.comment              232                     0
.debug_pubnames        37                     0
.debug_info          4741                     0
.debug_abbrev         252                     0
.debug_line             0                     0
Total                9384
```

2.21 Getting Information About ELF Files

- Use `readelf`(1) to inspect sections of an ELF (Executable and Linking Format) file:

  ```
  readelf [options] filename1 [filename2...]
  ```

- You can print the ELF header, section headers, DSO library list, library information, and so on, by specifying different options (see man page)
- List dynamic shared library list using `readelf` or `ldd`:

```
% readelf -d main

Dynamic segment at offset 0xf40 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libutil.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
 0x000000000000000f (RPATH)              Library rpath: [.]
 0x000000000000000c (INIT)               0x40000000000006a0
 . . .
 0x0000000000000000 (NULL)               0x0
% ldd main
       libutil.so => ./libutil.so (0x2000000000048000)
       libc.so.6.1 => /lib/libc.so.6.1 (0x2000000000204000)
       /lib/ld-linux-ia64.so.2 => /lib/ld-linux-ia64.so.2 (0x2000000000000000)
```

2.22 Listing Global Symbol Table Information

- Use `nm`(1) to list global symbol table information for object files and archives

```
% nm example.o
                 U func5
0000000000000000 G global_initialized
0000000000000050 C global_uninitialized_array
0000000000000000 T main
                 U sqrt
```

Abbreviation	Description
B	Uninitialized data section symbol (bss)
C	Common symbol (uninitialized)
D	Initialized data section symbol
G	Small object initialized data section symbol
U	Undefined symbol
T	Text (code) section symbol

2.23 Using nm(1) To Find Unresolved Symbols

- When compiling a program, you may get an error from the linker that a symbol is unresolved (it cannot find where the symbol is defined):

```
% cc myprog.c -lmy_lib
ld:
Unresolved:
Missing_Symbol
```

- If you do not know where `Missing_Symbol` is defined, you can search available object files and libraries for the symbol
  - For example, use a combination of `nm` and `grep` to search for this symbol in local object files, libraries, and DSOs:

```
% foreach i (*.o *.a *.so)
? nm  $i | grep Missing_Symbol | grep ' T '
? echo $i
? echo
? end
```

2.24 Disassembling Object Files

- To disassemble your object files or executables into IPF assembly language
  - See how the optimizer is rearranging your source code
  - Hand-tune in assembly
- To disassemble an object file, use `objdump`(1):

```
% objdump -d filename1 [filename2...]
```

  - Use the `-S` option to mix source, if possible, with the assembly code

2.25 Stripping Executables of Symbol Table Information

- Use `strip`(1) to remove all symbol table information, thereby decreasing the size of your executables:

```
% strip [options] filename1 [filename2...]
% ecc -g -o main main.o libutil.a -lm
% ls -l main
-rwxr-x---    1 gerardo  sdiv       259839 Apr 15 10:45 main*
% strip main
% ls -l main
-rwxr-x---    1 gerardo  sdiv       211912 Apr 15 10:45 main*
```

- Stripped executables cannot be debugged symbolically, and `nm`(1) gives an error
- Stripping also provides a measure of intellectual property protection when distributing binary code

2.26 Name Mangling and c++filt

- C++ use of polymorphism and overloading requires compiler generation of unique function names for different instantiations called name mangling
- You can use the `c++filt` tool to help demangle these names for the programmer
- For example, pipe the results of `nm` through `c++filt`:

  ```
  % nm myc++prog | /usr/lib/c++/c++filt
  ```

- `nm` also has a `-C` option that provides demangling

  ```
  % nm -C myc++prog
  ```

Lab: Using the SGI Altix Compiler Environment

Objectives

- Compile a program using the SGI compilers
- Create and use an archive library
- Create and use a DSO
- Get information from object files

Lab: Introduction to the SGI Altix Programming Environment

Creating and Using an Archive Library

In this exercise, you compile the program `main`, which is made up of a series of object files (for example, *main.o*, *object1.o*, *object2.o*).

1. Copy the Altix/SGI_programming_environment directory to your home directory.
2. Change to the SGI_programming_environment/labs/{f,c}src directory.
3. Create the object files main.o, object1.o, object2.o, object3.o, object4.o, object5.o without linking them into an executable.
   (There are C and FORTRAN objects available). List the command you used:

   _____

   _____

4. Link the object files. List the command you used:

   _____

   _____

5. Run `main` and note its output.

Creating and Using an Archive Library

In this exercise, you create an archive library from the *object?.o* files used in the previous exercise.

1. Create an archive called libutil.a by appending the file *object1.o* to it. List the command you used:

   _____

   _____

2. Add *object{2-4}.o* to the archive.
3. List the contents of the archive. List the command you used:

   _____

   _____

4. Relink the main program using the archive. Run `main` to verify that it works.

Creating and Using a DSO

1. Make libutil.a a DSO (call it libutil.so). List the command you used:

   _____

   _____

2. Relink `main` using the DSO.
3. Rerun `main` to verify that it works.

Getting Information From Object Files

In this exercise, you get information about object files.

1. What kind of file is *libutil.so*? Which command did you use to determine this?

   _____

   _____

2. Compile the program *example.{c,f}*. Which symbol is undefined?

   _____

3. Write a command to search the local directory for all object files, archive libraries, and DSOs that may contain the symbol. Which command(s) did you use?

   _____

   _____

   _____

4. Recompile *example.{c,f}* with the identified library to make sure it works. Run the executable.

Module 3

SGI Altix Architecture

3.1 Module Objective

After completing this module, you should be able to identify major hardware components of the SGI Altix systems.

3.2 SGI Altix 3000

Altix 3300

- 4-12 Intel® Itanium® 2 processors
  - 900 MHz with 1.5 MB tertiary caches (“McKinley”)
  - 1.3 MHz with 3 MB tertiary caches (“Madison”)
- Fully modular design
- Up to 96 GB of shared memory

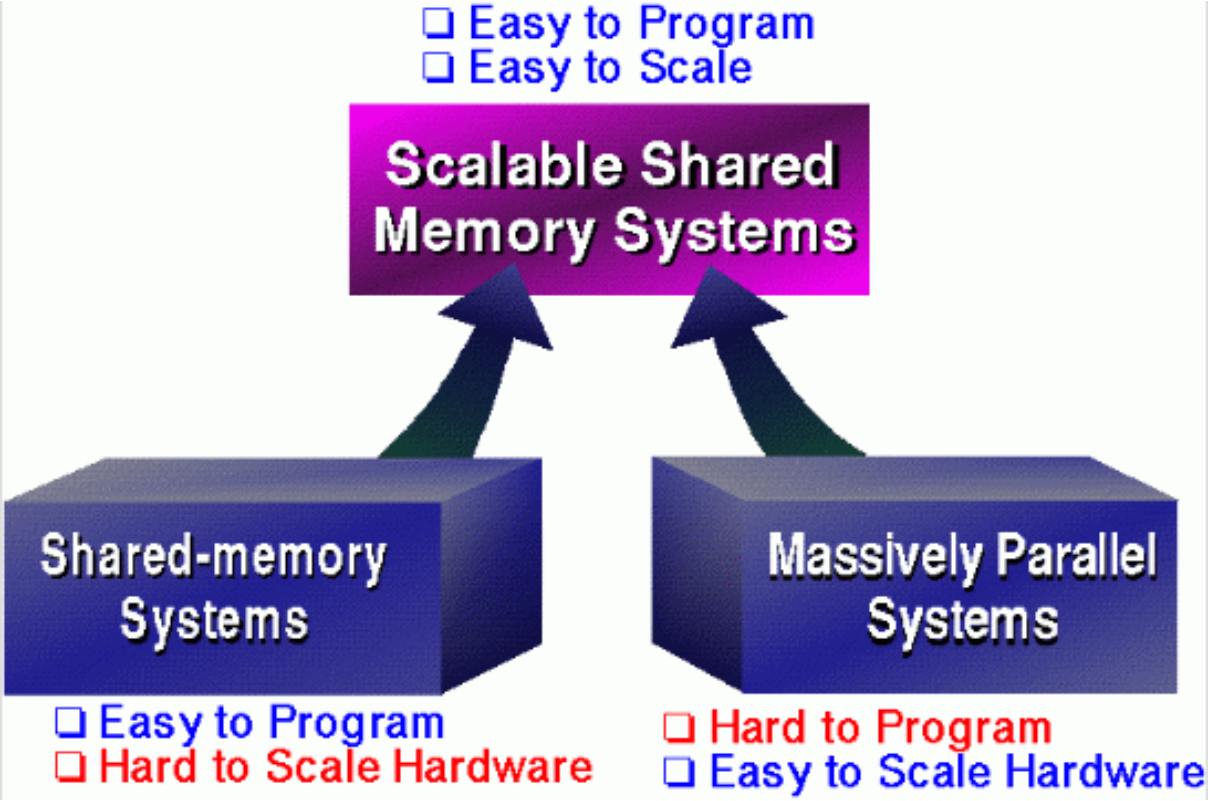
Altix 3700

- 16-64 Intel® Itanium® 2 processors in a single node
- Choice of processors
  - 900 MHz/1.5 MB L3 cache or 1 GHz/3 MB L3 cache (“McKinley”)
  - 1.3 GHz/3 MB L3 cache or 1.5 GHz/6 MB L3 cache (“Madison”)
- Fully modular design
- Up to 512 GB of shared memory

Altix 3700 Supercluster

- 1-8 nodes
- Each node up to 64 Intel® Itanium® 2 processors (up to 512 processors)
- Memory, up to 4 TB, is globally addressable across nodes

3.3 Scalable Cache Coherent Shared Memory

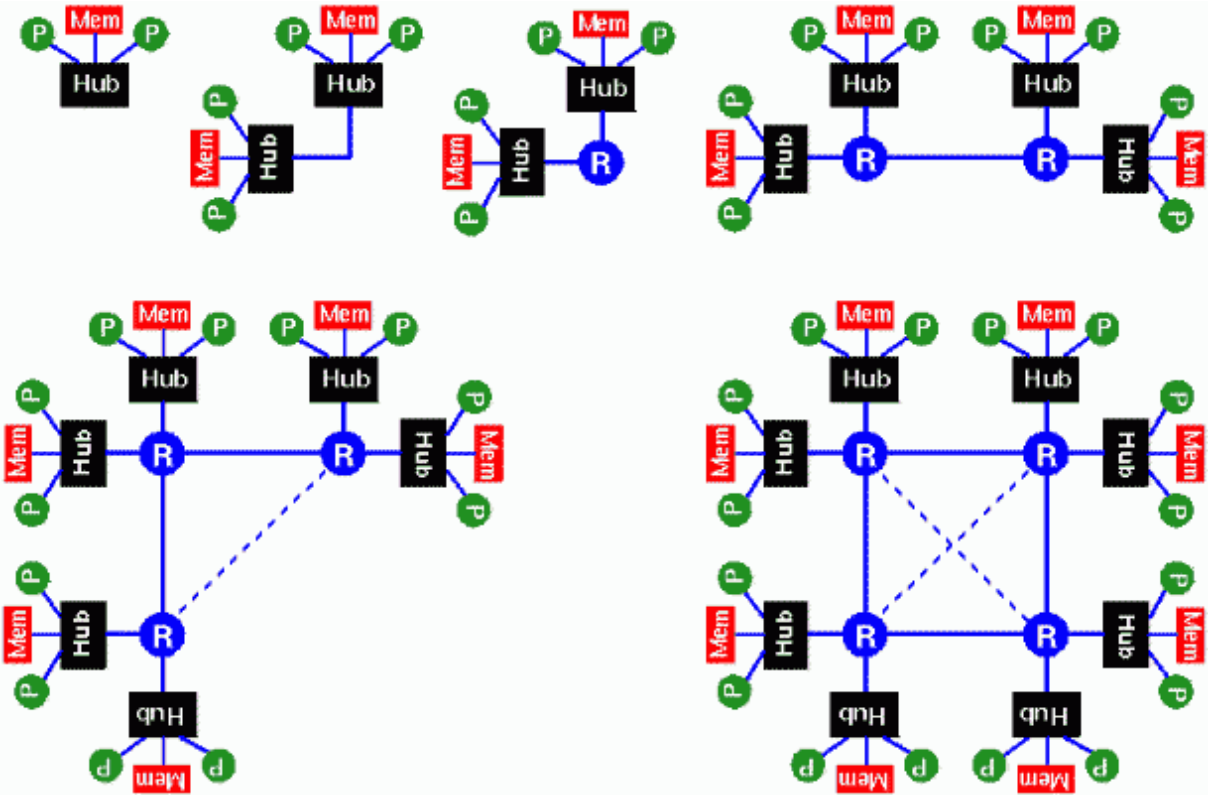


Notes:

Although a bus-based shared memory machine offers a familiar multitasking, multiuser environment and relative ease in parallel programming, the finite bandwidth of the bus can become a bottleneck and limit scalability. Distributed memory message-passing machines cure the scalability problem by eliminating the bus, but thrown out with the bus is the shared memory model and the ease of programming it allows. Ideally, you want one machine that combines the best of both approaches; this is what the scalable shared memory of the SGI Altix systems provide.

The SGI Altix uses physically distributed memory; there is no longer a common bus that could become a bottleneck. Memory bandwidth grows as processors and memory are added to the system, but the SGI Altix hardware treats the memory as a unified, global address space; thus it is shared memory just as in a bus-based system. The programming ease is preserved, but the performance is scalable. The improvement in scalability is easily observed in practice.

3.4 Shared Memory Without a Bus



Notes:

To understand how the NUMAflex scalable shared memory multiprocessor (S^2MP) architecture works, we first look at how the building blocks of an SGI Origin 2000 system are connected.

The above represent SGI Origin 2000 systems ranging from 2 to 16 processors. We start by considering the two-processor system in the upper left corner. This is a single SGI Origin 2000 node. It consists of one or two processors, memory, and a device called the hub. The hub is the piece of hardware that carries out the duties that a bus performs in a bus-based system; namely, it manages each processor's access to memory and I/O. This applies to accesses that are local to the node containing the processor, as well as to those that must be satisfied remotely in multinode systems.

The smallest SGI Origin 2000 systems consist of a single node. Larger systems are built by connecting multiple nodes. A two-node system is shown in the upper middle of the illustration. Because information flow in and out of a node is controlled by the hub, connecting two nodes means connecting their hubs. In a two-node system this simply means wiring the two hubs together. The bandwidth to local memory in a two-node system is double that in a one-node system: the hub on each of the two nodes can access its local memory independently of the other. Access to memory on the remote node is a bit more costly than access to local memory because the request must be handled by both hubs. A hub determines whether a memory request is local or remote based on the physical address of the data accessed.

When there are more than two nodes in a system, their hubs cannot simply be wired together. In this case, additional hardware is required to control information flow between the multiple hubs. The hardware used for this in SGI Origin 2000 systems is called a router. A router has six ports, so it may be connected to up to six hubs or

other routers. In a two-node system, one may employ a router to connect the two hubs rather than wiring them directly together; this is shown adjacent to the other two-node configuration in the illustration. These two different configurations behave identically, but because of the router in the second configuration, information flow between the two hubs takes a little more time. The advantage, though, of the configuration with the router is that it may be used as a basic building block from which to construct larger systems.

In the upper right corner of the illustration, a four-node (or, equivalently, eight-processor) system is shown. It is constructed from two of the two-node-with-router building blocks. Here, the connection between the two routers allows information to flow and, hence, the sharing of memory between any pair of hubs in the system. Because a router has six ports, it is possible to connect all four nodes to just one router, and this one-router configuration can be used for small systems. That is a special case, and in general, the two-router implementation is used because it conveniently scales to larger systems.

Two such larger systems are shown on the lower half of the illustration; these are 12- and 16-processor systems, respectively. From these diagrams, you can begin to see how the router configurations scale: each router is connected to two hubs, routers are then connected to each other forming a binary *n*-cube, or hypercube, where *n*, the dimensionality of the router configuration, is the base-2 logarithm of the number of routers. For the four-processor system, n is zero, and for the eight-processor system, the routers form a linear configuration, and *n* is one. In both the 12- and 16-processor systems, *n* is two and the routers form a square; for the 12-processor system, one corner of the square is missing. Larger systems are constructed by increasing the dimensionality of the router configuration and adding up to two hubs with each additional router. Systems with any number of nodes may be constructed by leaving off some corners of the *n*-dimensional hypercube. We will see these larger configurations later.

The key point here is that the hardware allows the physically distributed memory of the system to be shared, just as in a bus-based system; however, because each hub is connected to its own local memory, memory bandwidth is proportional to the number of nodes. As a result, there is no inherent limit to the number of processors that can be used effectively in the system. In addition, because the dimensionality of the router configuration grows as the systems get larger, the total router bandwidth also grows with system size (proportional to 2*n*, where *n* is the dimensionality of the router configuration). Thus, systems may be scaled without fear that the router connections will become a bottleneck.

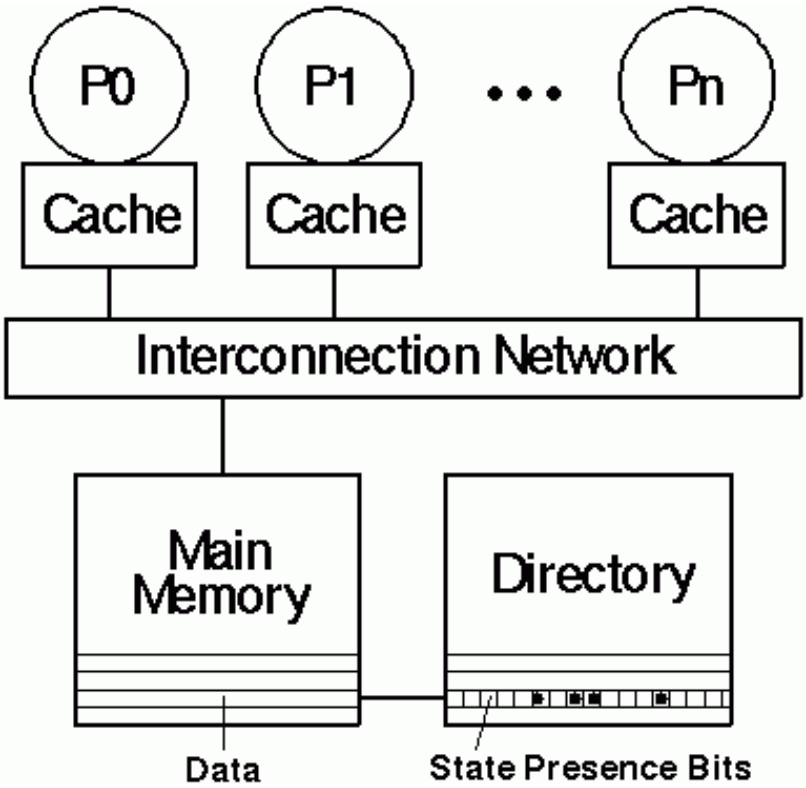
To allow this scalability, however, one nice characteristic of the bus-based shared memory systems has been sacrificed—namely, the access time to memory is no longer uniform: it varies depending on how far away the memory being accessed is in the system. The two processors in each node have quick access through their hub to their local memory. Accessing remote memory through an additional hub adds an extra increment of time, as does each router the data must travel through. But several factors combine to smooth out these nonuniform memory access (NUMA) times:

- The hardware has been designed so that the incremental costs to access remote memory are not large.
- The processors operate on data that are resident in their caches. If programs use the caches effectively, the access time to memory, whether it is local or remote, is unimportant because the vast majority of memory accesses are satisfied from the caches.
- Through operating system support or programming effort, the memory accesses of most programs can be made to come primarily from local memory. Thus, in the same way that the caches can make local memory access times unimportant, remote memory access costs can be reduced to an insignificant amount.
- The processors can prefetch data that are not cache resident. Other work can be carried out while these data move from local or remote memory into the cache; thus the access time can be hidden.

The architecture of NUMAflex-based systems, then, provides shared memory hardware without the limitations of traditional bus-based designs.

3.5 Cache Coherence

- Traditional (snoopy) cache coherence schemes do not scale
  - Must broadcast all memory requests
- Directory solves this problem



Notes:

Each CPU in an SGI Altix 3000 system has a private third-level cache memory of 1.5 MB to 6 MB. To achieve good performance, the CPU always fetches and stores data in its cache. When the CPU refers to memory that is not present in the cache, there is a delay while a copy of the data is fetched from memory into the cache. The CPU's use of cache memory, and its importance to good performance, is covered in a later section.

The point is that there can be as many independent copies of a memory location as there are CPUs in the system. If every CPU refers to the same memory address, every CPU's cache will have a copy of that address. (This could be the case with certain kernel data structures, for example.) But what if one CPU then modifies that location? All the other cached copies of the location have become invalid. The other CPUs must be prevented from using what is now "stale data." This is the issue of cache coherence—how to ensure that all caches reflect the true state of memory.

Cache coherence is not the responsibility of software (except for kernel device drivers, which must take explicit steps to keep I/O buffers coherent). For performance to be acceptable while maintaining the image of a single shared memory, cache coherence must be managed in hardware. Cache coherence is also not the responsibility of the

CPU. Cache management is performed by auxiliary circuits that are part of the hub chip that is at the heart of the NUMAflex interconnect.

The cache coherence solution in the SGI ccNUMA systems is fundamentally different from that used in the earlier Power Challenge and many current bus-based systems. Bus-based systems can use a "snoopy" scheme in which each CPU observes every memory access that moves on the bus. When a CPU observes another CPU modifying memory, the first CPU can automatically invalidate and discard its now-stale copy of the changed data. The ccNUMA systems have no central bus, and there is no way for a CPU in one node to know when a CPU in a different node modifies its own memory or memory in yet a third node. SGI Origin and Altix systems could mimic what is done in bus-based systems; that is, all memory accesses could be broadcast to all nodes so that stale cache lines could be invalidated. But since all nodes would need to broadcast their memory accesses, this coherency traffic would grow as the square of the number of nodes. As more nodes are added to the system, all the available internode bandwidth would eventually be consumed by this traffic. As a result, the system would suffer from limitations similar to bus-based computers: it would not be scalable. So to permit the system to scale to large numbers of processors, ccNUMA systems use what is called a directory-based cache coherency scheme.

Memory is organized by cache lines of 128 bytes. As shown above, associated with the data bits in each cache line is an extra set of bits, the state presence bits—one bit per node (in large systems one state presence bit may represent more than one node, but the principle is the same), and a single number, the number of a node that owns the line exclusively. Whenever a node requests a cache line, its hub initiates the memory fetch of the whole line from the node that contains that line. When the cache line is not owned exclusively, the line is fetched and the state presence bit for the calling node is set. As many bits can be set as there are nodes in the system.

When a CPU wants to modify a cache line, it must gain exclusive ownership. To do so, the hub retrieves the state presence bits for the target line and sends an invalidation event to each node that has made a copy of the line. Typically, invalidations need to be sent to only a small number of other nodes; thus, the coherency traffic only grows proportionally to the number of nodes, and there is sufficient bandwidth to allow the system to scale. Both CPUs on each node sent an invalidation request discard their cached copy of the line. The number of the updating node is set in the directory data for that line as the exclusive owner. When a CPU no longer needs a cache line (for example, when it wants to reuse the cache space for other data), the hub gives up exclusive access, if it has it, and clears its state presence bit.

When a CPU wants to read a cache line and the line is exclusively owned, the hub requests a copy of the line from the owning node. This retrieves the latest copy without waiting for it to be written back to memory. There are also protocols by which CPUs can exchange exclusive control of a line when both are trying to update it and protocols that allow kernel software to invalidate all cached copies of a range of addresses. The directory-based cache coherency mechanism uses a lot of dedicated circuitry in the hub to ensure that many CPUs can use the same memory, without race conditions, at high bandwidth. As long as memory reads far exceed memory writes (the normal situation), there is no performance cost for maintaining coherence. However, when two or more CPUs alternately and repeatedly update the same cache line, performance suffers, because every time either CPU refers to that memory, a copy of the cache line must be obtained from the other CPU. This performance problem is generically referred to as cache coherency contention, of which there are two variations:

- Memory contention, in which two (or more) CPUs try to update the same variables.
- False sharing, in which the CPUs update distinct variables that only coincidentally occupy the same cache line.

Memory contention occurs because of the design of the algorithm; correcting it generally involves an algorithmic change. False sharing is contention that arises by coincidence, not design; it can usually be corrected by modifying data structures.

Part of performance tuning of parallel programs is recognizing cache coherency contention and eliminating it.

3.6 Altix 3000 Architecture

- Modular architecture: No backplanes or midplanes
- Built from “Bricks”
  - C-brick: 4 Intel® Itanium® 2 processors, 2 to 32 GB memory
  - M-brick: CPU-less C-brick, up to 32 GB of memory
  - Ix- Brick: Basic I/O and System disk
  - Px-Brick: PCI-X expansion module
  - D-Brick2: JBOD storage array

Notes:

The SGI Altix 3000 is based on the the architecture of the third generation MIPS-processor-based SGI Origin 3000 ccNUMA system. The first generation was the Stanford DASH project, and the second generation is the SGI Origin 2000. It incorporates high-density nodes (4 processors per node) and more, higher-bandwidth ports per crossbar ASIC (initially in the hubs, later in the router). Higher density and lower latency through the routers help make the worst-case latency of the Altix 3000 much flatter than the worst-case latency in previous or competing systems.

3.7 Altix 3000 Limits

Category	Minimum	Maximum
Number of Processors	4	512 (8 64-CPU clustered nodes)
C-brick Memory Capacity	2 GB	32 GB
System Main Memory Capacity	2 GB (1 C-brick)	4 TB (128 C-bricks)
Number of I/O Channels	2	256

Notes:

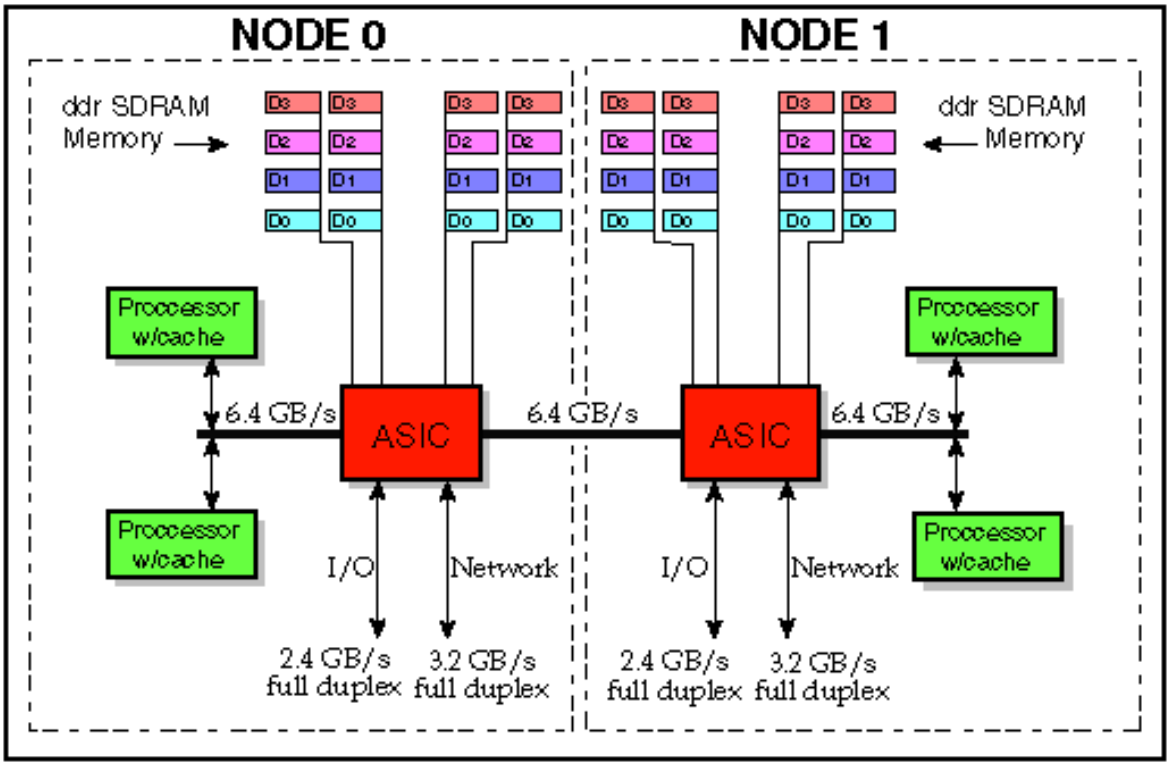
An SGI Altix 3000 C-brick has two nodes with two processors each. Two Intel® Itanium® 2 processors, each with 1.5 to 6 Mbytes of private tertiary cache, are connected through the 6.4 GB/s front-side bus (FSB) to a SHub ASIC. This SHub ASIC acts as a crossbar between the processors, local SDRAM memory, the network interface, and the I/O interface. Within a C-Brick, the two SHubs are connected internally by a 6.4 GB/s channel. An external 3.2/6.4 GB/s port on each of the SHubs is connected to the SHub in another C-Brick (in the SGI Altix 3000) or to one of the router planes. The router planes are constructed out of 8-ported routers, each port operating at 3.2 GB/s in NUMAlink3, in a fat-tree topology. The modularity of the DSM approach combines the advantages of low entry-level cost with global scalability in processors, memory, and I/O.

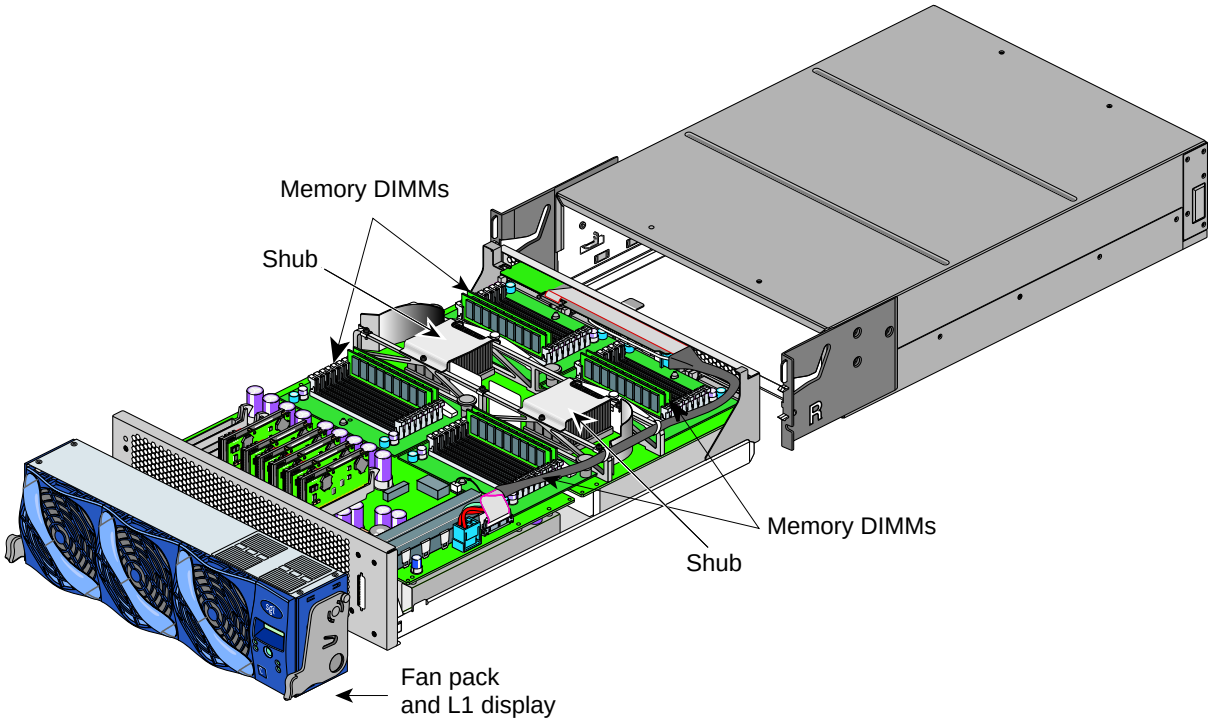
The SGI Altix 3000 series uses a PCI-X-based I/O subsystem as its primary I/O protocol.

3.8 The C-Brick

The Altix 3000 C-brick contains:

- 4 Intel® Itanium® 2 processors
  - 900 MHz with 1.5 MB L3 cache or 1 GHz with 3 MB L3 cache (“McKinley”)
  - 1.3 GHz with 3 MB L3 cache or 1.5 GHz with 6 MB L3 cache (“Madison”)
- 2 SHub ASICs
- 2 MB to 32 GB (32 memory DIMM slots)
- 4 L3 caches (one per processor)
- Two internal 3.2 GB/s (each direction) NUMAlink channels (one per SHub)
- Two NUMAlink 3 channel (1.6 GB/s in each direction; one per SHub)
- Two Xtown2 I/O channels (1.2 GB/s in each direction, each channel; one per SHub)
- One L1 controller and LCD display
- One USB port for system controller support
- One serial console port (DB9 connector)


NODE 0
NODE 1
ddr SDRAM Memory
ddr SDRAM Memory
D3
D2
D1
D0
Proccessor w/cache
6.4 GB/s
ASIC
6.4 GB/s
ASIC
6.4 GB/s
I/O
Network
I/O
Network
2.4 GB/s full duplex
3.2 GB/s full duplex
2.4 GB/s full duplex
3.2 GB/s full duplex


Memory DIMMs
Shub
Memory DIMMs
Shub
Fan pack and L1 display

Notes:

The node electronics, L1 controller, and power regulators are contained on a half-panel power board. The two SHubs, four processors, and four processor power pods are housed on a second half-panel printed circuit board (PCB). This second PCB also provides connections for the four memory daughter cards.

The Altix 3000 series systems use commodity off-the-shelf memory DIMMs. The DIMM card is a JEDEC standard 184-pin card.

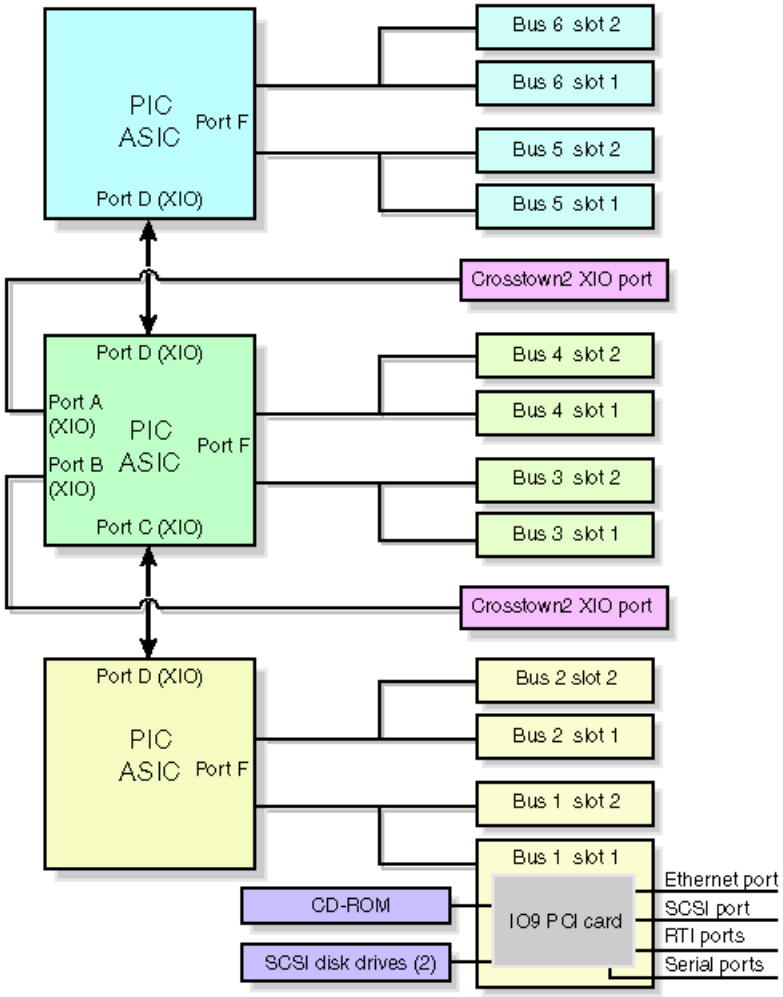
The C-brick has the following restrictions:

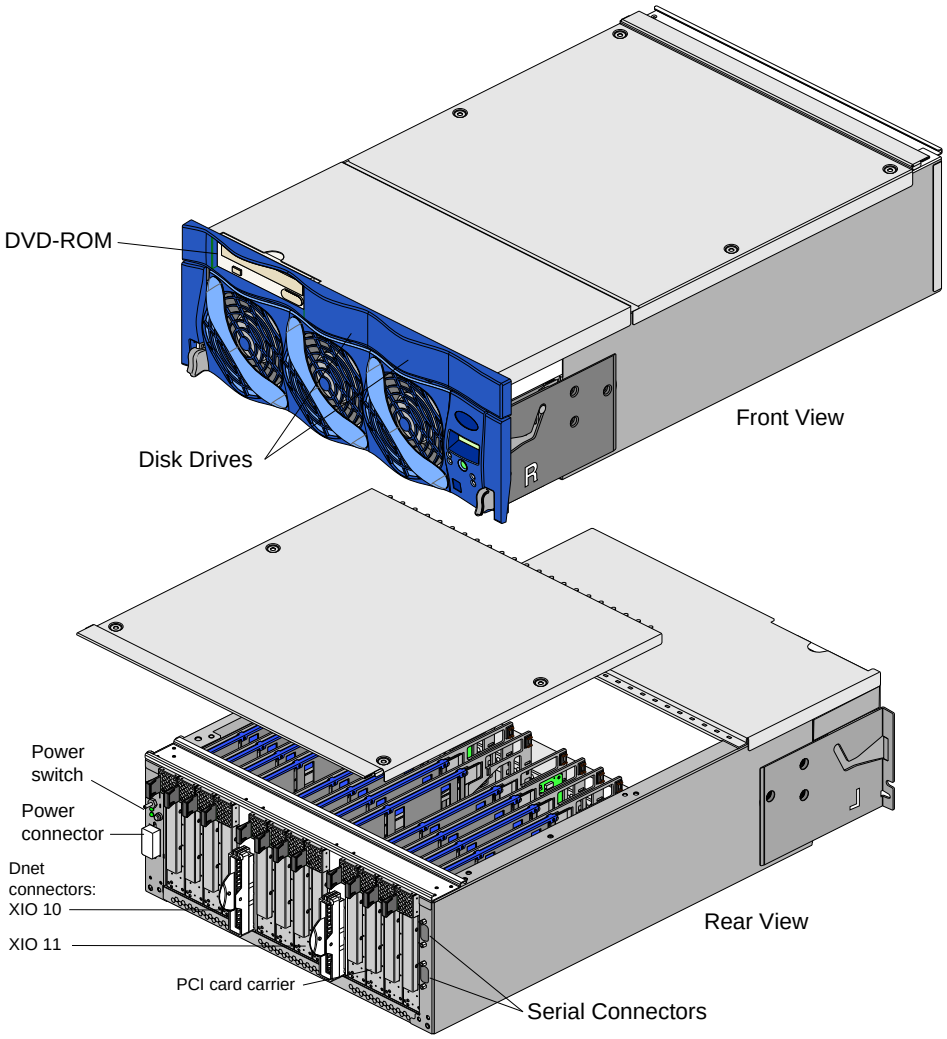
- All processors within the C-brick must be the same frequency; however, C-bricks within a partition or system can have different processor speeds.
- All processor revisions must be the same within a processor node.
- The processor revisions of CPUs between processor nodes can differ by not more than 1.
- All memory DIMMs within a memory bank must be the same speed, capacity, and use the same chip technology.
- Different logical banks within a C-brick can have different DIMM capacities and chip technologies.
- Memory DIMMs must be added in groups of eight DIMMs.

3.9 The IX-Brick

The Altix 3000 IX-brick contains:

- Two SCSI disk drives. These customer-removable, sled-mounted SCSI disk drives are used to house the operating system and other application software.
- DVD-ROM device
- 12 PCI-X slots
- SCSI 68-pin VHDCI connector
- Two DB-9 RS-232 serial port connectors
- One 10/100/1000 BaseT Ethernet RJ45 connector
- RTI and RTO connections (Real Time sync I/O)
- Two Xtown2 1200 MB/s or 800 MB/s (each direction) ports
- Three hot-swappable fans


PIC ASIC
Port F
Port D (XIO)
Bus 6 slot 2
Bus 6 slot 1
Bus 5 slot 2
Bus 5 slot 1
Crosstown2 XIO port
Port D (XIO)
Port A (XIO)
Port B (XIO)
PIC ASIC
Port F
Port C (XIO)
Bus 4 slot 2
Bus 4 slot 1
Bus 3 slot 2
Bus 3 slot 1
Crosstown2 XIO port
Port D (XIO)
PIC ASIC
Port F
Bus 2 slot 2
Bus 2 slot 1
Bus 1 slot 2
Bus 1 slot 1
CD-ROM
SCSI disk drives (2)
IO9 PCI card
Ethernet port
SCSI port
RTI ports
Serial ports



Notes:

The IX-brick provides the boot I/O functions for all SGI Altix 3000 series systems. It provides 12 PCI-X slots that support up to 12 PCI or PCI-X cards. The 12 slots are configured as six 2-slot buses.

Various types of PCI-X or PCI cards can be used in the IX-brick, such as SCSI, Fibre Channel, ATM, Gigabit Ethernet, etc. PCI cards can be installed in 11 of the 12 PCI slots. One PCI-X slot (the leftmost slot) is reserved for an IO9 PCI card. This card is required for the base I/O functions.

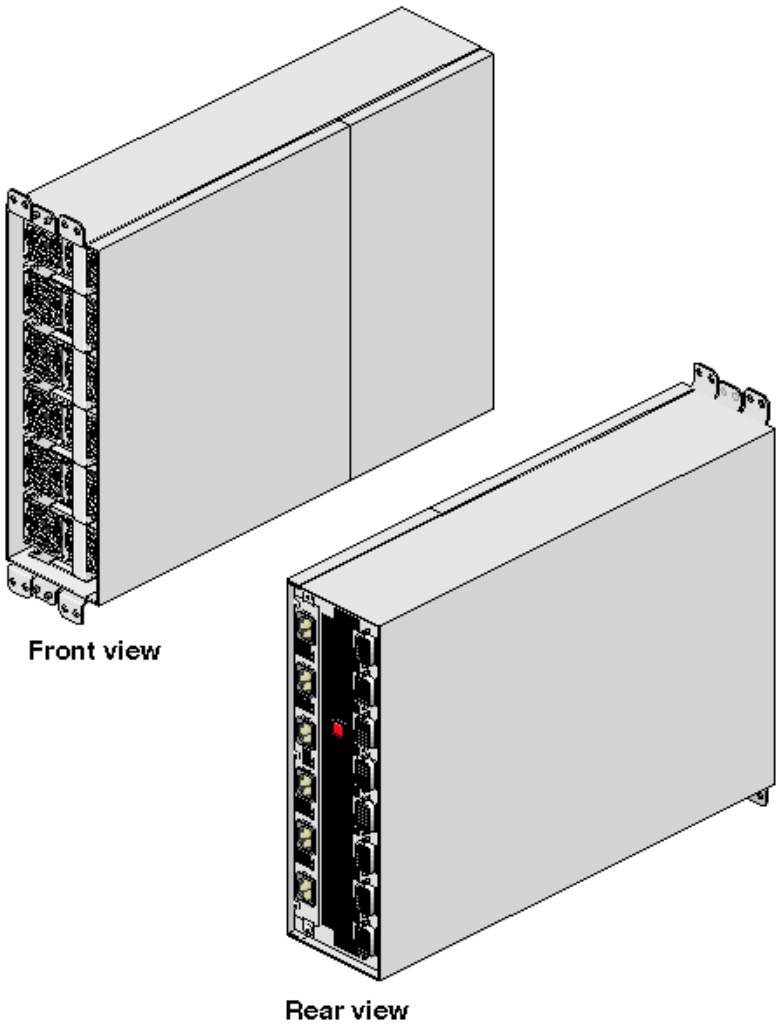
Three PIC (PCI interface chip) ASICs are key components of the IX-brick architecture. These ASICs support two 1200- or 800-MB/s Xtown2 XIO ports and six PCI-X buses (see block diagram above). Each bus has two card slots in which you can install PCI cards. (Slot 1 of bus 1, however, seats the IO9 card.)

Also important to the IX-brick architecture is the IO9 PCI card. This card contains logic that controls the DVD-ROM and internal SCSI disk drives, and it provides the following connectors:

- External VHDCI SCSI port connector.
- Internal SCSI port connector that connects to two SCSI disks.

- Gigabit Ethernet RJ45 connector.
- Two RT interrupt stereo jack connectors (one input connector labeled RTI, and one output connector labeled RTO).
- Two RS-232 DB-9 serial port connectors. (These two connectors are not located on the IO9 PCI card; instead, they are located on the right side of the IX-brick rear panel)
- An optional daughtercard can also be added to the IO9 card that adds two RS-232 DB-9 serial port connectors to the IX-brick.

3.10 The Power Bay



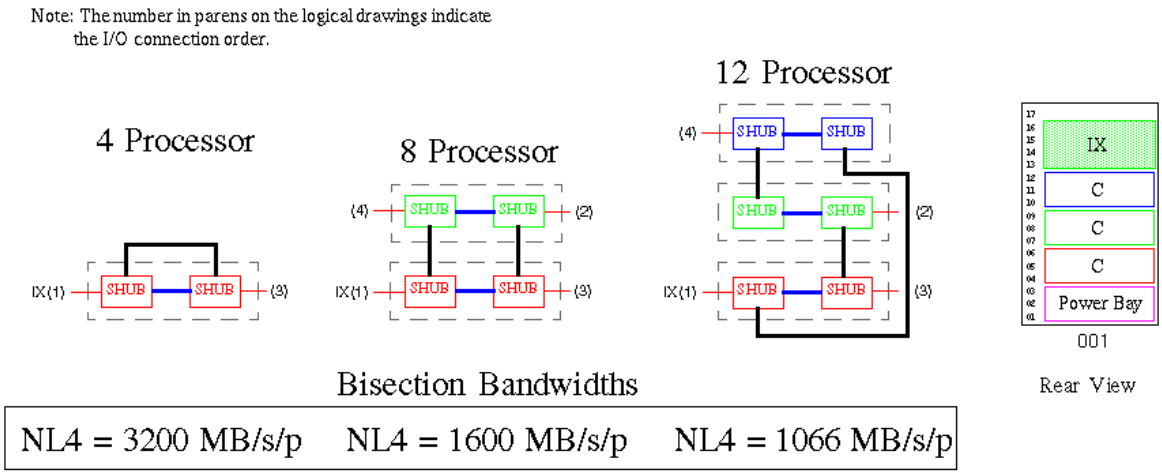
Notes:

The power bay is a 3U-high enclosure that holds a maximum of six hot-swappable distributed power supply modules (DPSs). The power bay monitors, controls, and supplies AC power to the DPSs. Although the power bay can hold a maximum of six DPSs, in this system the power bay contains three or five DPSs. The compute racks require power bays with five DPSs and the I/O racks require power bays with three DPSs.

Each DPS inputs single-phase AC voltage and outputs 950 W at 48 VDC and 42 W at 12 VDC. The outputs of the DPSs are bused together. For example, when the power bay contains three DPSs, the DPSs are bused together to provide approximately 2,760 W of 48 VDC power and 135 W of 12 VDC power.

3.11 SGI Altix 3300: Systems Up to 12 Processors

A few NUMAlink cables is all it takes to build an 8- or 12-processor system from two or three C-Bricks:

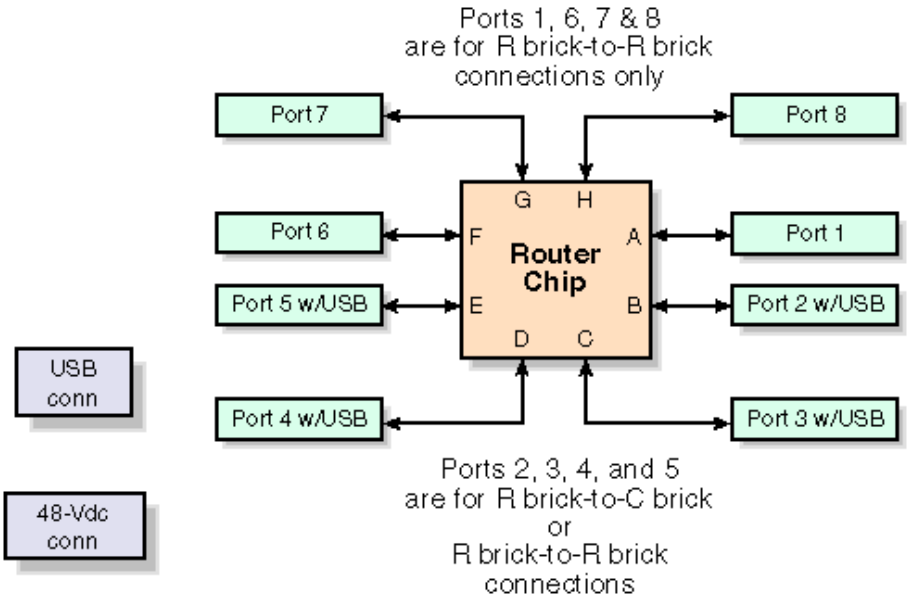


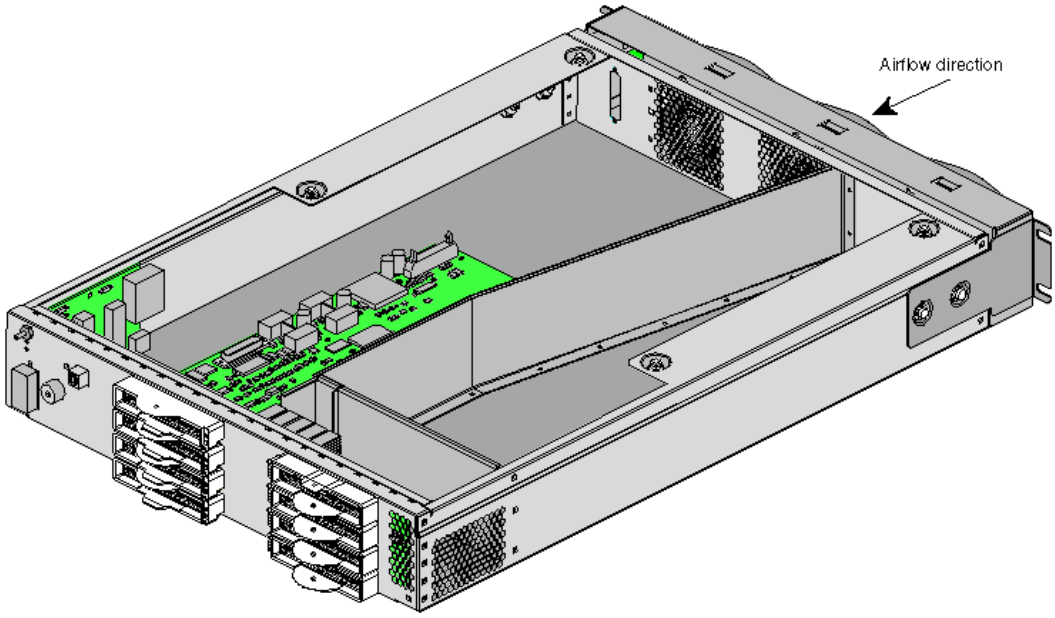
A complete system would require an IX-Brick, a Power Bay, and a short rack. Additional racks containing D-brick2s and TP900 storage modules can be added to the SGI Altix 3300 server system.

3.12 The R-Brick

The R-Brick contains:

- One 8-port router chip
- One USB port (that connects to the L2 controller)
- Eight NUMAlink3 connectors
- Hot-swappable fans





Notes:

The R-brick is an eight-port crossbar that connects any input-link channel to any of seven possible output-link channels. It contains a router ASIC that is mounted on a PCB with its associated power circuitry, L1 controller, and a USB hub. The hub fans out USB signals from the L2 controller to the L1 controller inside the R-brick and to the four nodes (C bricks) that may be connected to the router.

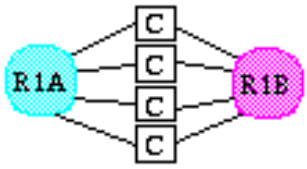
The R-brick has a total of eight 100-pin link connectors located on its rear panel. Four of these connect to C-bricks and carry USB signals as well as link signals. The others are only for connection to other routers and do not carry USB signals. Metarouters and repeat routers use all eight ports to connect to other R-bricks.

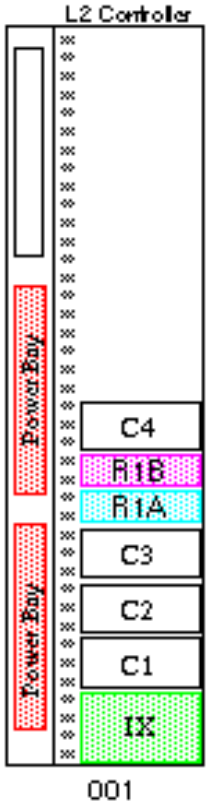
When an R-brick-to-R-brick connection is made through ports that carry USB signals, the USB signals are ignored. USB signals to the C-bricks are distributed over the network cables. Because an R-brick can have a maximum of four C-bricks attached to it, only four of the R-brick's 100-pin network connectors have USB signals routed to them. Ports 2, 3, 4, and 5 carry USB signals. Therefore, a C-brick must connect to an R-brick via port 2, 3, 4, or 5.

Each R-brick has a dedicated USB connection to the L2 controller through a 4-pin USB connector on its rear panel. Therefore, it is not necessary for an R-brick to distribute USB signals to other R bricks. R-brick-to-R-brick network connections are normally made through the four port connectors that do not carry USB signals; however, they are not restricted to these four ports.

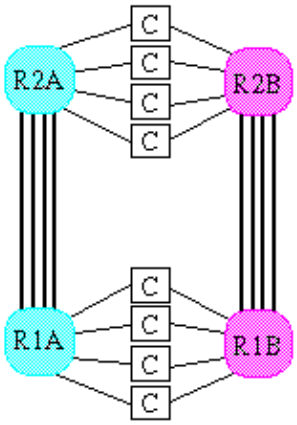
3.13 Systems Up to 16 Processors

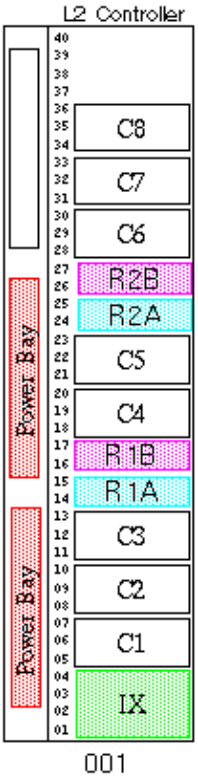
Up to four C-bricks can connect to each of two R-bricks. The remaining ports on the R-bricks connect to other routers in larger systems, using a dual-plane fat-tree configuration.



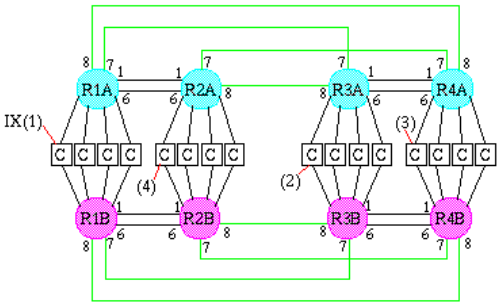


3.14 32-processor System



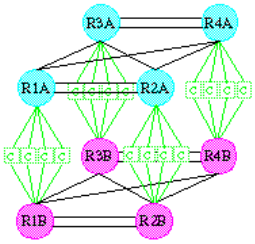


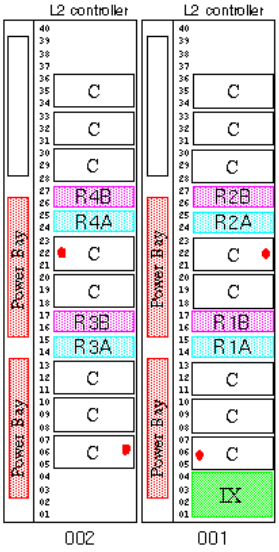
3.15 64-processor System



Note: The number in parens on the logical drawings indicate the I/O connection order for a non-partitioned system.

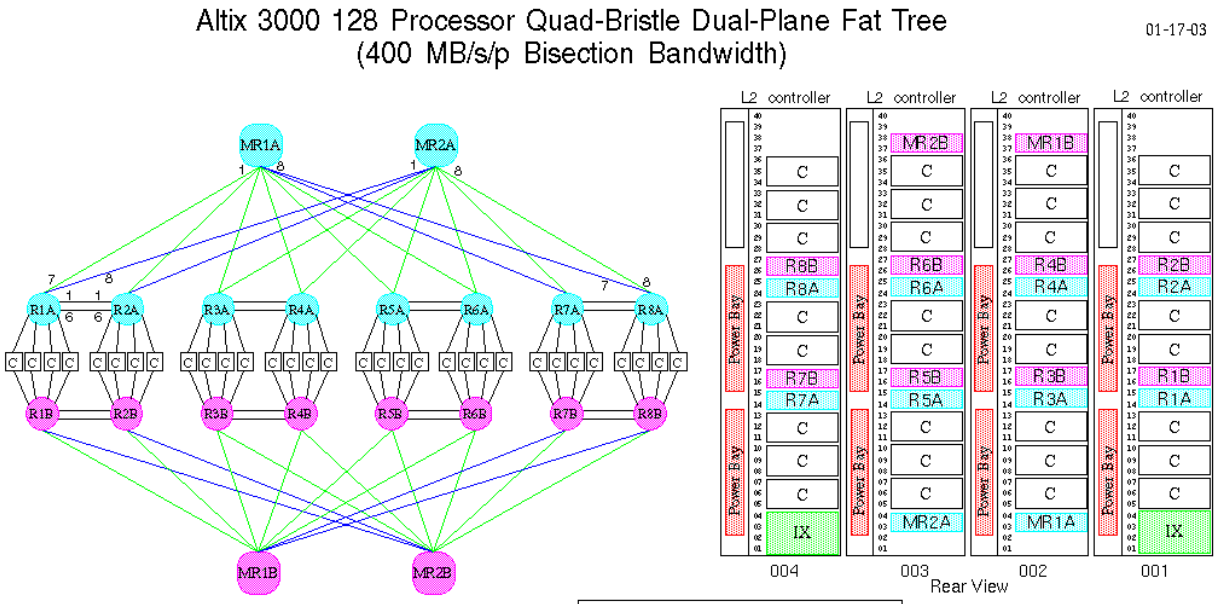
Alternate drawing





Rear View

3.16 128-processor System



Maximum Number of Hops per System	
16 Proc.	3 Hops
32 Proc.	4 Hops
64 Proc.	5 Hops
128 Proc.	5 Hops

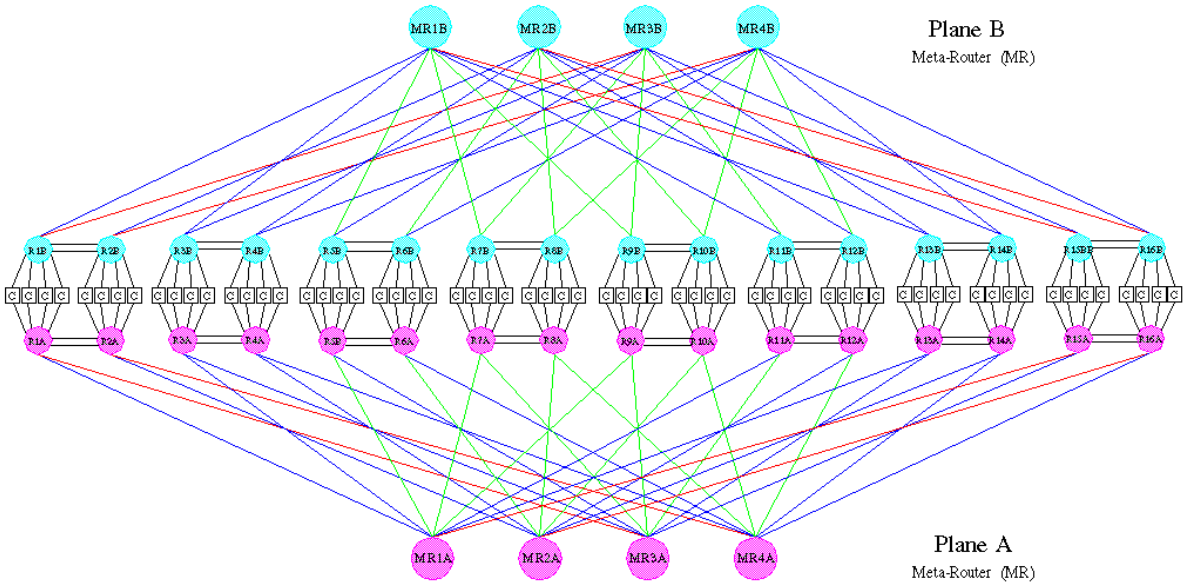
Cable Color Code Chart

Color	Length	Quantity
Black	1 Meter	80
Green	2 Meter	24
Blue	3 Meter	8

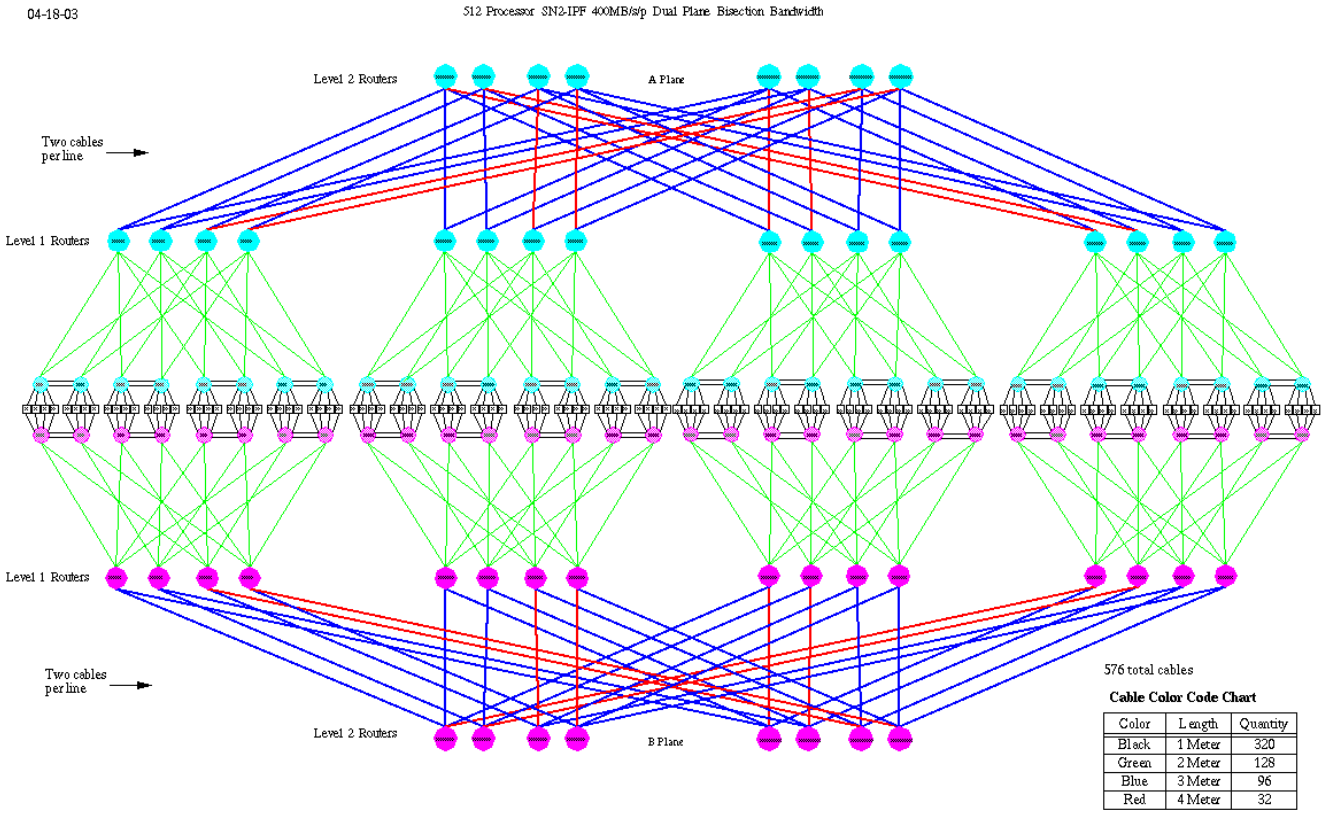
A "Hop" is defined as a NUMAlink connection between two devices. The devices can be a SHUB or a router. The maximum number of hops is the total number of NUMAlink connections between a source SHUB and destination SHUB with in a specified group size.

Note: This is laid out with SN1 (NL3) routers.

3.17 256-processor System



3.18 512-processor System



Color	Length	Quantity
Black	1 Meter	320
Green	2 Meter	128
Blue	3 Meter	96
Red	4 Meter	32

3.19 SGI Altix 3700 Memory Limits and Bandwidths

No. of SHubs or nodes	Maximum No. of processors	NUMAlink3 Bandwidth MB/s/CPU	NUMAlink4 Bandwidth MB/s/CPU	Maximum Memory 1 GB DIMMs	No. of Routers	Maximum No. of Router hops
8	16	800	1600	128 GB	2	3
16	32	800	1600	256 GB	4	4
32	64	400	800	512 GB	8	4
64	128	400	800	1 TB	20	5
128	256	400	800	2 TB	40	5
256	512	400	800	4 TB	112	7
512	1024	400	800	8 TB	288	10
1024	2048	400	800	16 TB	576	10

Notes:

The SGI Altix 3000 SHub's interconnect ports operate at the speed of the 4th generation of the NUMAflex technology, NUMAlink 4. NUMAlink 4 provides double the bandwidth of NUMAlink 3 while maintaining compatibility with NUMAlink 3 physical connections. This increase in bandwidth is achieved by employing advanced bidirectional signaling technology.

As shown in the previous pages' diagrams, the NUMAflex network for the SGI Altix 3000 is configured in a fat tree topology, which enables the system performance to sclae well by providing increases in bisection bandwidth as the system increases in size. Initial SGI Altix systems use the NUMAlink 3 router brick, so that when the NUMAlink 4 router brick becomes available the bisection bandwidth will be doubled, allowing the system's capabilities to grow along with the demands of new generations of Itanium 2 family processors.

In the above table maximum memory sizes are given assuming 1 GB DIMMs are used; other possibilities include 512 MB and 2 GB DIMMs. Since memory DIMMs can be added to the SGI Altix 3000 using M-bricks, the memory can be scaled independently of the processors. Hence it is entirely possible to build a system with 16 processors and 4 TB of shared cache-coherent memory.

The physical address space is split into a memory address of 36 bits (architectural limit of 64GB per SHub), and compute node address of 10 bits (1K nodes, or 2K processors). While the initial system is capable of coherently sharing cache lines among up to 512 processors, it is possible to build a single NUMAflex network with globally upgradeable memory of up to 2,048 processors. Communications among the four 512-processor sharing domains on the NUMAflex network use coherent I/O semantics for moving data between the sharing domains, while still utilizing the low-latency, high-bandwidth characteristics of NUMAlink. Future enhancements may increase the size of the coherence limit to thousands of processors.

Previous generations of NUMAflex systems used proprietary memory DIMMs to provide data storage and the additional directory storage for tracking cache coherence within the system. The Altix 3000 family breaks away from this limitation and embraces industry-standard DIMM technology, providing the economic benefits of this standard technology without sacrificing performance. The memory subsystem of Altix 3000 uses PC-style double data rate (DDR) SDRAM DIMMs. Each SHub ASIC in a C-brick supports four DDR buses. Each DDR bus may contain up to four DIMMs (each DIMM is 72 bits wide, 64 bits of data and 8 bits of ECC). The four memory buses are independent and can operate simultaneously to provide up to 12.8GB per second of memory bandwidth.

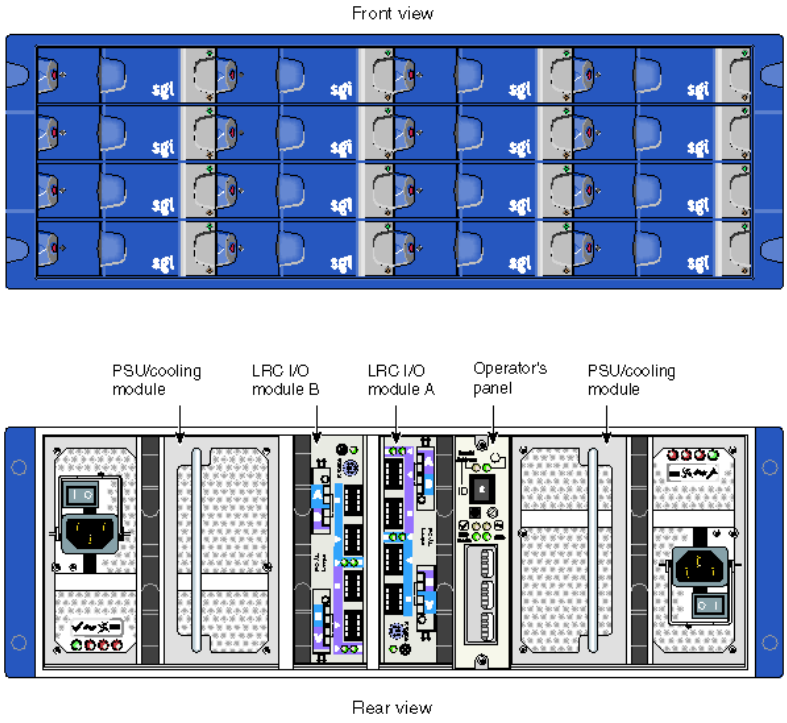
The system supports the use of PC2100 (DDR 266 MHz), PC2700 (DDR 320 MHz), and PC3200 (DDR 400 MHz) DIMMs. The aggregate memory bandwidths using these DIMMs are 8.5GB per second and 10.2GB per second respectively. Using 256Mb DRAM tech-nology yields a per-DIMM capacity of 512MB. This gives each

node a base capacity of 8GB, 4GB per processor. One-gigabyte DIMM tech-nology will provide a capacity of 32GB per C-brick or 8GB per processor. Different DIMM sizes may be mixed in a C-brick but must be added in sets of eight DIMMs at a time. The system is designed to accommodate 2GB DIMMs when they become commercially available.

Each SHub ASIC contains a directory cache for the most recent cache-coherency state information. This allows for efficient utilization of the memory subsystem for performing data operations, by minimizing the amount of memory bandwidth that is needed to look up cache-coherency state information. When the system is booted, it sets aside approximately 3% of the memory space to store the cache-coherency directory information (in comparison, typical memory structures set aside 12% of memory space to store error-detection and correction codes) for the system. This directory space is used to store directory information that is not being actively used in the directory cache. The directory information is stored in parallel with the cache-line data, but on a different DIMM bus. So if the directory state for a memory reference is not currently available in the on-chip directory cache, the directory information can be read from memory at the same time as the data (which resides on a different DIMM bus). This optimized storage scheme ensures that the system can achieve maximum delivered data bandwidth by minimizing bus usage conflicts. While the local processor bus has a peak bandwidth of 6.4GB per second, the local memory subsystem has enough bandwidth to fully saturate the local processor demands while leaving available bandwidth to service remote processor and I/O memory requests.

3.20 D-Brick2

The optional D-brick2 module is a high performance, large-scale non-RAID storage system for the SGI rackmounted systems. Each enclosure contains a minimum of 2 and maximum of 16 disk drives, and the component modules that handle I/O, power and cooling, and operations. Optional RAID storage systems are also available.

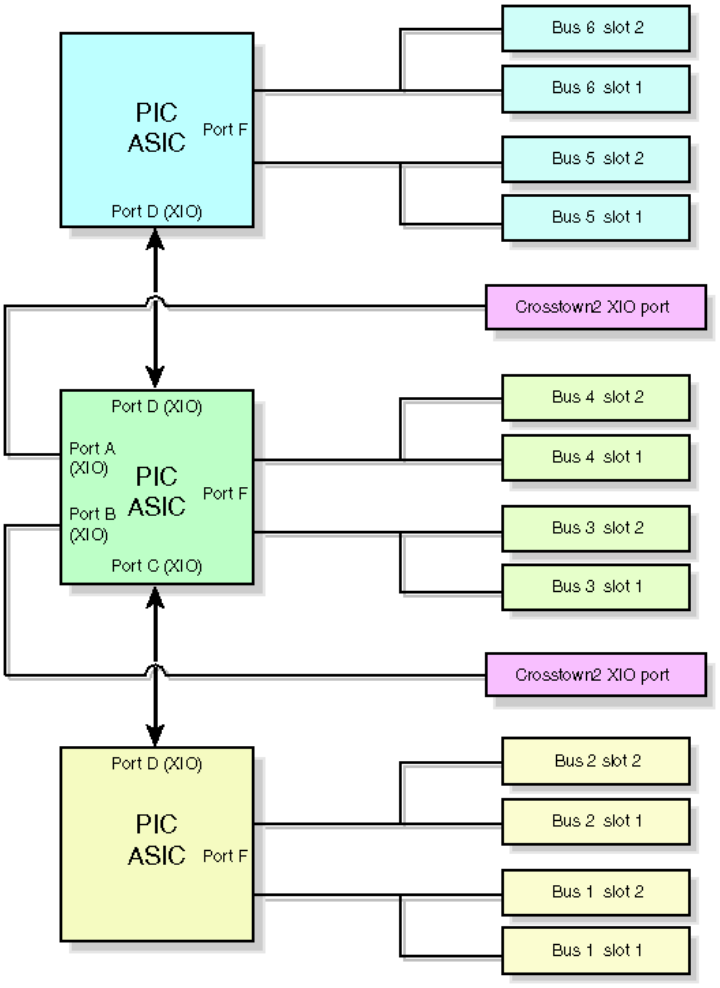


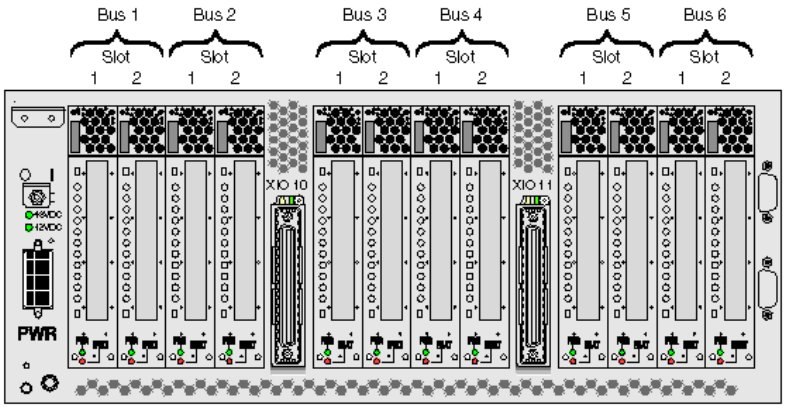
Notes:

The D-brick has the following features:

- Maximum configuration of up to 96 drives (six D-brick2 units)
- 1×16 (more storage) and 2×8 (more bandwidth) disk topologies in each brick
- Dual power feeds with dual power supplies
- Redundant cooling
- Non-disruptive component replacement
- Enclosure services interface (ESI) for SCSI enclosure services (SES)

3.21 PX-Brick


Bus 6 slot 2
Bus 6 slot 1
PIC ASIC
Port F
Port D (XIO)
Bus 5 slot 2
Bus 5 slot 1
Crosstown2 XIO port
Port D (XIO)
Port A (XIO)
PIC ASIC
Port F
Port B (XIO)
Port C (XIO)
Bus 4 slot 2
Bus 4 slot 1
Bus 3 slot 2
Bus 3 slot 1
Crosstown2 XIO port
Port D (XIO)
PIC ASIC
Port F
Bus 2 slot 2
Bus 2 slot 1
Bus 1 slot 2
Bus 1 slot 1


Bus 1
Bus 2
Bus 3
Bus 4
Bus 5
Bus 6
Slot
1 2
XIO 10
XIO 11
PWR

Notes:

The 4U-high PX-brick provides 12 card slots to support up to 12 PCI or PCI-X cards. The 12 slots are configured as six 2-slot buses. Three PIC (PCI interface chip) ASICs are key components of the PX-brick architecture. The PIC ASICs support the following

- Two 1200- or 800-MB/s Xtown2 XIO ports. (You can select the MB/s setting with the L1 controller command XIO. For more information, see the SGI L1 and L2 Controller Software User's Guide.)
- Six PCI/PCI-X buses. Each bus has two card slots in which you can install PCI or PCI-X cards.

SGI supports various PCI and PCI-X cards. These cards can be purchased from SGI or another manufacturer. A list of supported PCI cards can be obtained from SGI sales representatives.

3.22 Intel Itanium 2 Processor

- EPIC (Explicitly Parallel Instruction Computing) architecture
  - 128 general (integer) registers; up to 96 rotating
  - 128 floating-point registers; up to 96 rotating
  - 64 1-bit predicate registers; up to 48 rotating
  - 8 branch registers
  - 128 application registers (e.g., loop or epilog counters)
  - Performance Monitor Unit (PMU)
  - Advance Load Address Table (ALAT)
  - 3 predicated instructions in a single 128-bit bundle
  - 2 bundles per clock cycle
  - 6 integer units
  - 2 loads and 2 stores per clock
  - 11 issue ports

3.23 Intel Itanium 2 Processor (continued)

- Instruction Level Parallelism support
  - Speculation (to hide memory latency)
  - Predication (to remove branches)
  - Software pipelining
  - Branch prediction
  - Prefetch instructions to hide memory latency
- Special instructions (`popcnt`, multimedia, etc.)
- IA32 mode for x86 compatibility

3.24 Intel Itanium 2 Processor (continued)

- Three-level cache memory hierarchy
  - 16 kB L1 instruction cache (64-byte lines, 4-way associative)
  - 16 kB L1 data cache (64-byte lines, 4-way associative, write-through)
  - 256 kB unified L2 cache (128-byte lines, 8-way associative, write-back)
  - “McKinley”: 1.5 MB or 3 MB unified L3 cache (128-byte lines, 6/12-way associative, 32 GB/s)
  - “Madison”: 3 MB or 6 MB unified L3 cache (128-byte lines, 6/12-way associative, 48 GB/s)
- Front Side Bus: 400 MHz, 128-bit wide system bus, 6.4 GB/s bandwidth
- 50-bit physical addressing, 64-bit virtual addressing
- Maximum page size of 4 GB

3.25 Itanium 2 Instruction Bundle

- 1 instruction coded on 41 bits
- 3 instructions grouped into 128-bit bundle
- Bundle type is specified through 5-bit template

```
{ .mfi                                // template (mem-fp-int)
   (p16) ldfd f39=[r2],16             // load fp, post-increment
   (p19) fnma.d.s0 f49=f42,f6,f45 // multiply-add
   (p16) adds r32=16,r33              // integer add immediate
};
{ .mib                                // template (mem-fp-br)
   (p16) ldfd f42=[ r33]              // load fp, post-increment
   (p16) adds r40=8,r33
       br.ctop.dptk.few .BB13_ ;; // counted loop branch
};
```

3.26 Itanium 2 Branch Optimization

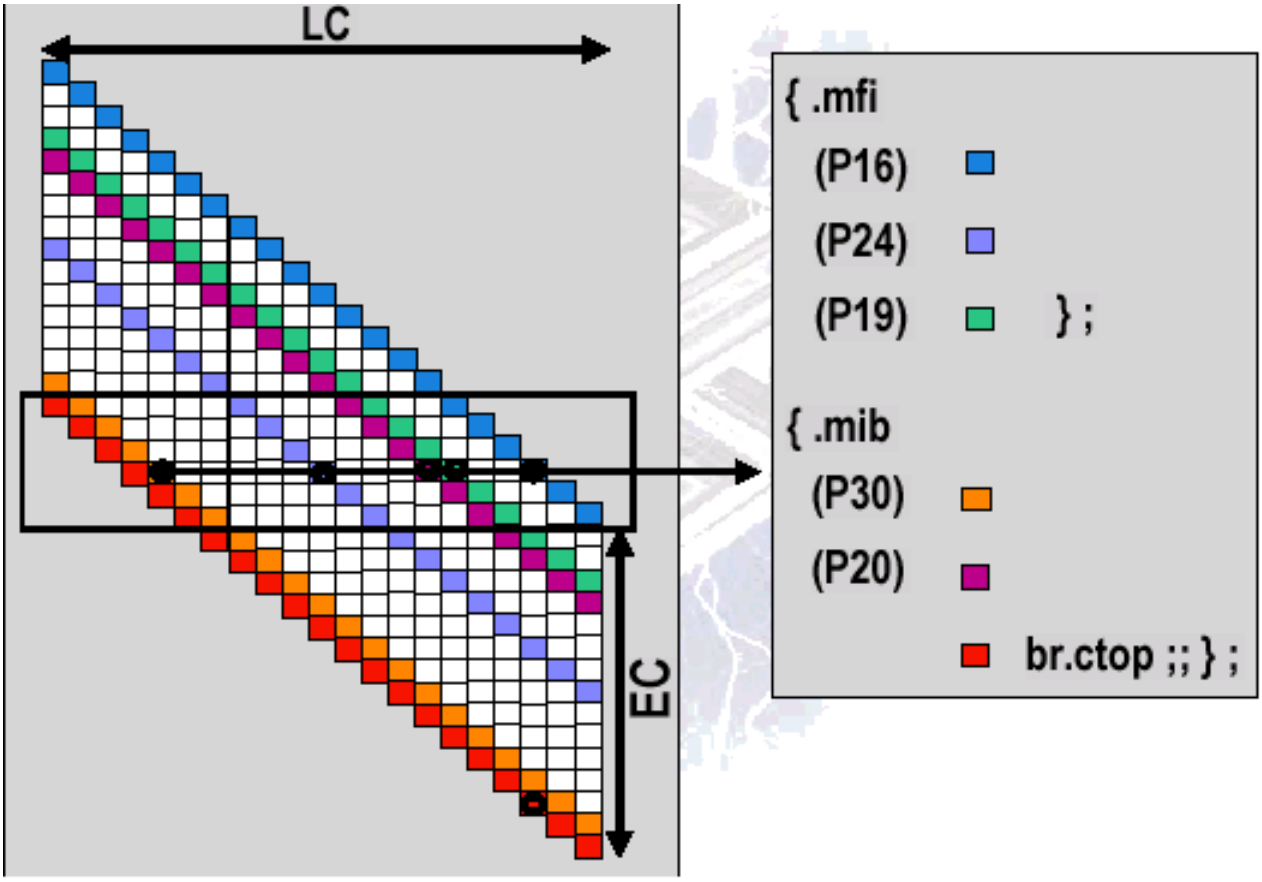
- Predication allows removal of (small) branches

```
if ( i == j) {      cmp.eq p1,p2=r32,r33 ;;   // cycle 0
   k += l;          (p1) add r1=r1,r3         // cycle 1
   x = y + a * b;   (p1) fma.d f31=f3,f4,f2   // cycle 1
} else {
   k = m - 3;       (p2) sub r1=3,r4          // cycle 1
   y = *p_fp++;     (p2) ldfd f31=[r34],8     // cycle 1
}
```

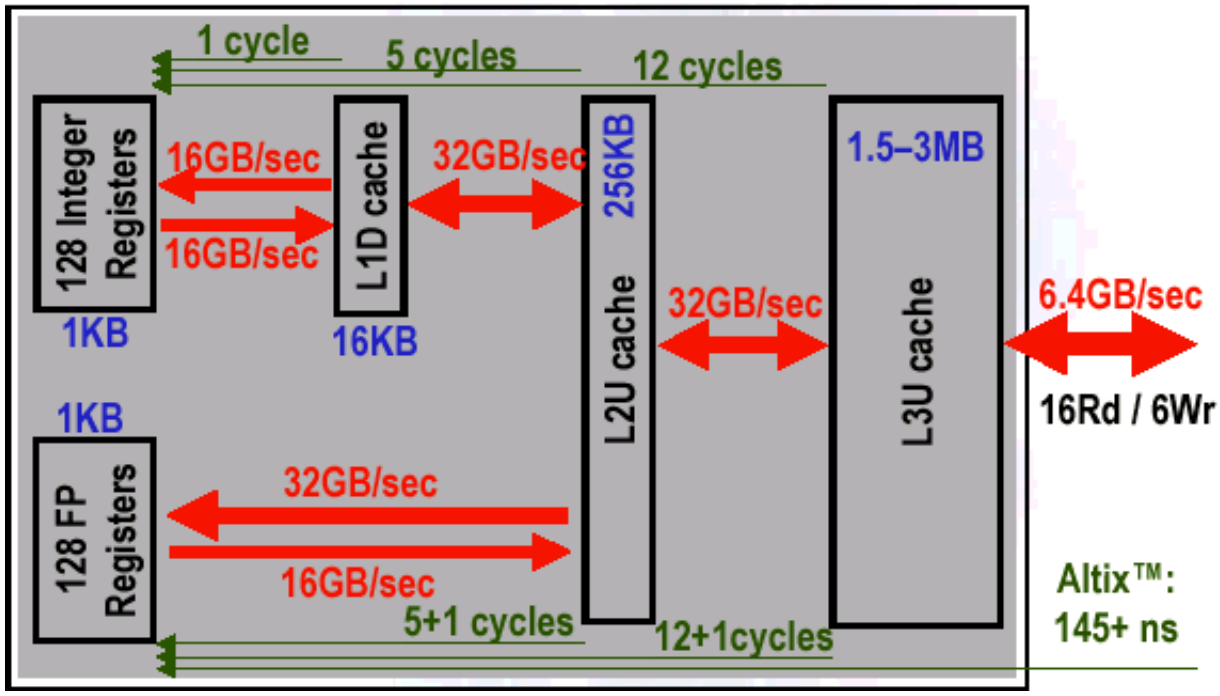
3.27 Itanium 2 Loop Optimization

- Counted loops are optimized with hardware support
  - Loop counter
  - Epilog counter
  - Predication registers for each instruction
  - Rotation of registers

3.28 Itanium Software Pipelining



3.29 Itanium 2 Data Flow (900MHz/1 GHz)



3.30 Memory Access Latencies on the SGI Altix

- Local latency: 145ns
- Same C-brick, other node: 290ns
- NUMAlink 3 Router traversal: 50ns
- 1 m NUMAlink cable: 10ns

3.31 Itanium 2 Translation Lookaside Buffer (TLB)

- Page sizes: 4, 8,16, 64, 256 kB; 1, 4, 16, 64, 256 MB; 1, 4 GB
- Addresses: 50-bit physical, 64-bit virtual
- 2 levels of TLB for data and instructions
- L1 DTLB: 32 (4K) entries, fully associative
- L2 DTLB: 128 entries, fully associative, (up to 64 TRs)
- Integer store and FP accesses have no penalty for L1 TLB miss
- L1 DTLB miss $\rightarrow$ L1D miss ; if L2 DTLB hit: +4 cycle penalty
- L2 DTLB miss $\rightarrow$Hardware Page Walker (HPW)
- HPW hit in L2 $\rightarrow$ 25 cycle penalty
- HPW miss in L2, hit in L3 $\rightarrow$ 31 cycle penalty
- HPW miss in L2 and in L3 $\rightarrow$ OS trap

3.32 TLB Miss Cost

Linux® kernel currently supports up to 64 kB pages: what is the performance impact?

- TLB misses are handled in hardware in most cases
- TLB miss costs are low compared to memory access:
  - Streaming data, worst case: 20 cycles
  - Random hops: 30 cycles
- TLB misses should rarely be a problem on Itanium® 2

Module 4

SGI Altix NUMAtools

4.1 Module Objectives

After completing this module, you should be able to

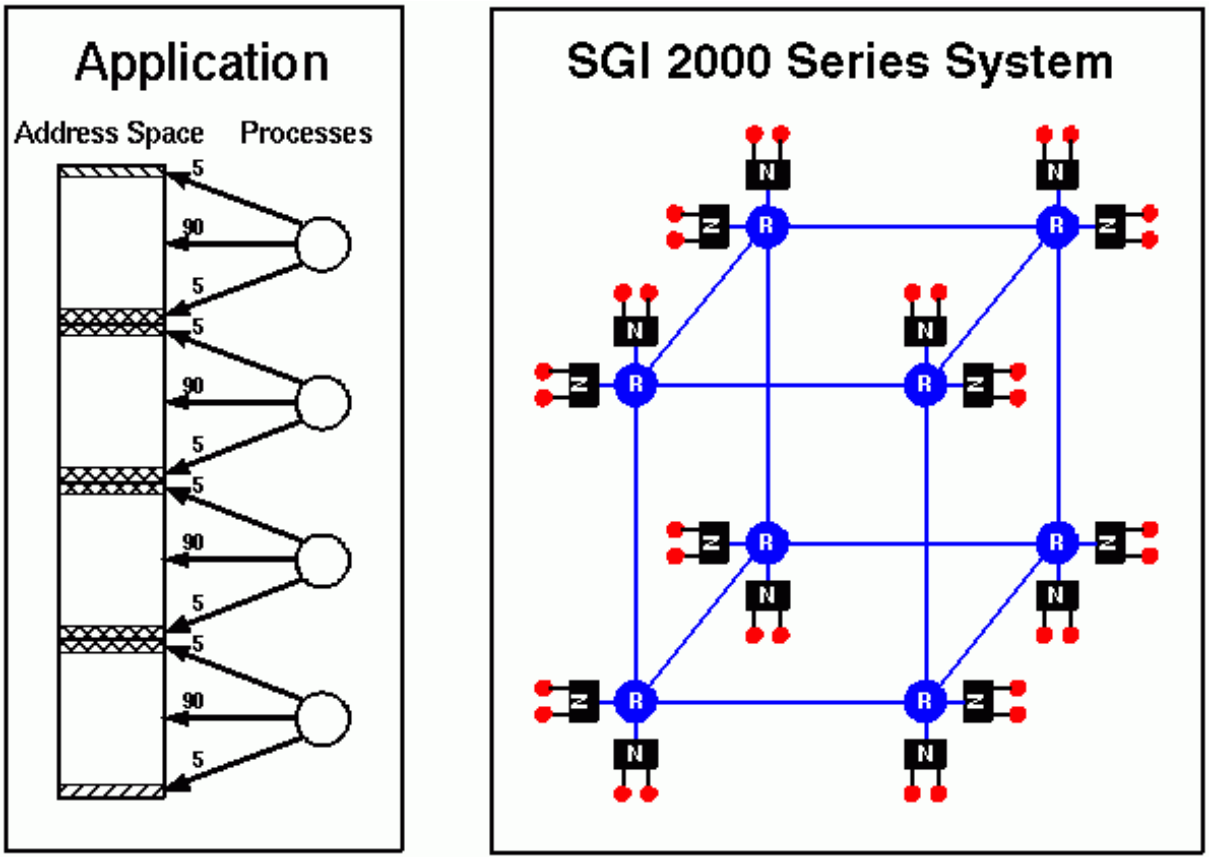
- Understand the issues of memory locality and nonuniform memory access
- Use the `dlook` and `dplace` tools to improve performance of processes running on an SGI Altix system

4.2 It’s Just Shared Memory

Physically, it is distributed memory

- Scalable network replaces bus architecture
- NUMAtools enhancements to Linux work to minimize NUMA effects

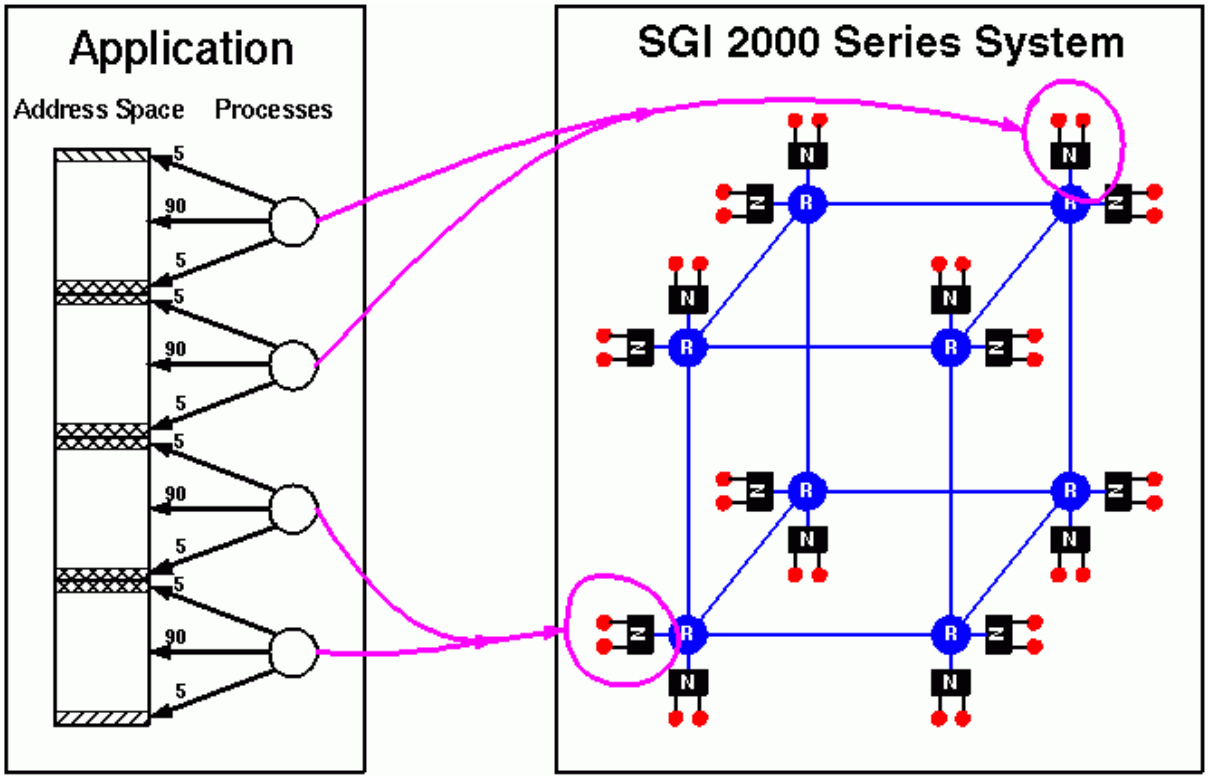
4.3 Simple Memory Access Pattern



Notes:

Here is a shared memory application that we want to run on four processors. This particular application exhibits a relatively simple memory access pattern, namely, 90% of each process's cache misses are from memory access to an almost unshared section of memory, 5% to a section of memory shared with another process, and the remaining 5% to a section of memory shared with a third process. We now consider how the four processes might be mapped onto processors of an SGI Origin 2000 System.

4.4 Non-Optimal Placement

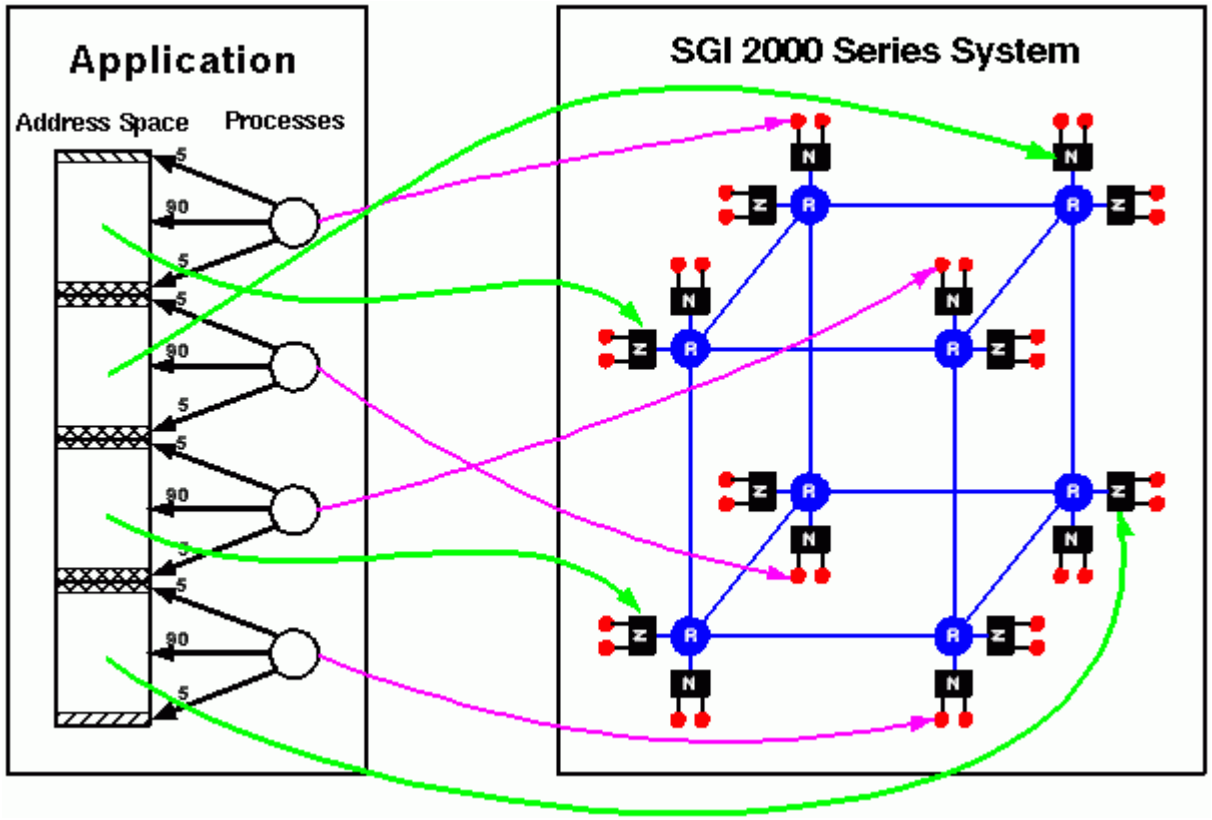


Notes:

Neglecting attention to memory locality can result in the situation shown above: a couple of processes running in one corner of the machine, and the other processes running in an opposite corner. This situation results in the second and third processes incurring longer-than-optimal memory latencies to the shared section of memory.

Actually, this result is not that bad. Because the SGI Origin 2000 hardware has been designed to keep the variation in memory latencies relatively small, and because accesses to the shared section of memory only account for 5% of two of the process's cache misses, the non-optimal placement has only a small effect on the performance of the program.

4.5 Random Placement

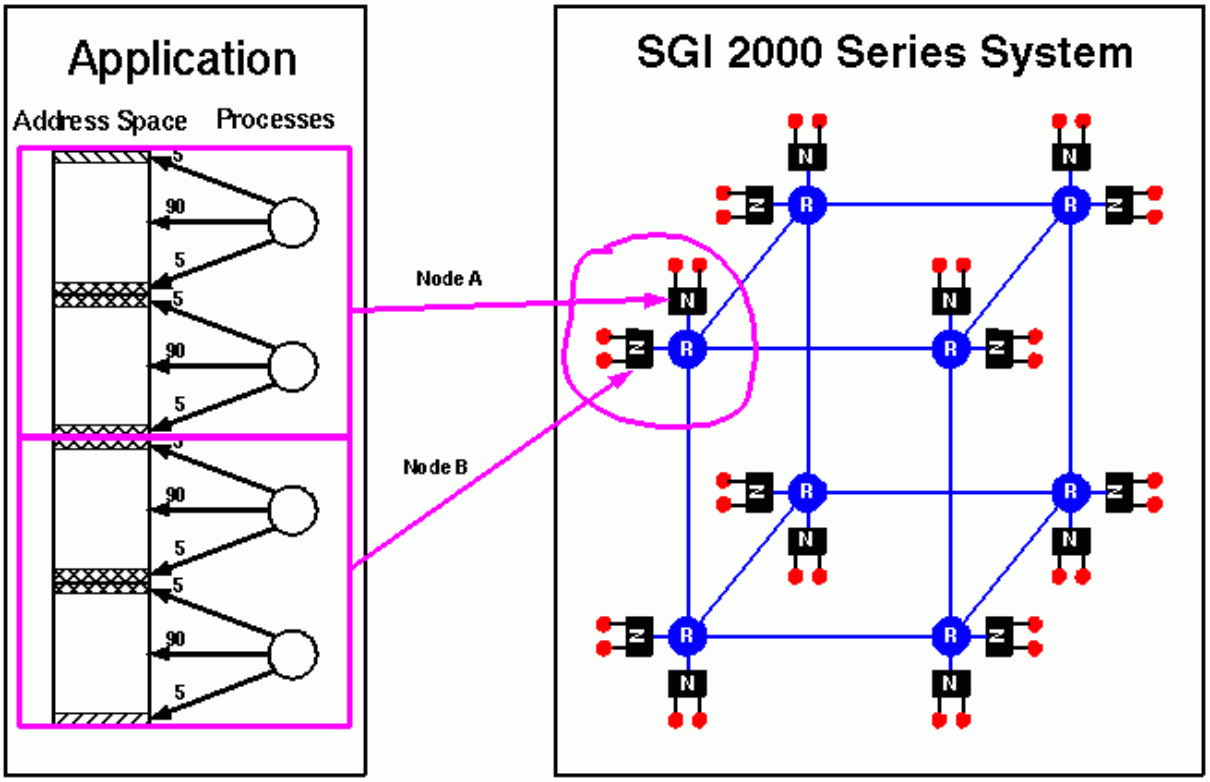


Notes:

There are situations, however, in which performance can be affected significantly. If absolutely no attention is paid to memory locality, the processes and memory might result in the situation shown above: each process runs on a different and distant node, and the sections of memory they use have been allocated from yet a different set of distant nodes. In this case, even the accesses to unshared sections of memory—which account for 90% of each process's cache miss—are nonlocal, thus increasing the costs of accessing memory. In addition, program performance can vary from run to run, depending on how close each process ends up to its most-accessed memory.

The standard Linux kernel scheduling algorithm tends to move processes around on a multiprocessor system, with the ill effect that data may have been mapped to physical memory on one brick, while a process or set of processes accessing that data run on entirely different bricks.

4.6 Ideal Placement



Notes:

The NUMAtool utility `dplace` is used to avoid such situations in Linux. Ideally, the processes and memory used by this application are placed in the machine as shown here.

4.7 Data Placement Policy

- First-touch
  - Default policy
  - First processor to touch a page of memory causes it to be allocated from its local memory
  - Works well for fully parallelized programs, but serial initializations cause non-local accesses and bottlenecks

4.8 Running on Specific CPUs — `runon(1)` and `cpuset(1)`

- The `runon` command executes a command on a CPU or list of CPUs:
  - **`runon 1 3-5 ./a.out`**
  - find idle CPUs using `top`
  - `runon` restricts the process and its children to run on the set of listed CPUs, but does not prevent them from moving around within it.
- `cpuset(1)` allows you to run your programs on a restricted subset of processors and associated memory, called a cpumemset
  - It requires the prior creation of a cpumemset to run the program in
  - cpumemsets may be created:
    - manually by root
    - automatically by batch schedulers such as Platform's LSF or Altair Engineering's PBSPro
  - **`cpuset -q my_cpuset -A a.out`**
  - determine existing cpumemsets with `cpuset -Q`
- As usual, read the `man` pages!

4.9 Running on Specific CPUs — `dplace(1)`

- `dplace` is a tool for controlling placement of processes onto CPUs

  `dplace [-c` *cpu_numbers*`] [-s` *skip_count*`] [-n` *process_name*`] [-x` *skip_mask*`]`
  command `[`*command-args*`]`

- To determine which processes are currently running that were bound by `dplace`

  `dplace -q`

Notes:

By default, memory is allocated to a process on the node that the process is executing on. If a process moves from node to node while it running, a higher percentage of memory references will be to remote nodes. Remote accesses typically have higher access times. Process performance may suffer.

`dplace` is used to bind a related set of processes to specific CPUs or nodes to prevent process migrations. In some cases, this will improve performance since a higher percentage of memory accesses will be to the local node.

Processes always execute within a cpumemset. The cpumemset specifies the CPUs that are available for a process to execute on. By default, processes usually execute in a cpumemset that contains all the CPUs in the system.

`dplace` invokes a kernel hook (PAGG—process aggregates) to create a placement container consisting of all or a subset of the CPUs of the cpumemset. The `dplace` process is placed in this container and (by default) is bound to the first CPU of the cpumemset associated with the container. Then `dplace` "execs" the *command*.

`man dplace` gives examples of `dplace` usage with various types of processes, including MPI and OpenMP applications.

4.10 Determining Data Access Patterns — `dlook(1)`

- `dlook(1)` allows you to display the memory map and CPU usage for a specified process

  `dlook [-a] [-c] [-h] [-l] [-o` *outfile*`] [-s` *secs*`]` *command* `[`*command-args*`]`
  `dlook [-a] [-c] [-h] [-l] [-o` *outfile*`] [-s` *secs*`]` *pid*

- For each page in the virtual address space of the process, `dlook(1)` prints the following information:
  - The object that owns the page, such as a file, SysV shared memory, a device driver, etc.
  - The type of page, such as random access memory (RAM), FETCHOP, IOSPACE, etc.
  - For RAM pages, the following are also listed:
    * memory attributes (SHARED, DIRTY, etc.)
    * node that the page is located on
    * physical address of page, if option `-a` is specified

- With option `-c`, `dlook(1)` also prints the amount of elapsed CPU time that the process has executed on each physical CPU in the system.

Notes:

Two forms of the `dlook(1)` command are provided. In one form, `dlook` prints information about an existing process that is identified by a process ID (PID). To use this form of the command, you must be the owner of the process or be running with root privilege. In the other form, you use `dlook` on a command you are launching and thus are the owner.

Besides the -a and -c options explained above, the following options are available:

- `-h` Explicitly list holes in the address space.
- `-l` Show libraries.
- `-o` Output file name. If not specified, output is written to `stdout`.
- `-s` Specifies a sample interval in seconds. Information about the process is displayed every *secs* of CPU usage by the process.

4.11 Recommendations for Achieving Good Performance

- Remember: It's just shared memory
  - CPUmemsets, `dplace` and `dlook` are there to help them run well
- Memory is allocated and distributed in pages
- First tune single-processor performance
- Cache friendly programs will run well on the NUMA architecture
- Memory-intensive, cache unfriendly programs
  - If scaling is less than expected, data placement may be a problem
  - Make sure the data are uniformly distributed
- When developing new code use first-touch and program with it in mind

Module 5

Virtual Memory Management

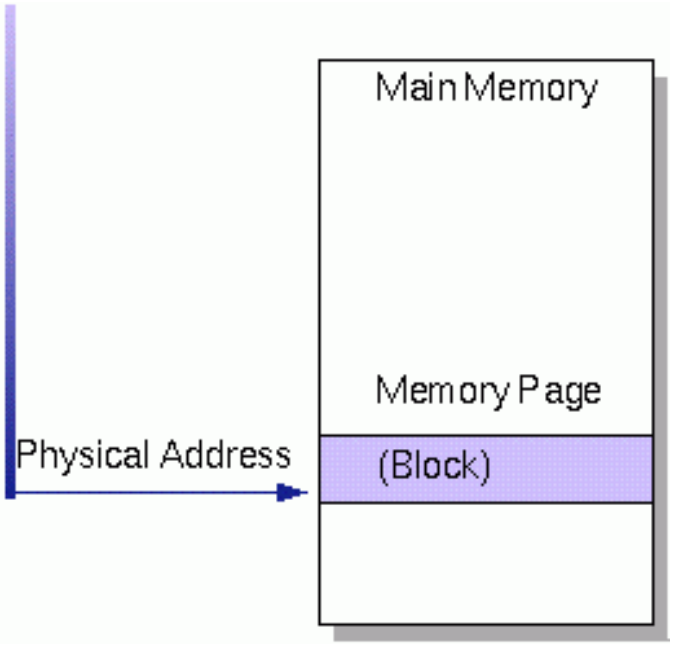
5.1 Module Objectives

After completing this module, you will be able to

- Understand the basics of virtual memory

5.2 Pages

- A page is the smallest unit of system memory allocation
- Pages are added to a process when either a
  - *Validity* fault occurs
  - `malloc` allocation request occurs
- Number of pages in a process can be determined from `ps(1)`or `top(1)`



Notes:

Virtual memory (VM), or virtual addressing, is used to divide the system's relatively small amount of physical memory among the potentially larger amount of logical processes in a program. It does this by dividing physical memory into pages, and then allocating pages to processes as the pages are needed.

VM provides each process with 16-kB pages and allocates more as the process needs it 16-kB increments. Pages in main memory are indexed with a physical address. This process is shown in the above slide.

The number of pages a process has is reported from the BSD form of `ps` (`ps -l`) under the column RSS.

5.3 Process Size

- Process size is measured in pages
- Two sizes associated with every process
  - Total size
  - Resident set size (RSS)
- Use `ps(1)` or `top(1)` to determine process size
- Use `free(1)` for system memory usage
- Shared memory and dynamically shared objects (`.so` libraries) are counted several times

5.4 Process Memory

- Addresses within codes are virtual
- Virtual addresses can map anywhere in physical memory
- Virtual addresses are mapped to physical memory address through the Translation Lookaside Buffer (TLB)
- There is a limited number of TLB entries

Notes:

A process is assigned physical memory only when it is using it. Parts of the address space that are not being used can be swapped out, or never assigned.

Virtual addresses must be looked up in the TLB to determine where they are in the physical memory. There are a limited number of TLB entries on the chip. If an address cannot be found in the TLB, interrupt will occur and the OS will update the TLB using an LRU strategy.

Virtual addresses are consecutive within physical memory only within a page. This allows the OS great freedom in assigning physical memory. This can work to the advantage of a multiple CPU shared memory program. Consecutive memory addresses within a program can be spread out across the machine.

5.5 Running Out of Physical Memory

When physical memory becomes scarce, the OS can:

- Reclaim memory by paging
- Terminate processes manually
- Terminate processes automatically

Notes:

The CPU can only reference data and execute code if the data or code are in main memory (RAM). Because the CPU executes multiple processes, there may not be enough memory for all the processes. If you have very large programs, they may require more memory than is physically present in the system. So, processes are brought into memory in pages; if there is not enough memory, the operating system frees memory by writing pages temporarily to a secondary memory area, the swap area, on a disk.

5.6 Paging

- Linux is a *demand paging* operating system
- Validity fault
  - maps pages into physical memory when first referenced
  - brings pages back into memory if swapped out
- Least recently used paging algorithm

5.7 Swap Space

- Used for temporarily saving parts of programs when there is not enough physical memory
- Swap space may be
  - On the system drive
  - On an option drive
  - A swap file on a filesystem
- To avoid swapping, do not oversubscribe memory

Notes:

Swap space is disk space allocated to be used as medium-term memory for the operating system kernel. Lack of swap space limits the number and size of applications that may run simultaneously on your system and can interfere with system performance.

Using swap space can help kill program performance. `top(1)` can be used to monitor swap space usage.

Module 6

Linux System Utilities

6.1 Module Objectives

After completing this module, you will be able to

- Identify which system monitoring tool is most applicable to a particular situation
- Determine hardware characteristics of the SGI Altix Series
- Determine how heavily various system components are being used (CPU, memory, I/O devices)

6.2 System Monitoring Concepts

- Use monitoring tools to better understand your machine’s limits and usage
- Observe both overall system performance and single-program execution characteristics

6.3 Determining Hardware Inventory

- `hinv(1)` displays the contents of the system hardware inventory
  - Contents include brick configuration, processor type, main memory size, disk drives, etc.
- `hinv [-v] [-c` *class*`] [-b [`*brick_ID*`]]`
  - *class* may be `serial`, `processor`, `memory`, `scsi`, `otherscsi`, `ethernet`, `otherpci`, `io9`

```
% hinv
Sorry only root user can get scsi information from /dev/xscsi/pci01.01.0/target0/lun0/ds
0 P-Brick
3 C-Brick
12  900 MHz Itanium 2 Rev. 6 Processor
Main memory size: 21.79 Gb
IO9 Controller Card (Silicon Graphics, Inc.) (rev 49). on pci01.01.0
  QLogic 12160 Dual Channel Ultra3 SCSI (Rev 6) on pci01.03.0
     Disk Drive: unit 1 on SCSI controller pci01.03.0-1
     Disk Drive: unit 2 on SCSI controller pci01.03.0-1
  BROADCOM Corporation NetXtreme BCM5701 Gigabit Ethernet (rev 21). on pci01.04.0
```

6.4 Determining Hardware Graph With `topology`

```
% topology

Machine piton.americas.sgi.com has:
  12 cpu’s
   6 memory nodes
The cpus are:
cpu 0 is /dev/hw/module/001c11/slab/0/node/cpubus/0/a
cpu 1 is /dev/hw/module/001c11/slab/0/node/cpubus/0/c
cpu 2 is /dev/hw/module/001c11/slab/1/node/cpubus/0/a
. . .
cpu 10 is /dev/hw/module/001c17/slab/1/node/cpubus/0/a
cpu 11 is /dev/hw/module/001c17/slab/1/node/cpubus/0/c
The nodes are:
node 0 is /dev/hw/module/001c11/slab/0/node
node 1 is /dev/hw/module/001c11/slab/1/node
node 2 is /dev/hw/module/001c14/slab/0/node
node 3 is /dev/hw/module/001c14/slab/1/node
node 4 is /dev/hw/module/001c17/slab/0/node
node 5 is /dev/hw/module/001c17/slab/1/node
The topology is defined by:
/dev/hw/module/001c11/slab/0/node/link/1 is /dev/hw/module/001c11/slab/1/node
/dev/hw/module/001c11/slab/0/node/link/2 is /dev/hw/module/001c17/slab/1/node
. . .
/dev/hw/module/001c17/slab/1/node/link/1 is /dev/hw/module/001c17/slab/0/node
/dev/hw/module/001c17/slab/1/node/link/2 is /dev/hw/module/001c11/slab/0/node
```

6.5 Determining System Load

- `uptime(1)` returns information about system usage and user load
- This system usage information includes
  - Time of day
  - Time since the last reboot
  - Number of users currently on the system
  - Average number of processes waiting to run for the last 1, 5, and 15 minutes

Notes:

User load reported by `uptime` may be inflated if users access data from slow NFS servers, as processes waiting for NFS I/O are placed in the “short wait” queue which is counted as load, even though CPUs are idle.

6.6 Determining Who's Doing What

- `w(1)` command
- Displays who is on the system and what they are doing
  - `uptime(1)` data
  - Duration of user sessions
  - Processor usage by user
  - Currently executing user command

```
% w
  3:15pm  up  2:17,  3 users,  load average: 0.31, 0.21, 0.54
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
gerardo  pts/0    dhcp122.mexico.s 12:59pm  0.00s  0.16s  0.01s  w
pmc      pts/1    cf-vpn-sw-corp-6  2:14pm 21:36   0.14s  0.12s  -csh
n6965    pts/3    fsgi418           2:38pm 36:45   0.11s  0.09s  -csh
```

6.7 Determining Active Processes

- `ps(1)` lets you see a "snapshot" of the process table
- Environment variable `PS_PERSONALITY` allows `ps` to behave the way it does under your favorite Unix flavor
- Frequently used options to `ps`:

-e	Every process running on the system
-l	Long listing
-f	Full listing
-u	List for a specific user only

```
% ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
100 S 22370  4539  4538  0  75   0    -   350 ia64_r pts/0    00:00:00 tcsh
000 S 22370  6191  4539  0  75   0    -   315 ia64_r pts/0    00:00:00 nwscrm6
000 S 22370  6693  6191  0  76   0    -   178 wait4  pts/0    00:00:00 time
000 S 22370  6694  6693  0  75   0    -   286 schedu pts/0    00:00:00 mpirun
000 S 22370  6697  6694  0  75   0    - 1007567 schedu pts/0  00:00:00 nwchem-fb
040 R 22370  6702  6697 99  85   0    - 1778381 -    pts/0    00:17:13 nwchem-fb
040 R 22370  6703  6697 99  85   0    - 1778353 -    pts/0    00:17:14 nwchem-fb
000 R 22370  6759  4539  0  77   0    -   493 -      pts/0    00:00:00 ps
```

6.8 Monitoring Running Processes

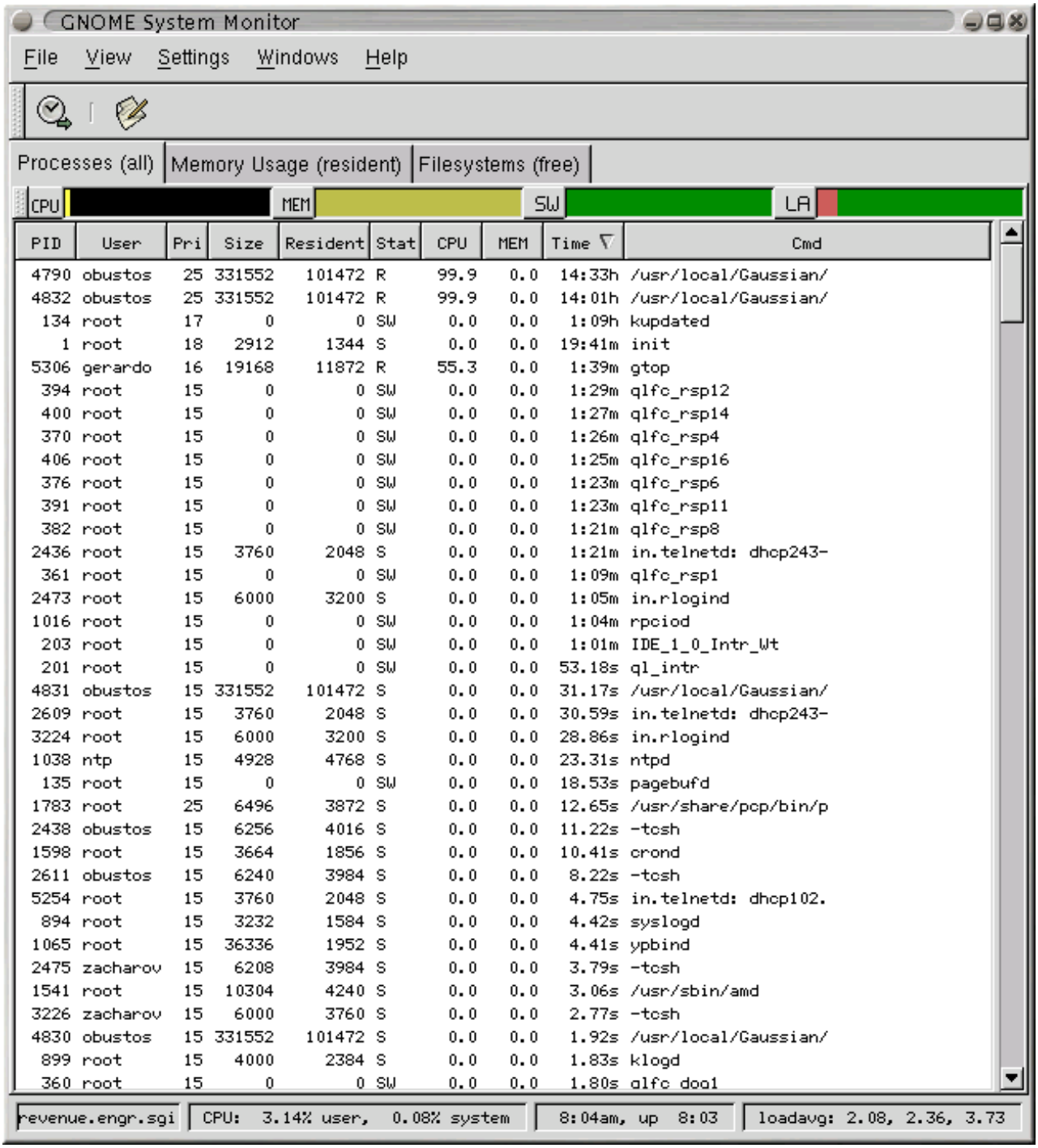
`top(1)` lets you see a sorted list of the top CPU utilization processes updated at a specified interval

- `top [-] [d` *delay*`] [p` *pid*`] [q] [c] [C] [S] [s] [i] [n` *iter*`] [b]`
- `top` has many interactive commands, and it is possible to set up a configuration file, *~/.toprc*

```
  7:56am  up  7:55,  5 users,  load average: 2.21, 3.97, 5.06
Mem:  255353824K av, 3784784K used, 251569040K free,       0K shrd,     432K buff
Swap: 9438176K av,       0K used, 9438176K free                1933792K cached
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 4832 obustos   25   0  323M  99M  320M R    99.9  0.0  73:29 l502.exe
 4790 obustos   25   0  323M  99M  320M R    99.8  0.0  76:41 l502.exe
 5288 gerardo   15   0  4832 2864  4064 R     1.2  0.0   0:00 top
    1 root      15   0  2912 1344   208 S     0.0  0.0   1:51 init
    2 root      0K   0     0    0     0 SW    0.0  0.0   0:00 migration_CPU0
. . .
```

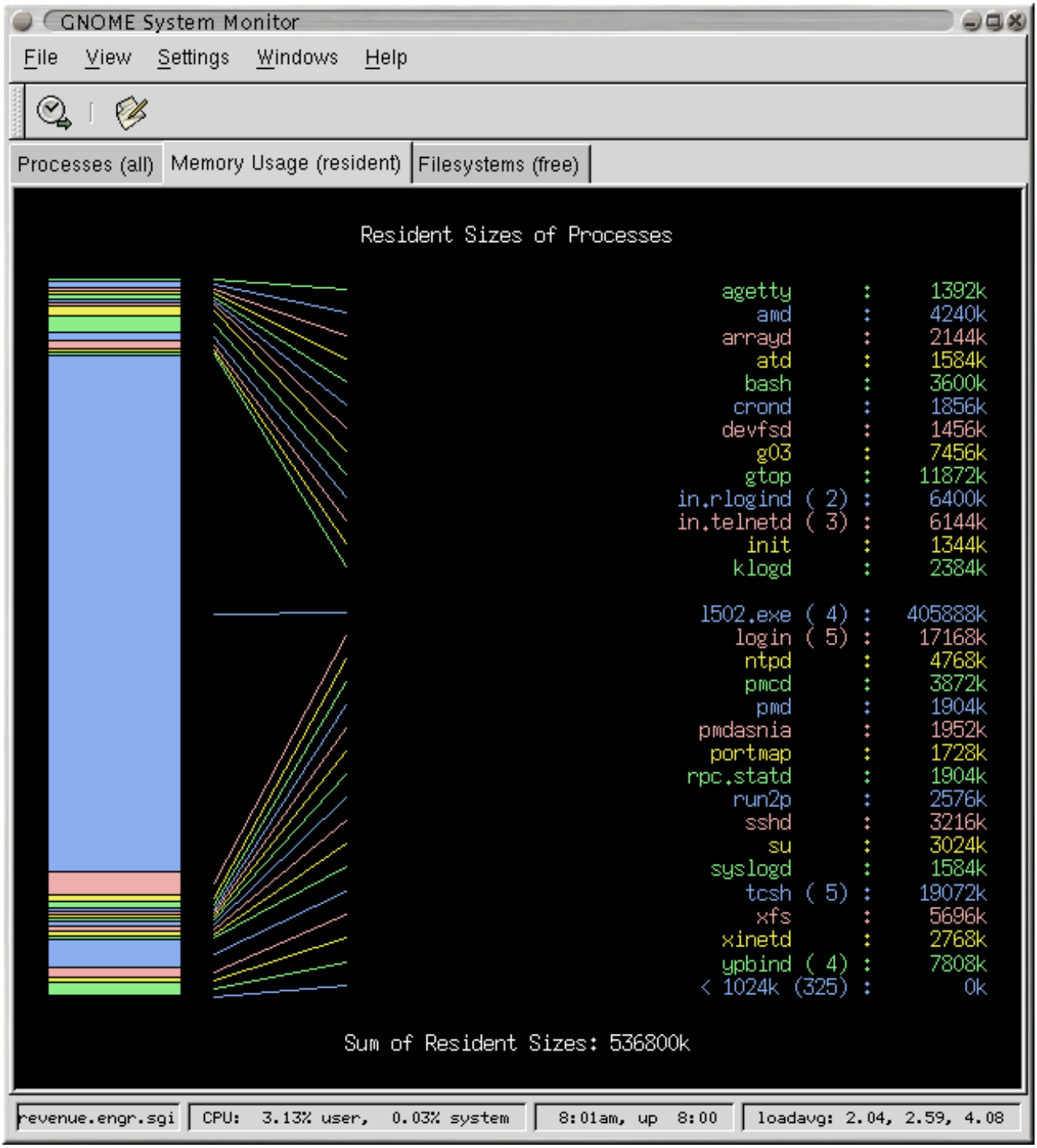
6.9 Monitoring system resource usage — `gtop`

- `gtop` is a graphical tool to monitor system resource usage
  - Processes (based on `top` display), memory, filesystems



6.10 Monitoring Memory Use — `gtop`

- Memory chart shows how memory is used by the various processes present



Lab: Controlling Processors and Processes

1. Determine how many processors you have on your system.
   What is their speed?

   ---

2. List the active processes on your system.
   How many are running?

   ---

3. Pick one process and determine its PID, PPID, process priority, and niceness.
   Which processes are using most of the CPU time?

Module 7

Debuggers

7.1 Module Objectives

After completing this module, you will be able to

- Set Debugger break points and watch points
- View
    - Call stacks
    - Variables
    - Processor registers
    - Memory locations
    - User-defined expressions
    - Arrays and structures
- Follow system calls and interrupts

7.2 Available debuggers

- `idb`: The Intel® debugger — available if you have licenses for the Intel® compilers
  - Fully symbolic debugger
  - Supports debugging of Fortran, C and C++ programs
- `gdb`: The GNU project debugger
  - Supports debugging of C, C++, and Modula-2
  - Supports Fortran 95 debugging when `gdbf95` patch is installed
  - The patch can be found at *http://sourceforge.net/projects/gdbf95/*
- `ddd`: A graphical interface to `gdb` and other debuggers
  - Simple command line option allows selecting debugger to use

7.3 Debugger Syntax

The basic command-line syntax to start the various debuggers is as follows:

- `gdb [`*exec_file* `[`*core_file*`|`*process_id*`]]`
- `idb [-pid` *process_id*`] [-gdb] [`*exec_file* `[`*core_file*`]]`
- `ddd [--debugger` *name*`] [`*exec_file* `[`*core_file*`|`*process_id*`]]`
- `[-pid]` *process_id* lets you debug a running process with the specified process id (as long as there does not exist a file whose name is *process_id*, if `gdb` or `ddd` are used)
- *exec_file* specifies the executable file (optional)
- You can specify a core file (with its executable) to help determine and localize the cause of segmentation violations or other abnormal termination conditions; if an executable core file exists, it is used by default

7.4 gdb

- Help is available
  - `gdb`'s own `help` command
  - `info gdb` at the shell command prompt
  - *http://sources.redhat.com/gdb/onlinedocs/gdb_toc.html*
- Debugging `gcc`-optimized code (`-g -O[1|2|3]`) works fine
- Some challenges with `ecc` at levels 2 and 3
  - Can't print values of register variables
- Assembly level debugging works
  - `gdb` understands rotating registers: useful for stepping through software-pipelined loops

7.5 `idb`

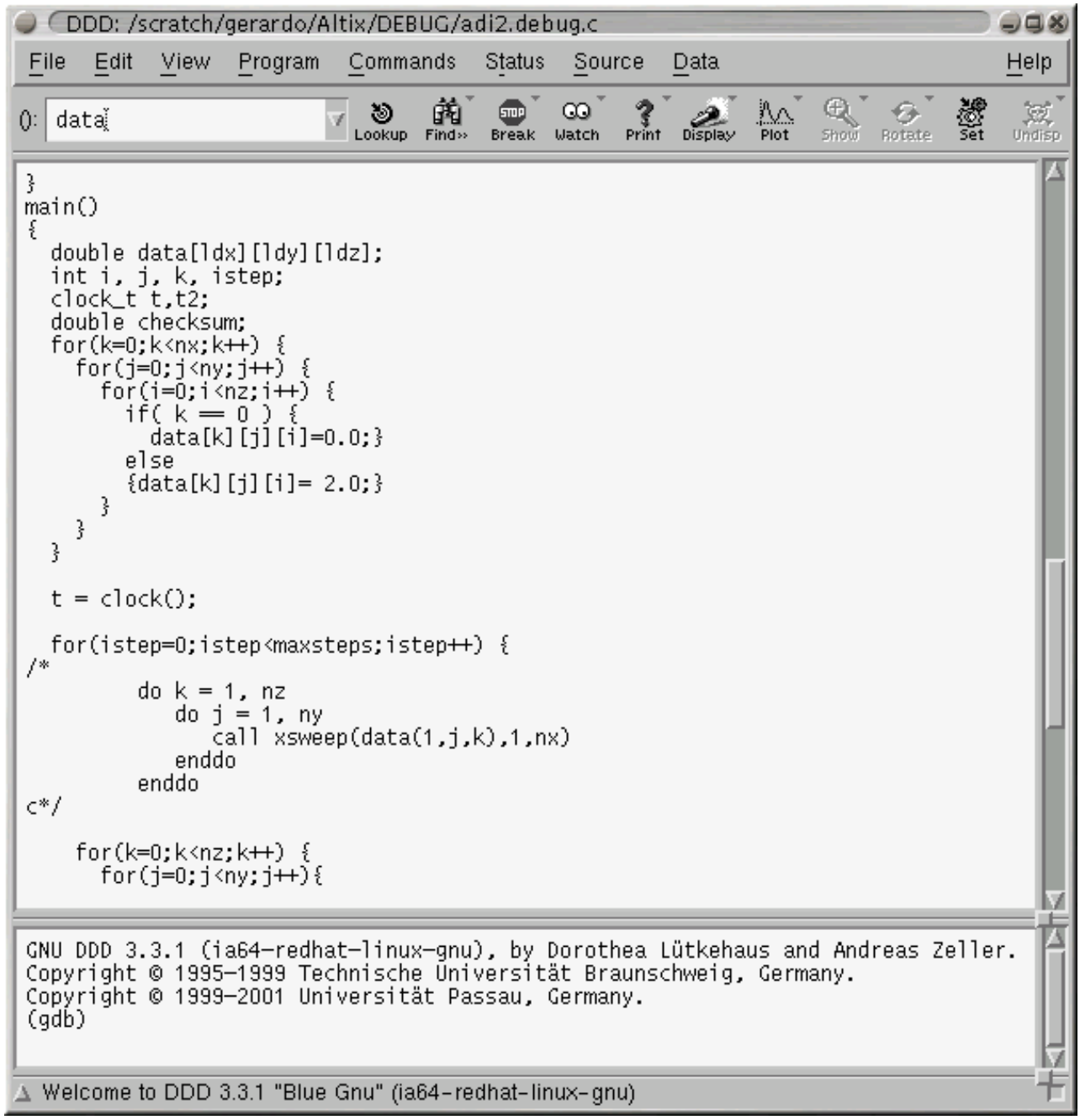
- Intel debugger, part of Intel compiler installation
- Supports C, C++, Fortran 77, Fortran 90
- `dbx-` (default) and `gdb`-like interfaces
- Debugging of optimized code limited
- Good for getting Fortran stack traces
- Supports multithreaded applications (pthreads and OpenMP)

7.6 Data Display Debugger — `ddd`

- Front end GUI to `gdb` and other debuggers, written by Dorothea Lütkehaus and Andreas Zeller
- Home page at *http://www.gnu.org/software/ddd/*
- Features an interactive graphical data display, where data structures are displayed as graphs
- Works best with `gdb`, but can work with `idb` in `dbx` mode
    - **`ddd --debugger idb --dbx ./a.out`**

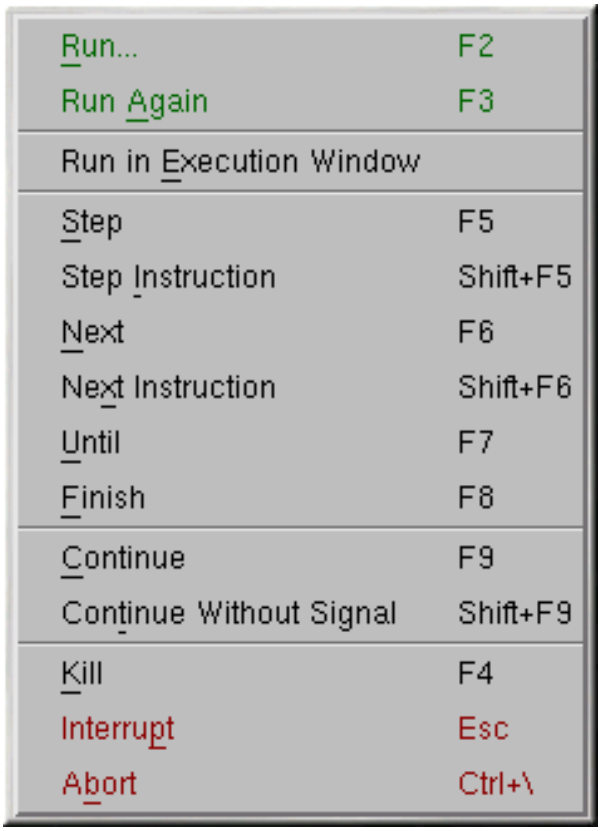
7.7 Main Window

- By default displays the Menu Bar, Tool Bar, Source Window, Debugger Console and Status Line
- The Data Window, when invoked, appears above the Source Window, and an optional Machine Code Window appears below the Source Window



7.8 Command Tool/Program Menu

- A free-standing window displayed when `ddd` starts
- Can be respositioned with Alt-8 or selecting **View→Command Tool** on the Main Window
- Can be configured to appear as a command tool bar above the source window (**Edit → Preferences → Source → Tool Buttons Location**)
- The tool provides easy access to many frequently-used debugger commands
- The same functions are accessible from the Program Menu and the keyboard shortcuts listed in it



7.9 Execution Window

- By default the program being debugged will run in the Debugger Console
- Selecting **View→Execution Window** will open an `xterm` window and enable the **Run in Execution Window** item in the **Program** menu
- Alternately, enabling **Run in Execution Window** in the **Program** menu will cause an Execution Window to be opened when you click on **Run** in the Command Tool

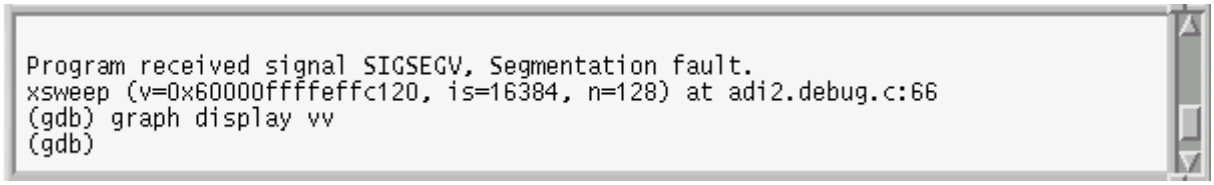
7.10 Tool Bar

The Tool Bar has two parts:

- The argument field, labeled **()**, where any item may be entered
- The tool icons, which represent functions that can be applied to the item in the argument field; only those functions that make sense for the argument will be enabled

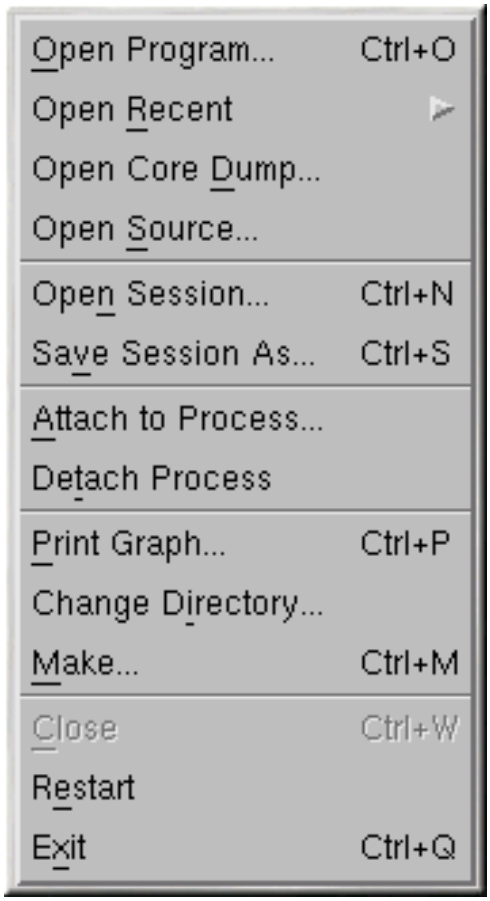
7.11 Debugger Console

- A command-line interface to the debugger is at the bottom of the main window
  - You can use the underlying debugger's commands here
- You can type in debugger commands instead of using the GUI



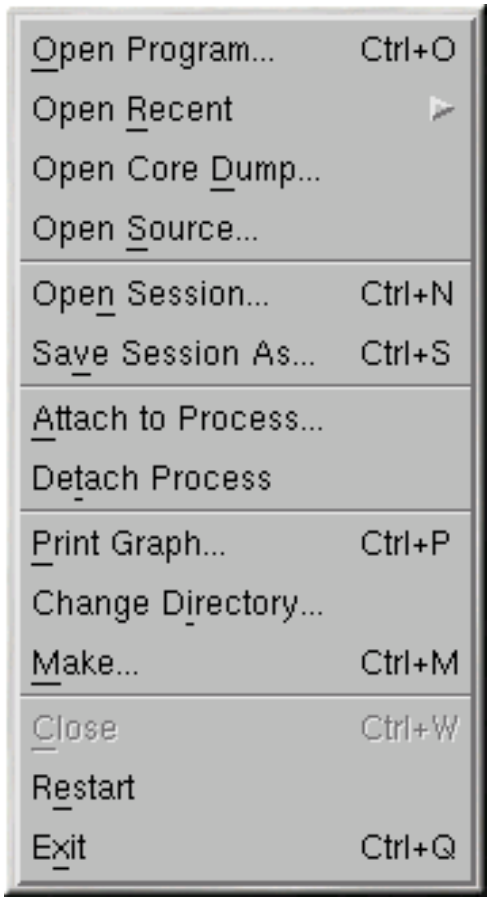
7.12 File Menu

The **File** menu groups file-related operations



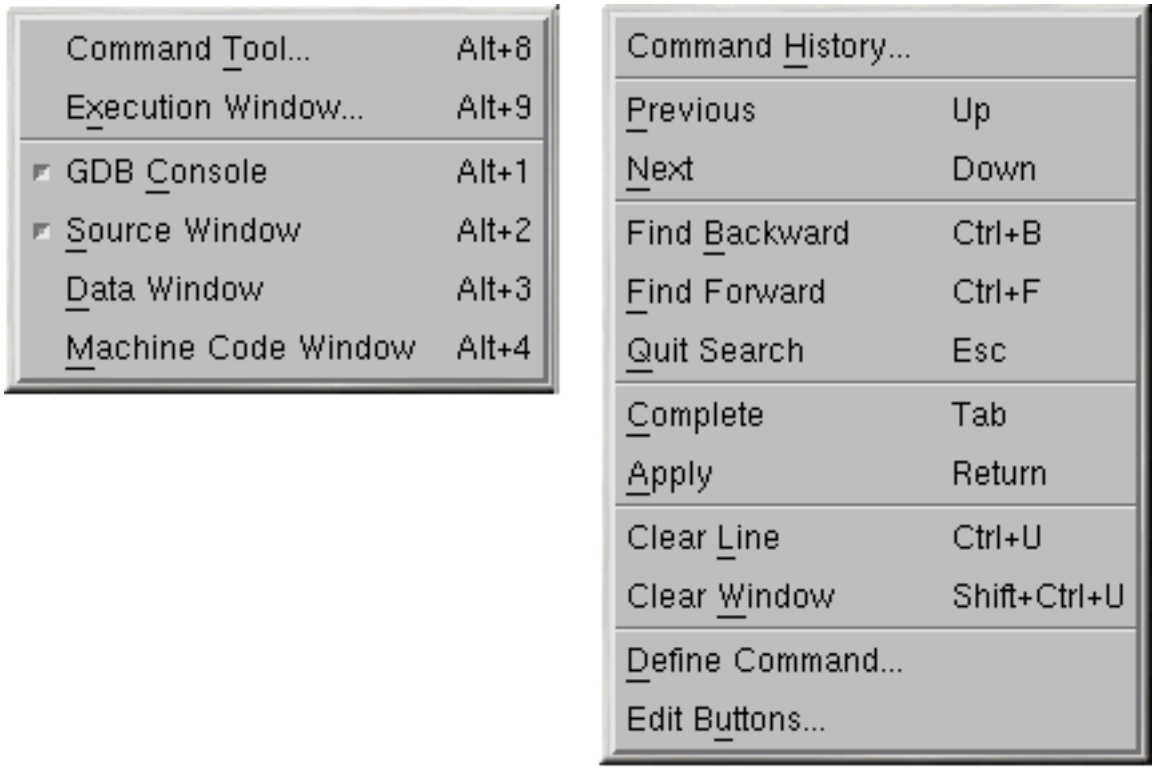
7.13 Edit Menu

- The **Edit** menu allows setting preferences and debugger configuration parameters, besides the usual editing functions



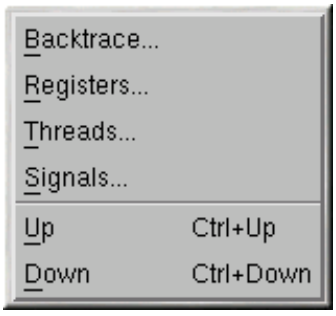
7.14 View and Command Menus

- The **View** menu allows displaying the optional standalone windows and showing or hiding the main view windows
- The **Command** menu helps perform operations related to `ddd` commands



7.15 Status Menu

The **Status** menu has options for examining and modifying the program state: call stack, machine register contents, threads and signals, as well as moving up and down the call stack.



7.16 Setting/Clearing Breakpoints

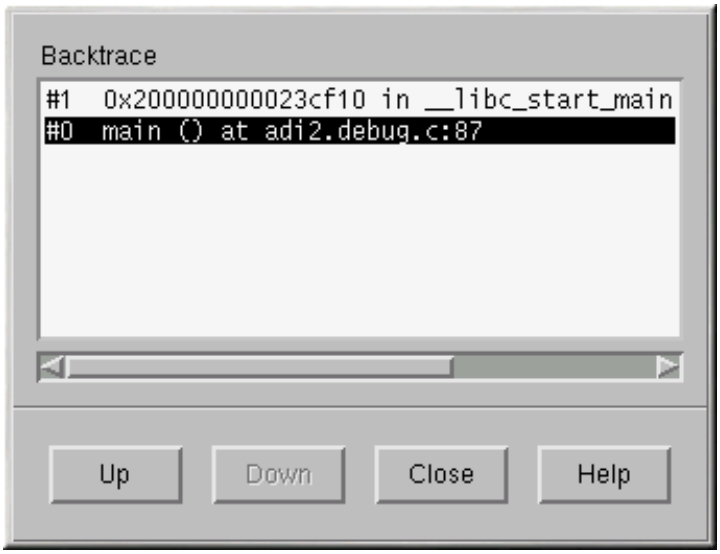
- There are several ways to set breakpoints:
  - Left-click on the whitespace to the left of the source line where you want to set a breakpoint so that it appears in the Argument Field, then use the **Stop** icon on the Tool Bar (which should be labeled "**Break**")
  - Triple left-click on the whitespace to the left of the source line where you want to set a breakpoint
  - Right click-and-hold, choose from the resulting pop-up menu
  - Type the appropriate `break` command in the Debugger Console
- To clear a breakpoint:
  - Left click on the stop sign next to the line with the breakpoint to make it appear in the Argument Field, then use the **Stop** icon on the Tool Bar (which should now be labeled "**Clear**")
  - Right click-and-hold, choose from the resulting pop-up menu
  - Type the appropriate `delete` command in the Debugger Console

7.17 Examining Variables

- There are three ways of showing the value of a variable:
  - Point to it with the cursor: The answer will appear in the Status Line at the bottom margin of the main window
  - Print it on the debugger console with the `print()` command
  - Display it graphically with the `display()` command
- Printing or displaying can be achieved by
  - Left-clicking on the data item, which places it on the Argument Field, and then selecting the appropriate tool
  - Using the right mouse button to click-and-hold on the data item, and selecting from the resulting pop-up menu

7.18 Backtrace Window

The call stack is shown in the Backtrace window, displayed from **Status→Backtrace**



Lab: Setting Break Points (C, C++, Fortran)

Objectives

- Start up a program in the `ddd`
- Use the Debugger to set a break point
- Step into and step over function calls
- Exercise the various ways to examine variables
- Use the Backtrace window

7.19 Edit - Compile - Debug Loop

Editing, compiling, and debugging are tightly integrated in `ddd`:

1. EDIT
   Clicking on **Edit** in the Command Tool opens a window with your favorite X-Windows editor if the environment variable `XEDITOR` is defined, or else an `xterm` with either the editor given by the environment variable `EDITOR`, or `vi`. The program being displayed in the Source Window will be loaded into the editor window.

2. COMPILE
   Clicking on **Make** in the Command Tool executes `make`. A makefile should be present; you can choose a target from the **File** →**Make** menu option. The new executable is attached automatically when you click on **Run**. Note that `idb` does not support the `make` command; however, a `shell make` *target* command may be typed in the Debugger Console.

3. DEBUG
   Continue debugging your code.

7.20 Traps

Traps are used to inspect data at points during execution of the program

There are two types of traps:

- Breakpoint
  - Halts the process so you can examine data manually
  - You can add conditions to control in detail whether the program stops at the breakpoint
- Watchpoint
  - Stops the program when the value of an expression changes

7.21 Setting and Clearing Breakpoints

You can set breakpoints in the following ways:

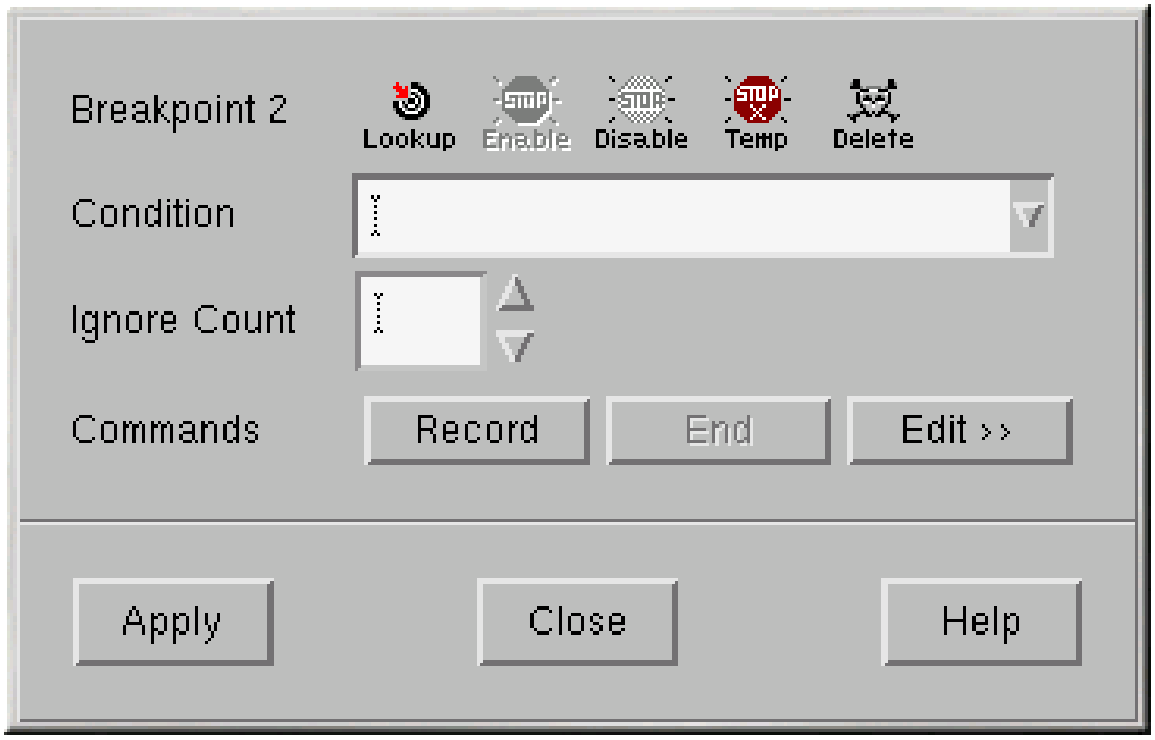
- Triple click on a line with the left mouse button (to set)
- Click and hold the right mouse button, select from the pop-up menu
- Use the **Stop** sign icon on the Control Panel
- Use the Console panel commands (`break`/`clear`)

7.22 Breakpoints - What Can You Examine?

- Variables (value, type, addresses)
- Value of expressions
- Call Stack
- Data structures (graphically)
- Arrays
- Machine code
- Memory/registers

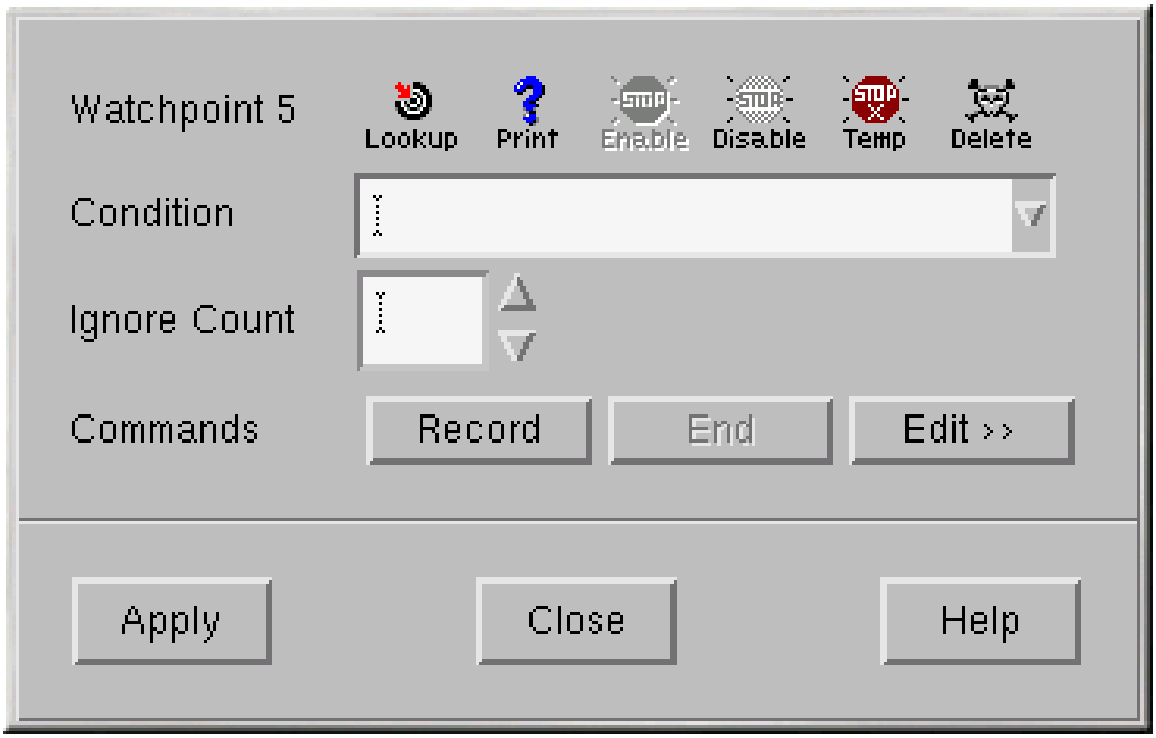
7.23 Breakpoint Properties

- Once a breakpoint is set, right click-and-hold on the **Stop** sign icon at the breakpoint and select Properties from the pop-up menu, or with the breakpoint in the Argument Field, use the **Stop Tool** menu to select Breakpoint Properties
- A pop-up window appears where you can set a condition for stopping, a count for the number of times to ignore the breakpoint before stopping, or debugger commands to execute when arriving at the breakpoint.



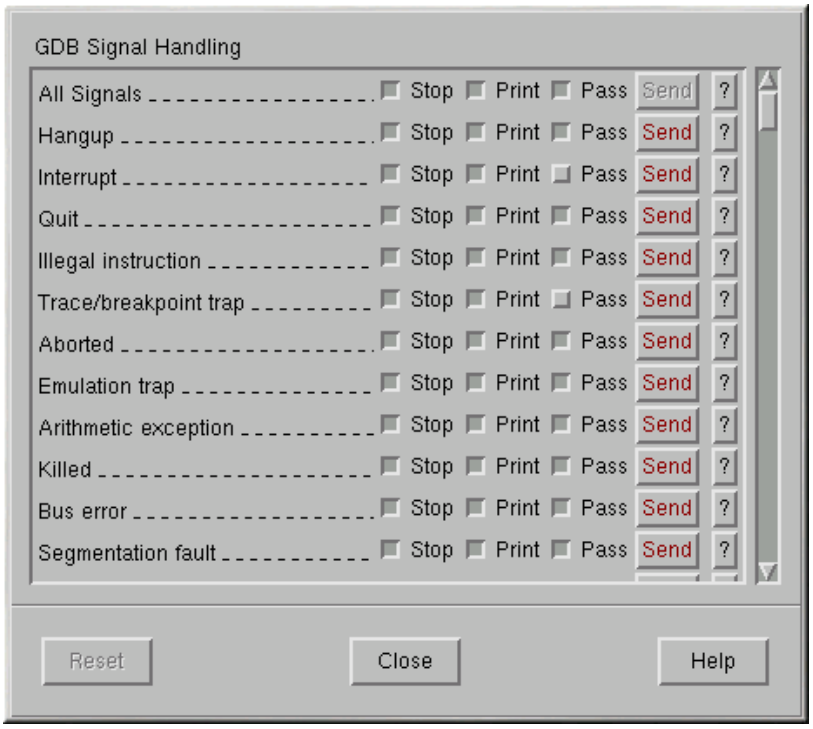
7.24 Watchpoints

- A frequent pointer mistake in C or C++ is overwriting data out of range
- Typically, the overwritten address range is known, but the pointer is not
- Watchpoints watch over a memory location for overwriting
- You can set watchpoints by left-clicking on the variable if it is visible or typing it in the Argument Field and then clicking on the **Watch** button
- Different types of watchpoints may be selected from the menu associated with the **Watch** button
- Once a watchpoint is set, watchpoint properties may be modified in the Watchpoin Properties window



7.25 Signals

- **Status** → **Signals** pops up a panel showing a list of all signals and how `gdb` has been told to handle each one.
    - Stop: Stop the program when the signal happens. (Setting Stop also sets Print.)
    - Print: Print a message when the signal happens. (Unsetting Print also unsets Stop.)
    - Pass: If set, allow the program to see the signal and handle it, if it has installed a handler, or be killed, if no handler has been installed.

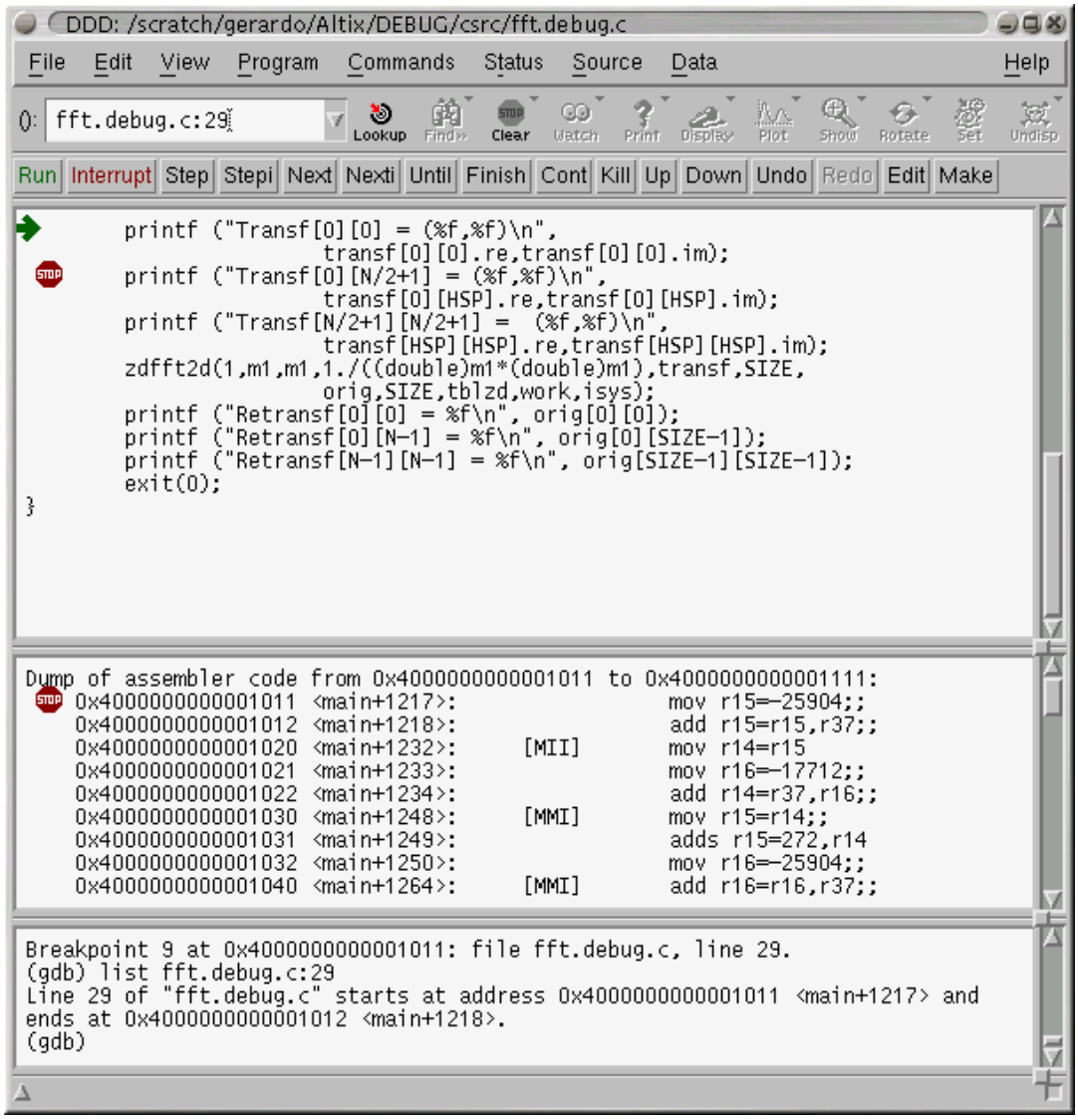


7.26 Data Display

- The Data Display provides a graphical representation of variables, arrays, structures and linked lists
- The Data Display opens automatically when an item is selected for display
    - Left click on the item, click on the **Display** button
    - Right click-and-hold, select **Display** from the popup menu
    - Double click on the variable
- There are many display and formatting options

7.27 Machine Code Window

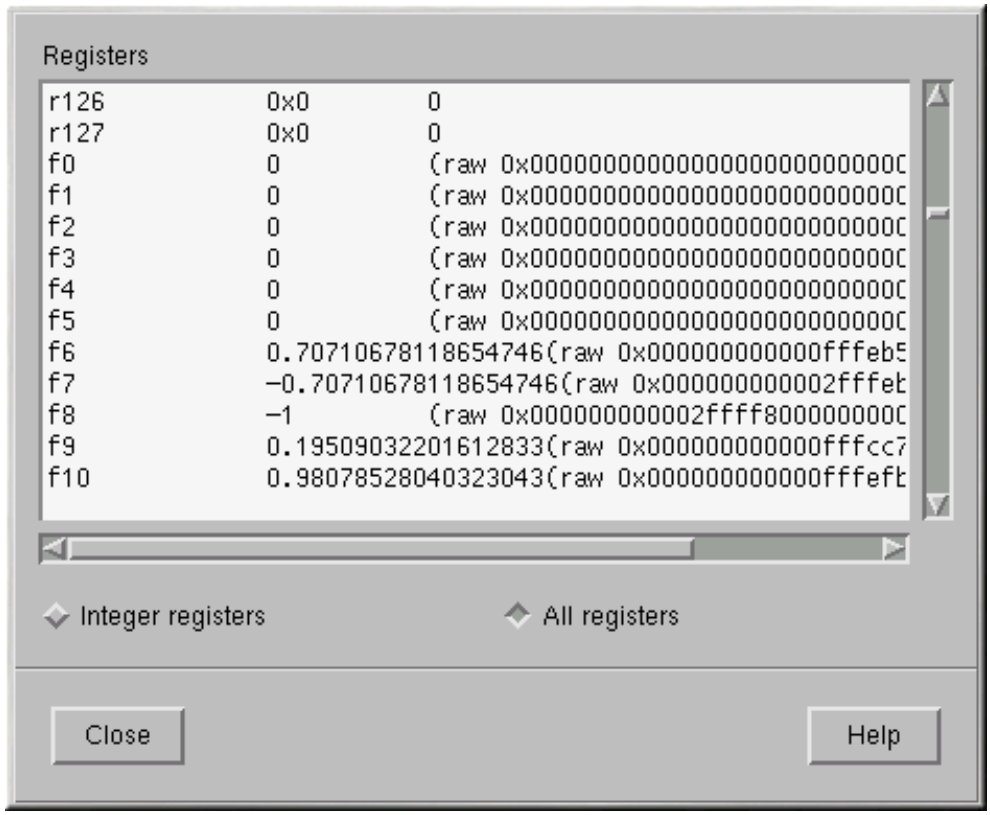
- **Source** → **Display Machine Code** opens up an additional window, usually below the Source Window, that contains the machine code for the current function
- Breakpoints can also be set and cleared in this window
- If source code is not available, only the machine code window is updated



7.28 Register Window

The Register View lets you view processor registers

- The window pops up selecting **Status** → **Registers**



Lab: Using Browsers and Views (C/C++/Fortran)

Objective

Use the Display Window, Machine Code Window, and Register Window

Module 8

Data Decomposition

8.1 Module Objectives

By the end of this module, you should be able to

- Understand why data decomposition is important in parallel computing
- Identify implicit and explicit data decomposition constructs

8.2 Parallelism

To use multiple processors, you must find tasks that can be performed at the same time. There are two basic methods of defining these tasks:

- Functional parallelism
    - Different processors perform different functions
    - Natural approach for programmers trained in modular programming
    - Disadvantages include:
        - Defining functions as the number of processors grow
        - Defining functions that use balanced amounts of CPU time
        - May require a lot of synchronization and data movement
- Data parallelism
    - Different processors perform the same function on different parts of the data
    - Takes advantage of the large cumulative memory
    - Requires that the problem domain be decomposed

8.3 Data Parallelism

Requires three basic steps:

1. Breaking up the data
2. Mapping the data to the processors
3. Dividing the work amongst the processors

The programmer may need to manually perform these steps or use tools that will do some or all of these steps. The first two steps are called data decomposition and, if done correctly, make the third step easy.

8.4 Data Decomposition

- Choose a data decomposition to minimize data transfer
- Choose a data decomposition to balance the load

8.5 Data Decomposition in Data Parallelism

- Data (for example, arrays) is decomposed when there is enough parallel work in code segments that reference it
- In a data parallel environment (loop-level parallelism), data can be decomposed implicitly or explicitly
  - Explicit (manual) decomposition is done by the programmer with no compiler help
    - Message passing (MPI, PVM) and data passing (shmem) codes require explicit data decomposition
  - Implicit (compiler directive based) decomposition is carried out by the compiler
    - MP directives find parallelism in implicitly decomposed data

8.6 Explicit Data Decomposition Example

Consider manually decomposing an array. The array to be modeled is A(4000,8000).

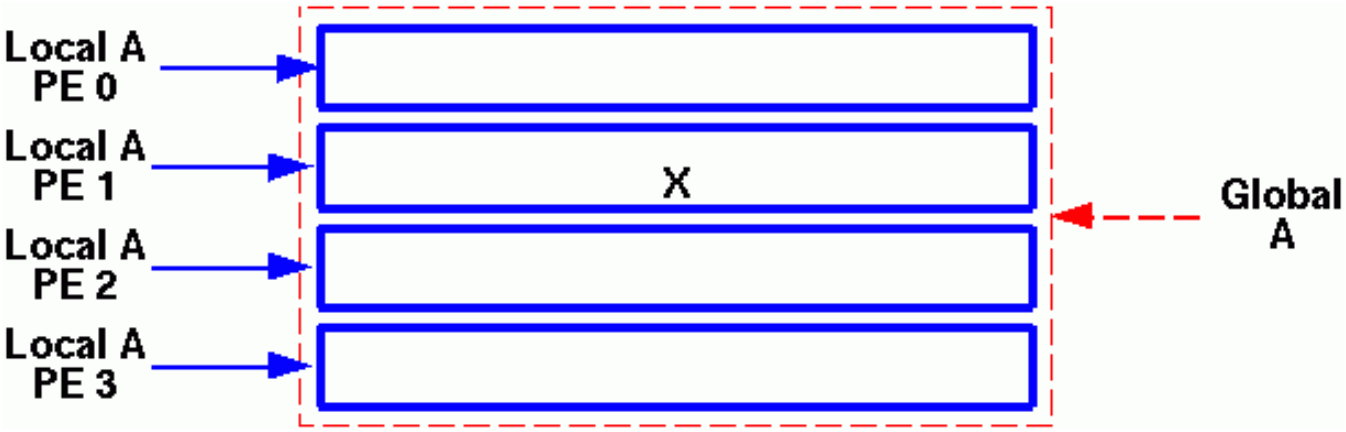
Using four PEs:

- Decomposing the first dimension:
  - Each PE dimensions a local array A(1000,8000)
  - Map each PE's local area into the global array

8.7 Explicit Data Decomposition Example (continued)

You may now need to know where

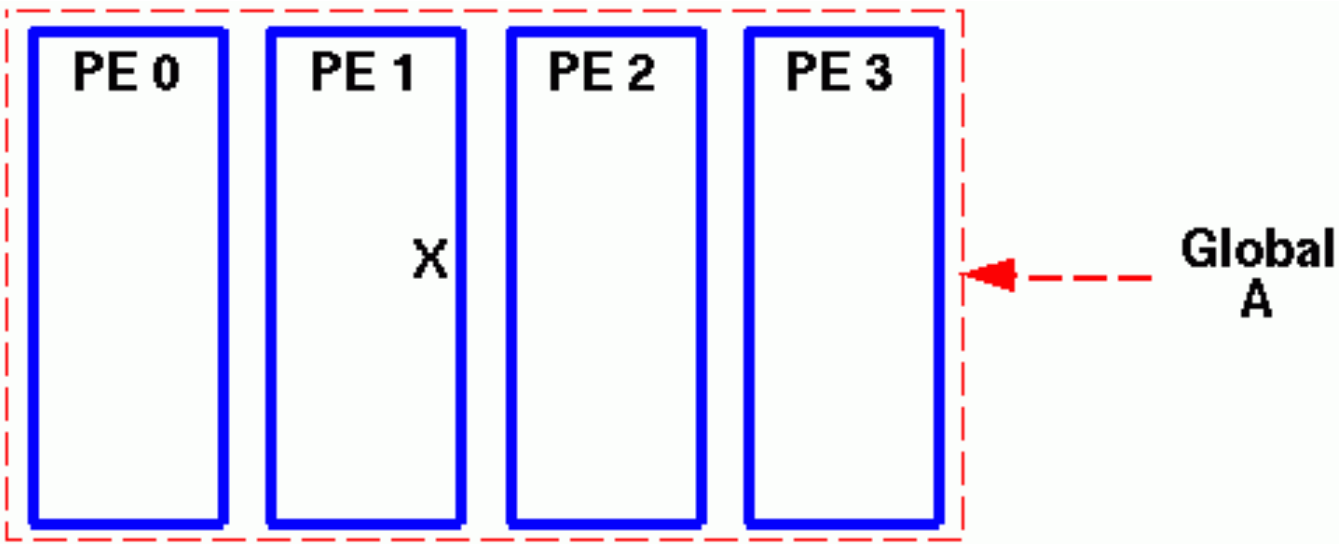
- Global data element A (2000, 4000) maps in local space
  - Global position A(2000,4000) is now A(1000,4000) on PE 1
- Element A (2000, 4000)’s nearest neighbors are in local space:
  - Neighbors are:
    - North A(999,4000) on PE #1
    - South A(1,4000) on PE #2
    - East A(1000,4001) on PE #1
    - West A(1000,3999) on PE #1



8.8 Explicit Data Decomposition Example (continued)

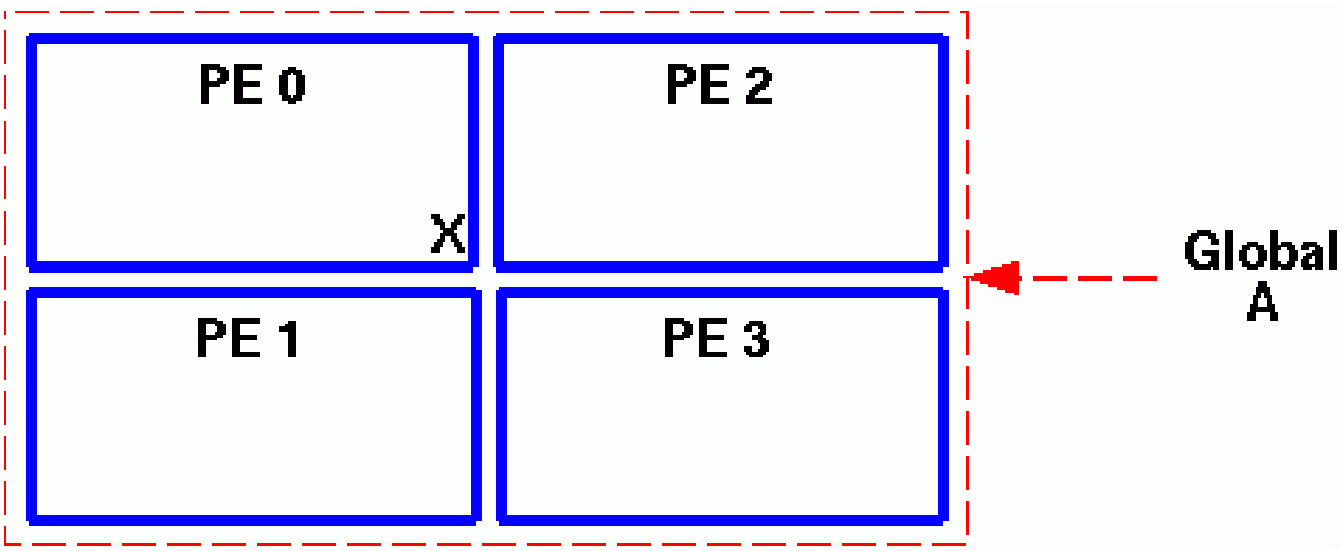
Decomposing the second dimension:

- Each PE allocates a local array A (4000,2000)
- Global position A(2000,4000) is now A(2000,2000) on PE 1
- Neighbors are:
  - North A(1999,2000) on PE #1
  - South A(2001,2000) on PE #1
  - East A(2000,1) on PE #2
  - West A(2000,1999) on PE #1



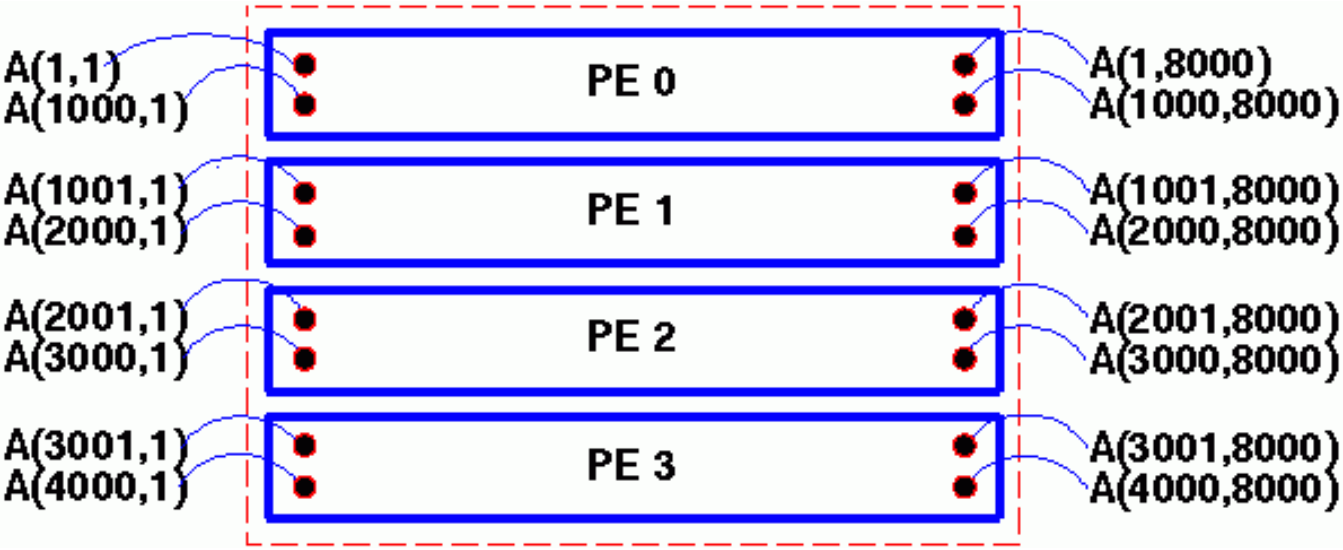
8.9 Explicit Data Decomposition Example (continued)

- Decomposing in both dimensions:
- Global position A(2000,4000) is now A(2000,4000) on PE 0
- Neighbors are:
  - North A(1999,4000) on PE #0
  - South A(1,4000) on PE #1
  - East A(2000,1) on PE #2
  - West A(2000,3999) on PE #0



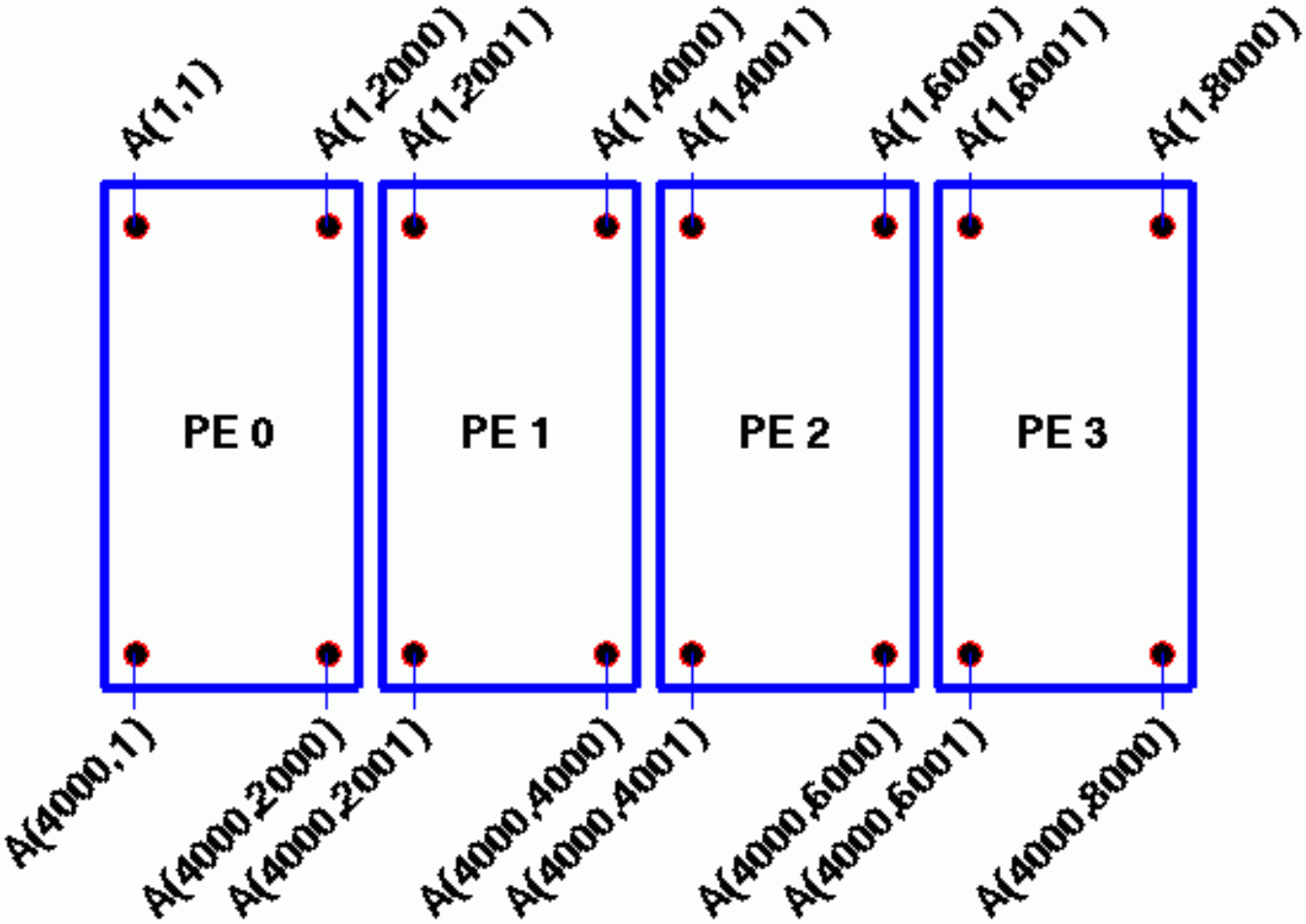
8.10 Implicit Data Decomposition Example

- Implicit data decomposition requires no “resizing” of the data—indices remain global
    - Example: breaking up REAL A(4000,8000) across the first dimension among four processes



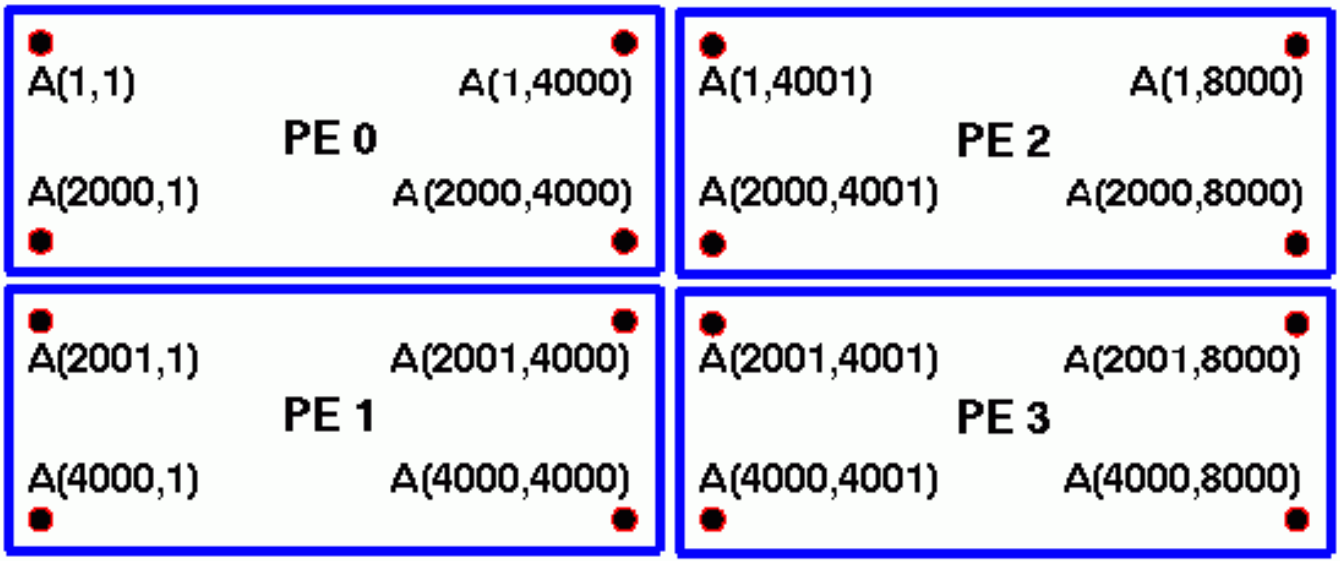
8.11 Implicit Data Decomposition Example (continued)

Decomposing the second dimension:



8.12 Implicit Data Decomposition Example (continued)

Decomposing in both dimensions:



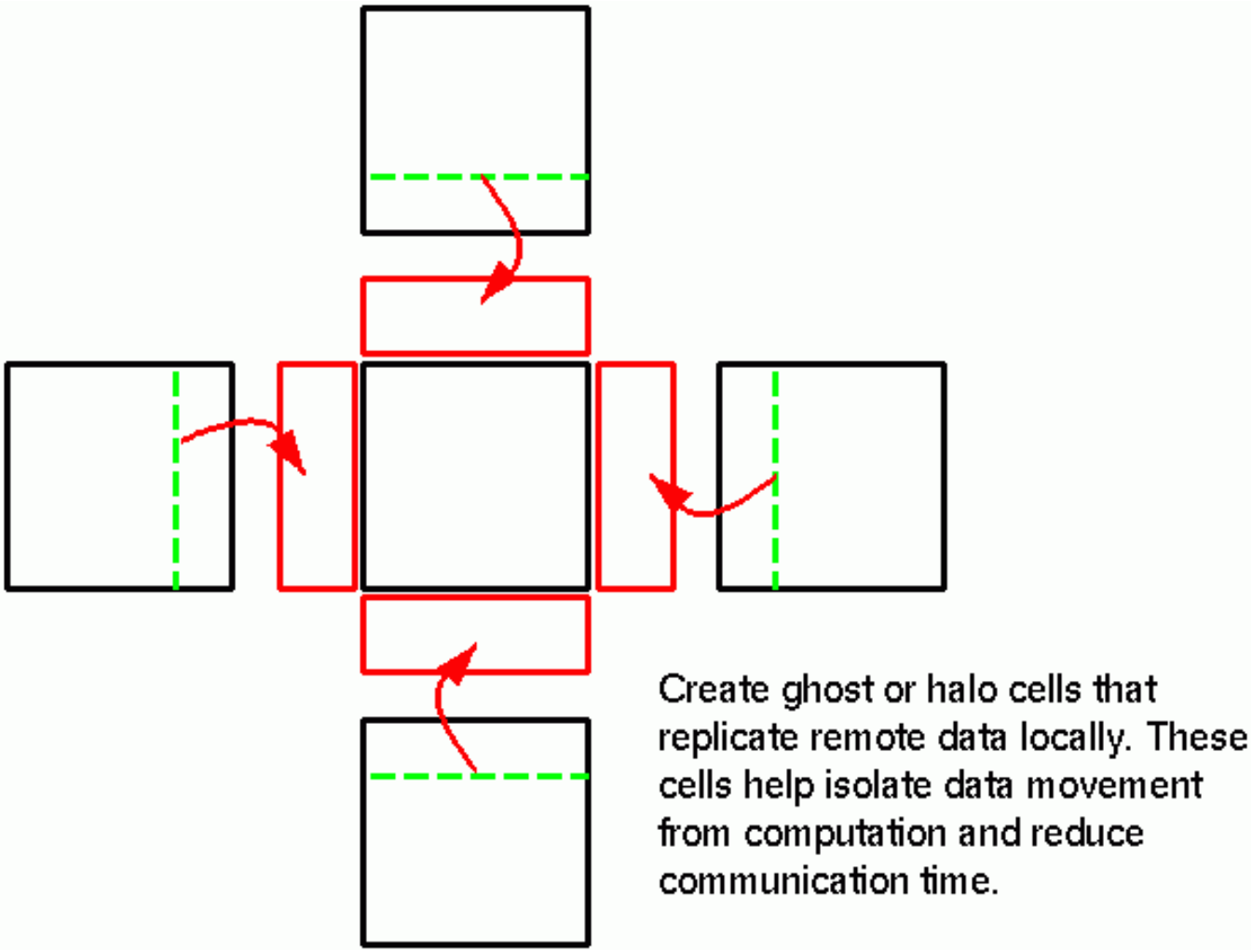
8.13 Implicit Versus Explicit Data Decomposition

- Implicit decomposition advantages:
  - No data resizing
  - All communication/synchronizations handled by the compiler implicitly
  - Source code easier to develop/port/read
  - Source code is portable to other systems with the corresponding programming model support (e.g., OpenMP)
- Explicit decomposition advantages:
  - All communication/synchronization calls inserted by the programmer (full control/knowledge of program flow)
  - Source code is portable to other systems (MPI, PVM)
  - Code performance can be better than implicitly parallelized codes

8.14 Implicit Versus Explicit Data Decomposition (continued)

- Implicit decomposition disadvantages:
  - Data communication hidden from user
  - Compiler technology not mature enough to deliver top performance consistently (yet)
- Explicit decomposition disadvantages:
  - Harder to program
  - Source code harder to read
  - Source code is longer (typically 40% or more)

8.15 Data Replication

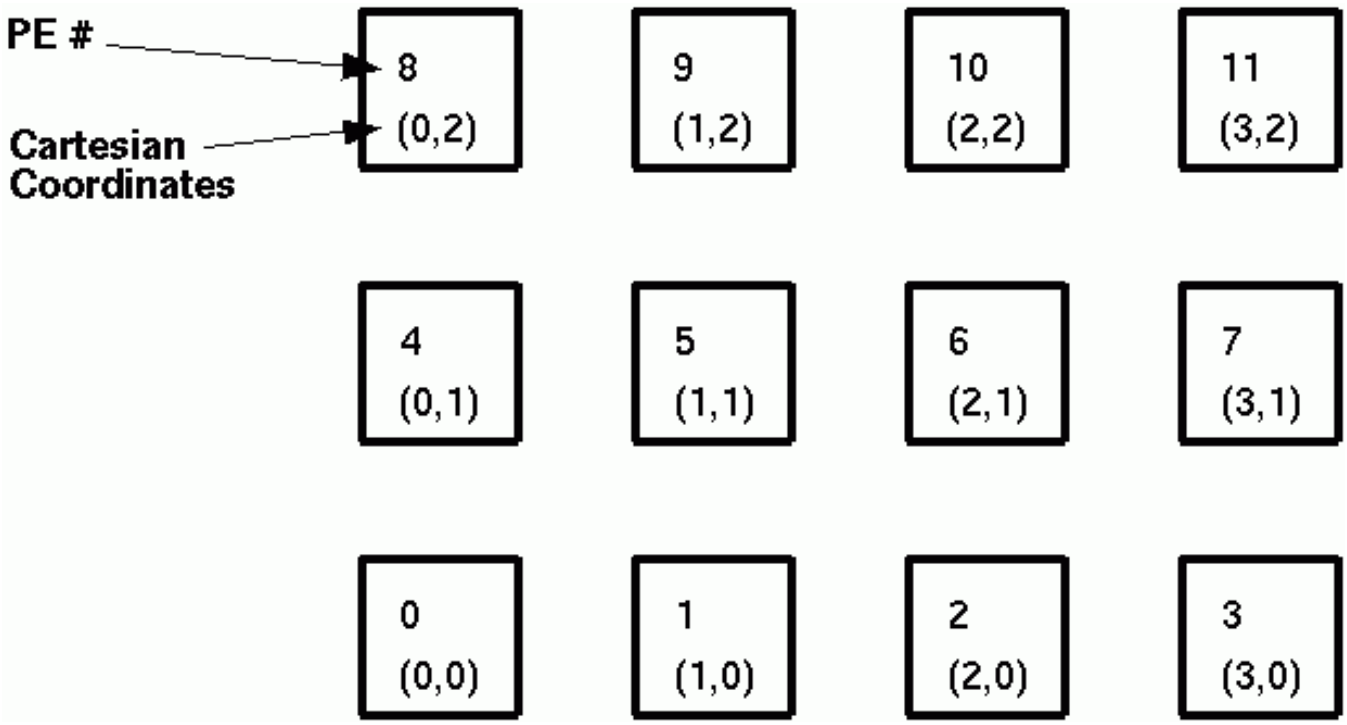


8.16 Tools

Some tools allow the programmer to view the process grid as a cartesian plane, using (X, Y) coordinates or column or row operations.

- MPI
  - MPI_CART_CREATE defines the size and shape of the process grid
  - MPI_CART_COORDS returns the coordinates of a process
  - MPI_CART_SHIFT returns a neighbor’s rank (process number)
- BLACS
  - BLACS_GRIDINIT allows the user to define the size and shape of the process grid
  - BLACS_GRIDINFO returns the calling process’s grid coordinates
  - Matrices are sent and received by grid coordinates

8.17 Processor Grids



Module 9

Open MP

9.1 Module Objectives

After completing this module, you will be able to

- Explain the origins of OpenMP
- Recognize OpenMP constructs and directives
- Understand incremental parallelism
- Write a parallel code using OpenMP

9.2 Motivation for OpenMP

- 1997: No standard for shared memory parallelism
  - Each SMP vendor had proprietary API
  - Portability only through MPI, PVM
- Parallel application availability
  - ISVs have big investment in existing code
  - Message passing ports are labor intensive
- OpenMP allows a partial port—incremental parallelism

Notes:

The SMP architecture works well with a variety of styles of parallel processing. Because all processors have access to all of memory, adding processors to assist a processor that is already executing a code is fairly straightforward. Additional processors can be added to the computationally intense areas of the code, and released when this kernel of work is completed. This ability allows the programmer to execute large portions of the code in a single CPU and modify only the computational kernels, yet still gain significant performance when multiple processors are available. This is known as incremental parallelism.

9.3 What Is OpenMP?

- Shared memory multiprocessing API
  - Standardizes existing practice
- Portable and standard
  - SGI, Compaq, Kuck and Assoc., DOE, IBM, Sun, HP, Intel®, and so on
  - Both UNIX and NT®
- First Fortran OpenMP implementation released by SGI October 1997
- First C/C++ OpenMP implementation released by SGI November 1998
- The Intel® compilers that run on the SGI Altix systems support OpenMP

Notes:

A variety of APIs have been available on SMP machines for many years. All APIs perform roughly the same functions, but they use different syntax and different execution models. Not all vendors supported all constructs, such as ATOMIC updates. These models varied greatly on when and where they used implied synchronization. These differences kept parallel codes from being portable.

9.4 Parallel Programming Execution Model

- OpenMP uses a fork and join execution model
- Execution begins in a single master thread
- Parallel regions are executed by a team
- Execution returns to a single thread at the end of a parallel region
- OpenMP provides constructs for
    - Data parallelism—Loop level
    - Functional parallelism

9.5 OpenMP Overview

- Scalable: fine and coarse grain parallelism
- Emphasis on performance
- Exploit strengths of shared memory
- Directive-based (plus library calls and environment variables)

Notes:

Using directives allows a code to be compiled to run on a single processor or by using a compiler option, compiled to run on multiple processors. It also allows execution of the code to easily move between single threaded and multithreaded regions.

9.6 Directive Sentinels

Fortran

- `!$OMP`
- `C$OMP`
- `*$OMP`

C/C++

- ```
  #pragma omp directive
  { structured block }
  ```

Notes:

These sentinels are recognized by the compiler when the OpenMP option is turned on. A variety of directives follow the sentinel to control the parallel execution of the code. In fixed form Fortran, the sentinels must appear in column one, and continuation directives must have the form `C$OMP+`, where `+` is any non-blank or non-zero character. In free form, `!$OMP` can appear in any column as long as it is proceeded only by white space. Directives are continued by using an ampersand (`&`) as the last non-blank character of the line. Continuation lines start with `!$OMP` or `!$OMP&` as the first non-blank characters.
  ```

9.7 Conditional Compilation

OpenMP allows certain statements to be conditionally compiled

Fortran

- `!$` Fixed or free form
- `C$` Fixed form
- When compiled with the OpenMP option these sentinels are replaced in the code with two spaces.

```
!$    record = ( loopcnt - 1 ) *
!$   + myid
```

C/C++

- Use the macro `_OPENMP`
- Can NOT be used with `#define` or `#undef`

Notes:

One of the goals of OpenMP is for a code to be compiled and run on a single-processor or multiprocessor system. Using sentinels that appear to a non-OpenMP compiler as a comment works well in most cases. Many times, some additional code needs to be added to the serial code to make it work properly in a multiprocessing environment; examples include calculations based on processor number or calls to OpenMP libraries. These calculations would be wasted on a single-processor system, and the library calls would be undefined. Use these sentinels to conditionally compile the code. When the OpenMP flag is on, these sentinels are replaced with two spaces; the rest of the line must be proper Fortran syntax. The line can include line numbers and executable statements.

9.8 Parallel Regions

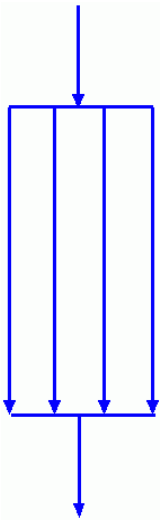
Fortran

```
!$ omp parallel
...
!$ omp end parallel
```

C/C++

```
#pragma omp parallel
{ structured block }
```

- Be careful of “throws”



Notes:

OpenMP codes execute as a single thread (master) until an `omp parallel` directive is encountered. All threads that join the group at this directive will execute all the code until an `omp end parallel` directive is encountered. If no further `omp` directives are in this region, then this is redundant code—parallel, but redundant. The master process will continue beyond the parallel region only after all threads in the team have finished.

The number of threads in the team is implementation dependent. The number of threads is constant within a parallel region, but the number may be changed by the user or the system between parallel regions.

If a thread in a team encounters another parallel region, it creates a new team to execute the parallel region and it becomes the master for the new team. By default, this team has only one player, the master.

A “throw” executed inside a parallel region must be caught by the same thread and must continue execution inside the dynamic extent of the same structured block.

9.9 Parallel Regions (continued)

Optional clauses

The following optional clauses may be added to the `omp parallel` directive:

```
private ( list )
shared ( list )
default ( private | shared | none )
firstprivate ( list )
reduction ( ( operator | intrinsic ) : list )
```

- operator must not be overloaded

```
if ( scalar_logical_expression )
copyin ( list )
```

Notes:

Part of the analysis required to execute a region of code in parallel is to scope the data items. By default, all data items are `shared`, unless the default is changed by the default clause.

Each thread will have its own storage for any variable in the `private` list. The value of the variable is undefined upon entry to and exit from the region. This behavior can be overridden by declaring the variable in the `firstprivate` list. This variable will be `private`, but initialized to the value of the same variable in the master thread.

The variables on the `shared` list have only one storage location which all processes may access if and when they reference these variables.

For variables listed in the `reduction` clause, a special `private` variable is created and initialized for each thread. At the end of the region, the operator or intrinsic is used to reduce the partial answers from each thread to one final answer, which is stored in the original variable. Possible operations and intrinsics include `+`, `-`, `*`, `AND` and `OR` (both logical and bitwise), `MAX` and `MIN`. OpenMP 1.0 only supports scalar reduction variables, but OpenMP 2.0 will support arrays as reduction variables.

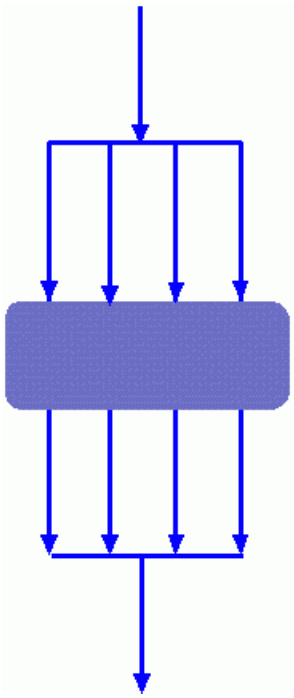
When the `if` clause is included, the region will be executed in parallel only if the `scalar_logical _expression` is true. One reason a region may be conditionally executed in parallel is that the amount of work varies from run to run, or even within a run.

The `copyin` clause applies to named `COMMON` blocks that have been declared `threadprivate`. All variables or whole common blocks in the list will be initialized from the master thread.

The `threadprivate` is a special declaration. It immediately follows the declaration of a variable that should be private to threads when in a parallel region, but whose value should be retained between parallel region invocations (`private` variables are automatic: they are created each time the parallel region is entered and conceptually destroyed each time the parallel region is exited).

9.10 Work Sharing: DO / FOR

```
!$ omp parallel        #pragma omp parallel
...                    { ...
!$ omp do                #pragma omp for
   do i = 1, n               for (i=0;i<n;i++)
   a(i) = 0                  a[i]=0.0
   enddo                  }
...
!$ omp end parallel
```



Notes:

The DO is the natural way to operate on arrays in Fortran 77, but Fortran 90/95 introduces array operations and FORALL, work constructs that only `imply` loops. While the autoparallelizer will try to autoparallelize such constructs, it may not do so in the manner one would want to (e.g., if the last array index only has a limited range). OpenMP 2.0 introduces a `workshare` directive that you can place to indicate that you want the compiler to distribute the units of work in such Fortran 95 constructs.

9.11 Work Sharing: DO/FOR (continued)

DO/FOR clauses

The following clauses can be added to the DO/FOR construct:

```
lastprivate ( list )
schedule ( type [ , chunk ] )
"type" is :
    static
    dynamic
    guided
    runtime
ordered
```

There is an optional `END DO` directive, with an optional clause:

```
nowait
```

Notes:

The `lastprivate` clause will cause variables on the list to be updated by the thread that executes the last iteration of the loop. Within the loop, each thread will have a `private` copy of these variables.

Iterations of a loop are assigned to threads based on the `schedule` clause. The clause is not required, but the standard does not specify a default.

`schedule can be`

- `(static,`*`chunksize`*`)`
  `chunksize` number of iterations of the loop to be assigned to each thread in a round-robin fashion, based on thread number. If *`chunksize`* is not specified, each thread gets exactly one chunk; the chunk size in this case is computed by dividing the number of iterations by the number of threads and any remainder is distributed among the threads in a manner determined by the implementation.

- `(dynamic,`*`chunksize`*`)`
  `chunksize` number of iterations of the loop will be assigned to threads as they become available for work.

- `(guided,`*`chunksize`*`)`
  the number of iterations assigned to threads decreases as work is completed. *`chunksize`* is the minimum number of iterations assigned.

- `(runtime)`
  scheduling will be determined by the environmental variable `OMP_SCHEDULE`.

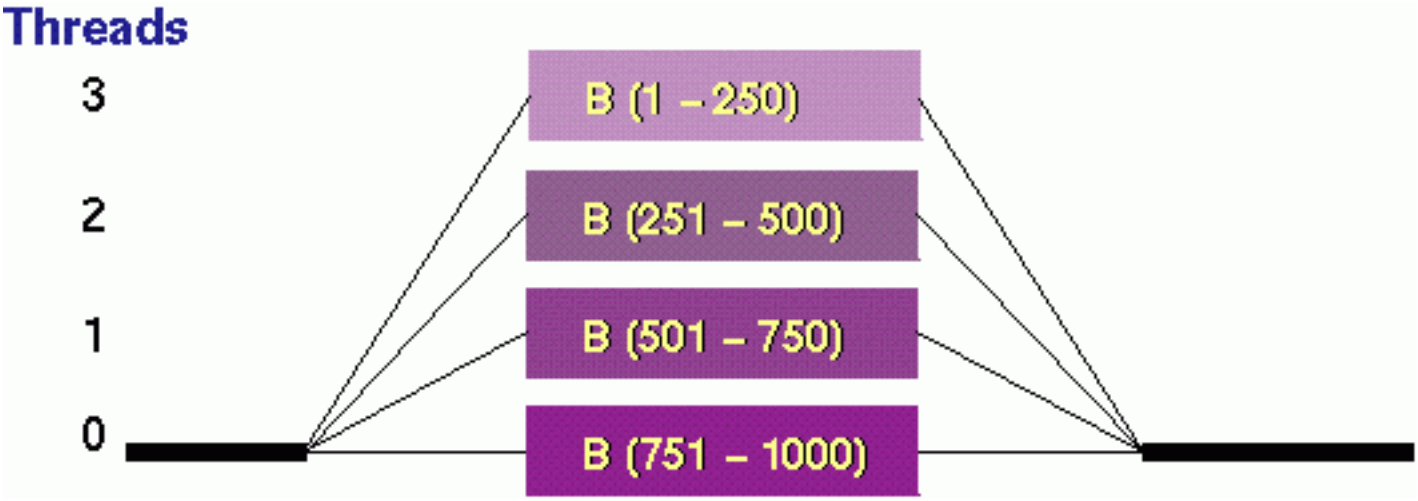
The `ordered` clause allows the loop to contain a region of code which will be executed in the same iteration order as it would execute on a single processor.

The `nowait` clause on the `END DO` directive will allow a thread to continue executing when all the iterations of the do loop have been assigned, but not yet completed. This is particularly important if one wants to free the CPUs occupied by threads that are idle while the remaining iterations complete; without the `nowait` clause the idle threads will busy-spin waiting for all iterations to complete, regardless of environment settings.

The `nowait` clause goes on the `for` pragma in C/C++.

9.12 Loop scheduling types

- If schedule is `static` with no `chunksize` specified each threads gets one chunk

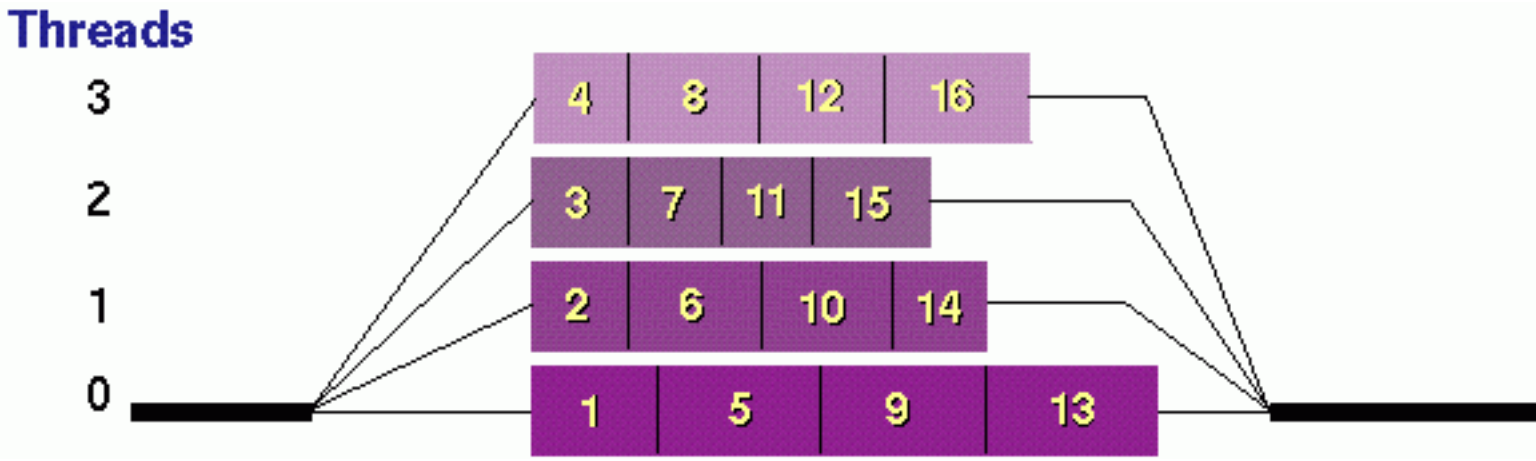



```
!$ omp parallel
...
!$ omp do
   do i = 1, 1000
   ! code block B
   enddo
...
!$ omp end parallel
```

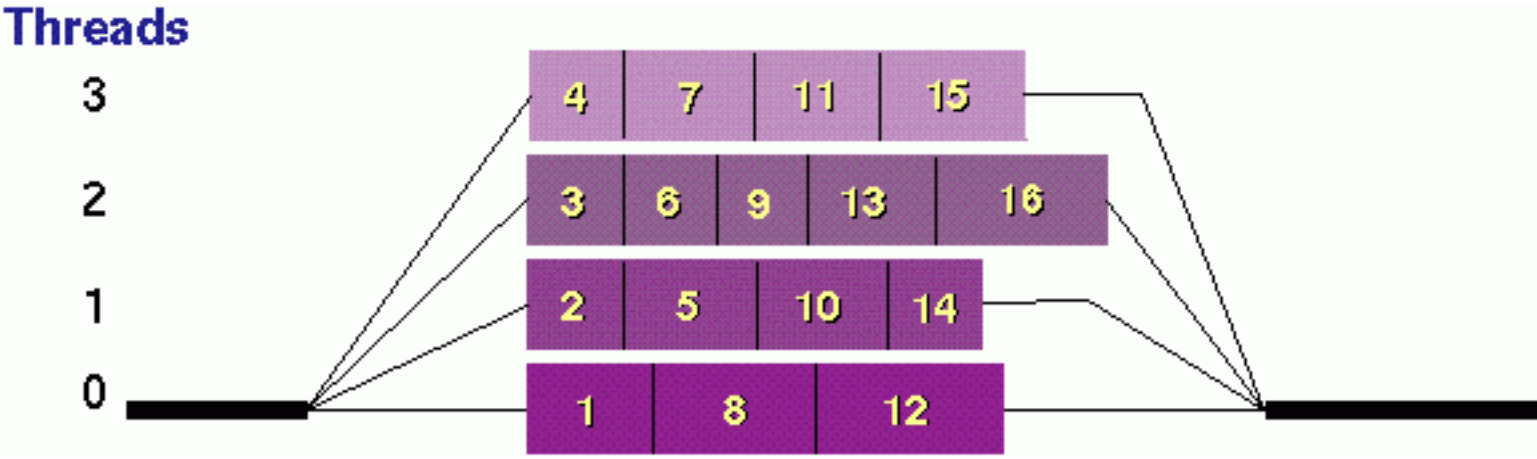
```
#pragma omp parallel
{ ...
  #pragma omp for
          for (i=0;i<1000;i++)
          /*code block B*/
 }
```

9.13 Loop scheduling types (continued)

- `static` with a *`chunksize`* distributes work cyclically in blocks of *`chunksize`* iterations

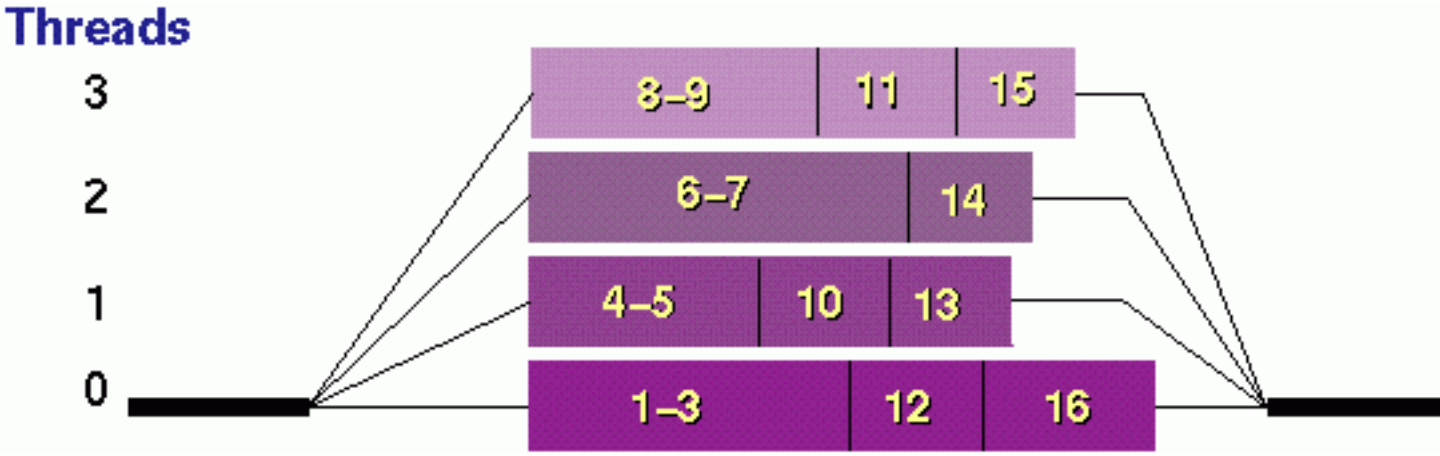


- `dynamic` dynamically allocates work in blocks of *`chunksize`* iterations



9.14 Loop scheduling types (continued)

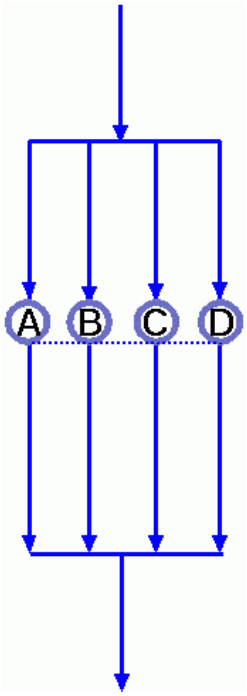
- `guided` is dynamic scheduling that starts with large chunks and ends with smaller chunks; *`chunksize`* is the minimum number of iterations assigned



- `runtime` indicates that the scheduling type and chunk size are to be set at run time by the environment variable `OMP_SCHEDULE`

9.15 Work Sharing: Sections

```
!$ omp parallel          #pragma omp parallel
      ...                { ...
!$ omp sections          #pragma omp sections
!$ omp section             { ...
      ...                #pragma omp section
!$ omp section                 { structured block }
      ...                #pragma omp section
!$ omp end sections            {structured block }
      ...                  }
!$ omp end parallel      }
```

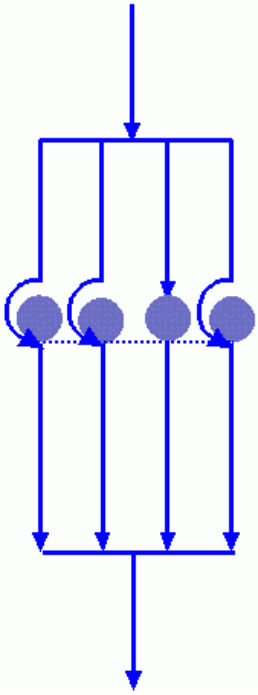


Notes:

Another method of using multiple processors is to have them execute different regions of code, which is known as *functional parallelism*. Each section of code is executed by a thread. Threads will synchronize at the `end sections` directive unless it has a `nowait` clause. The `sections` directive has data scoping clauses similar to the `do` directive.

9.16 Work Sharing: Single

```
!$ omp parallel                 #pragma omp parallel
      ...                       { ...
!$ omp single                   #pragma single
      ...                          {structured block }
!$ omp end single [nowait]      }
      ...
!$ omp end parallel
```

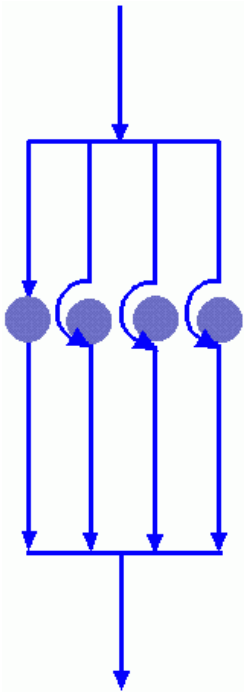


Notes:

The first thread to arrive at the `single` directive executes the code block. All other threads skip this block and wait for the rest of the team to reach the `end single` directive. The waiting can be overridden with a `nowait` clause on the `end single` directive.

9.17 Work Sharing: Master

```
!$ omp parallel          #pragma omp parallel
      ...                { ...
!$ omp master            #pragma master
      ...                    {structured block }
!$ omp end master              ...
      ...                }
!$ omp end parallel
```



Notes:

The `master` directive forces a block of code to be executed only by the master of the team. The rest of the team skips this block of code and continues processing.

9.18 Work Sharing: Shorthand

```
!$ omp parallel do
      do i=1,n
      ...
      enddo
!$ omp end parallel do

#pragma omp parallel for
   for (...) {...}
```

```
!$ omp parallel sections
!$ omp section
      ...
!$ omp section
      ...
!$ omp end parallel sections

#pragma omp parallel sections
#pragma omp section
  { structured block }
#pragma omp section
  { structured block }
```

9.19 Work Sharing: Orphaning

- Worksharing constructs may be outside lexical scope of parallel region

```
!$ omp parallel                    subroutine foo (...)
      ...                          ...
      call foo (...)          !$ omp do
      ...                          do i=1,n
!$ omp end parallel                ...
                                   enddo
                                   end
```

Notes:

If `foo` is called from outside a parallel region, the `DO` loop is executed by a team of one, the master. `foo` cannot be legally called from within a work sharing `DO` loop.

If `foo` also included a parallel directive, a second team would be created to execute this parallel region, with the encountering thread as the master. Usually this team will have only one member. This behavior may be changed by setting the `OMP_SET_NESTED` environmental variable to `TRUE`. Although this condition must be supported by all OpenMP implementations, the resulting team may still have only one member.

Note that when calling functions or subroutines from within a parallel region, variables declared in the function or subroutine cannot explicitly be specified as shared or private, as they are outside the lexical scope of the OpenMP directives. The storage class determines their status:

- automatic variables are created on the private stack of each thread (which is usually small!) and are thus private to threads
- static and external variables (on the heap) are shared, i.e., they refer to the same object in all threads
- dynamic variables can, of course, be allocated independently in all threads, but care must be taken to ensure the program does not leak memory by avoiding to free some of the objects allocated

9.20 Data Scoping Clauses

```
      common /mycommon/ f1, f2
!$omp threadprivate (/mycommon/)
      real a(n), b(n), sum
!$omp parallel shared(a) private(b)
      ...
!$omp end parallel
!$omp parallel default(private) shared(b) reduction(+:sum)
      ...
!$omp  end parallel
```

- `firstprivate, lastprivate`
- `reduction`
- `default (shared|private)`
- `threadprivate`

Notes:

The `threadprivate` directive must be in the declaration section and must follow the declaration of the named `COMMON` blocks it references. Each thread will have a storage block for the named `COMMON` block upon entry into a parallel region. The block is uninitialized unless the `COMMON` block name or variables within the block are referenced in the `copyin` clause. These `COMMON` blocks are private to the thread, but common within the thread. The values within these named `COMMON`s are not preserved across parallel regions, unless dynamic threads are disabled and the number of threads across all parallel regions is the same.

In C/C++ `threadprivate` is used to give a private copy of a file-scope variable to each thread.

9.21 Synchronization

- Barriers

```
!$omp barrier          #pragma omp barrier
```

- Critical sections

```
!$omp critical [printlock] #pragma omp critical [name]
!$omp end critical         { structured block }
```

- Lock library routines

```
omp_set_lock(var)
omp_unset_lock(var)
...
```

Notes:

All threads in a team will wait for each other at the `barrier` directive.

Threads wait at the beginning of a critical region until no other team members are in the region. The `critical` directive has an optional name. All unnamed critical regions map to the same name.

In C/C++ the `#pragma omp barrier` must be in a structured block; it is not a statement.

```
if ( x != 0)
#pragma omp barrier
```

is an illegal structure.

9.22 Synchronization (continued)

- Atomic

```
!$omp atomic              #pragma omp atomic
count = count + 1         count += 1;
```

- Flush

```
!$omp flush [(list}]      #pragma omp flush [(list)]
```

  - Maintain memory consistency
  - Restore/reload thread-visible variables to/from memory
  - Useful for custom synchronization between threads

Notes:

In C/C++ the binary operator for the atomic update must be one of the following: `+, *, -, /, &, ^, |, <<, >>,` and must not be overloaded.

In Fortran the operator can be: `+, * ,-, /, .AND., .OR., .EQV. or .NEQV.,` or the *lvalue* can be involved in one of the following intrinsics: `MAX, MIN, IAND, IOR, IXOR.`

There are implicit flushes (with no list) at a barrier, at entry and exit of a critical region, at the exit of worksharing constructs (unless `nowait` is specified), and at the exit of a parallel region. Volatile storage class variables need not be flushed, since by definition they cause the necessary flushes to be carried out.

Considering the different synchronization mechanisms, the more “heavyweight” mechanisms like barriers are the most expensive, but they do offer the advantage that they are less likely to yield correctness problems; synchronization using synchronization variables or locks can be much faster, but it is more difficult to ensure that problems such as deadlocks never occur.

9.23 Synchronization: DO/FOR ordered

```
!$omp do ordered          #pragma omp for ordered
      do i = ...            for (...)
      ...                   { ...
!$omp ordered             #pragma omp ordered
      print *,i,result(i)     { structured block }
!$omp end ordered             ...
      ...                     }
      enddo
!$omp end do
```

Notes:

The output from the PRINT statement will be the same when run in parallel or serial. This directive could be very expensive.

The DO/FOR directive must contain the ordered clause, the loop can only contain one ordered clause, and the clause can only be executed once with a given iteration of the loop.

9.24 Interoperability

OpenMP directives work with

- Automatic parallelization (`-parallel`)
- MPI
- SHMEM

On the Intel® compilers running in SGI Altix systems OpenMP directives are enabled with `-openmp` flag.

9.25 Run-Time Library Routines

A sample of the OpenMp library routines:

```
OMP_GET_NUM_THREADS
OMP_GET_THREAD_NUM
OMP_IN_PARALLEL
OMP_SET_NUM_THREADS
OMP_SET_DYNAMIC
OMP_SET_NESTED
```

- In C/C++ use

  ```
  #include <omp.h>
  ```

Notes:

These library routines can be used to query or override the environmental variables that control parallel execution.

9.26 OpenMP Environment Variables

The following variables are defined by the standard:

`OMP_SCHEDULE` (default `static`)
`OMP_NUM_THREADS` (default: number of processors in system)
`OMP_DYNAMIC` (default: `.FALSE.`)
`OMP_NESTED` (default: `.FALSE.`)

The following environment variables are Intel® compiler-specific:

`KMP_LIBRARY` (`serial`, `turnaround` or `throughput`)
`KMP_STACKSIZE` (default 4 MB)

See the Intel® Fortran Compiler User's Guide for details.

Notes:

These environmental variables help control the behavior of the parallel code.

9.27 OpenMP on the SGI Altix 3000

- Supported by the Intel® compilers, but not by GNU compilers
- For more information on OpenMP see www.openmp.org
- For more information on OpenMP support in the Intel® compilers, see the Intel® C++ Compiler User's Guide, the Intel® Fortran Compiler User's Guide, and the KAP/Pro Toolset Reference Manual Version 4.0
- The latest Intel® compiler User Guides and Reference Manuals can be found on *http://developer.intel.com/*

Lab: Multiprocessing on the SGI Altix systems

Change to the *Altix/OpenMP/lab* directory.

1. Run the code `mat_dist.f` to get a single processor timing.
2. Add OpenMP directives to the source code and recompile.
3. Run the code in 2, 4, and 8 processors. Use `time` to determine changes in elapsed times for the various runs.

Module 10

Compiling for Shared-Memory Parallelism

10.1 Module Objectives

- Introduce features for compiling for shared-memory parallelism
- Introduce methods of breaking data dependencies
- Introduce parallelization performance issues
- Use the compilers to parallelize serial code

10.2 Compiling for Shared-Memory Parallelism

- The Intel® C/C++, and Fortran compilers can generate parallel code
  - Enabled by using `-openmp` or `-parallel` options when compiling
  - Interprets directives for parallelization
  - Introduces parallelism into code without jeopardizing serial execution
- Directives are ignored when `-openmp` is not specified
- Generation of parallel code for loops is attempted when `-parallel` is used
- When both `-openmp` and `-parallel` are used no automatic parallelization is attempted in routines that already have OpenMP directives
- The `-openmp` option in Fortran sets the `-auto` option to ensure stack allocation of all local variables

10.3 Identifying Parallel Opportunities in Existing Code

- Loops with potential for parallelism
  - Loops without data dependencies
  - Loops with data dependencies because of
    - Temporary variables
    - Reductions
    - Nested loops
    - Function calls or subroutines
- Loops without potential for parallelism
  - Premature exit
  - Too few iterations
  - Programming effort to avoid data dependencies is too great

10.4 Parallelizing Loops Without Data Dependencies

- Simple C loop:

```
for (i = 0; i < max; i++) {
   a[i] = b[i] + c[i];
}
```

- Simple Fortran loop:

```
do i = 1, max
a(i) = b(i) + c(i)
enddo
```

- Variable types that are not specified default to `shared`

10.5 Parallelizing Loops Without Data Dependencies (continued)

- Simple C loop can be transformed into

```
#pragma omp parallel for \
        shared(a, b, c, max) private (i)
   for (i = 0; i < max; i++) {
      a[i] = b[i] + c[i];
   }
```

  - Loop index must be `private`

- Simple Fortran loop can be transformed into

```
c$omp parallel do
c$omp& shared(a, b, c, max), lastprivate(i)
      do i = 1, max
      a(i) = b(i) + c(i)
      enddo
```

  - Loop index must be `private` or `lastprivate`

10.6 Parallelizing Loops With Temporary Variables

- Temporary variables can create data dependencies if they are `shared` variables

```
for (i = 0; i < n; i++) {
   tmp = a[i];
   a[i] = b[i];
   b[i] = tmp;
}
```

- Use `lastprivate` if variable can be treated as `private`, but is to be used after loop
  - By default, index variables assumed to be `private`

```
c$omp parallel do lastprivate(i, x)
c$omp&            shared(a, b, n)
      do i = 1, n
         x = a(i) + b(i)
      enddo
      print(*) i, x
```

10.7 Parallelizing Loops With Temporary Variables (continued)

- Make variables `private` whenever possible

```
#pragma omp parallel for \
        shared(a, b, n) private (i, tmp)
    for (i = 0; i < n; i++) {
       tmp = a[i];
       a[i] = b[i];
       b[i] = tmp;
    }
```

10.8 Parallelizing Reductions

- Generating a single value from the elements of an array is referred to as a reduction
    - There is a data dependence because the same memory location is written to over multiple loop iterations
- Use the `reduction` clause in the corresponding directive or pragma

10.9 Parallelizing Nested Loops

- Nested loops provide options as to which loop to parallelize
- Reorder the loops to
  - Put the most amount of work in each iteration
  - Remove data dependence

  Data dependence due to `a[i][j]` (parallelizing the "`k`" loop):

```
for (k=0; k<n; k++)
   for (j=0; j<n; j++)
      for (i=0; i<n; i++)
         a[i][j] = a[i][j] + b[i][k];
```

  Reordering the loops removes the data dependence (parallelizing the "`i`" loop):

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      for (k=0; k<n; k++)
         a[i][j] = a[i][j] + b[i][k];
```

10.10 Parallelizing Loops With Subroutines and Functions

- Make sure called routines have no side effects
  - Modifies or uses only shared variables indexed by loop control variable or in a critical region
  - Must not use static variables
  - Each call from a thread must be independent of any call from another thread
- Simple test for side effects
  - If the function could be transformed into inlined code, it is probably safe
- Parallel-safe functions
  - All Fortran intrinsic functions
  - Functions in the C standard library
  - All functions in the math library

10.11 Unparallelizable Loops

- Recurrence: loop iteration order not changeable

```
DO I = 2,N
   X(I) = X(I-1) + Y(I)
ENDDO
```

- Exit branch

```
DO I = 2,N
   ! some work
   IF ( VALUE .GT. MAX ) GOTO 999
ENDDO
```

- Loops with too few iterations (not worth parallelizing)

```
DO I = 1,4
    X(I) = 1
ENDDO
```

- Loops with calls to functions with side effects

```
DO I = 2,N
   ! some work
   call unsafe_function
ENDDO
```

10.12 Reducing Parallelization Overhead

- Measuring parallelization overhead
  - Time serial run
  - Calculate one thread of parallel program
    - Time parallel program running single threaded
  - Compare single-thread time to original serial run
- Only parallelize a region if the performance gain is more than the overhead
- Use conditional parallelization modifier to control when a region is parallelized

10.13 Conditional Parallelization

- Can conditionally parallelize a region based on how much work it does
- Most often used with loops
- C/C++

```
#pragma omp parallel for private(i) if (max > 50)
   for (i = 0; i < max; i++) {
      /* loop body */
   }
```

- Fortran

```
C$omp parallel do private(i) shared(n) if (n .GT. 50)
      do i = 1, n
           ! work
      enddo
```

10.14 Load Balancing

- Keep all threads busy
- Find load imbalances
- Balance work done in independent blocks
- Examine loops to determine the best scheduling type for loops

10.15 Process Spin Time

- Processes spin when waiting for more work
  - Ready to get new work immediately
  - Wastes CPU time if no new work is available
- Set the amount of time to spin before blocking
- Reduce spin time if parallel regions are isolated or widely spaced
- Increase spin time for parallel regions that are clustered in one portion of the program

10.16 Guidelines for Parallelization

- Profile serial code
  - Determine regions in which time is significant
- Look for opportunities in significant regions
  - Introduce parallel loops
  - Identify potential independent blocks
  - Rearrange code for best scheduling
- Modify dependent code blocks
  - Identify suitable code blocks
  - Isolate dependence in critical section or single-processor blocks
- Guard against dependence
  - Synchronize

10.17 Automatic Parallelization Limitations

- C/C++
  - Only analyzes certain `for` loops
    - `for` loops using explicit array notation: `array[index]`
    - `for` loops using pointer increment notation: `*var++`
  - Cannot analyze `for` loops using pointer arithmetic notation: `*(var + index)`
  - Cannot analyze `while` or `do/while` loops
  - Does not look for blocks of code to be run in parallel
- Fortran
  - Only analyzes `DO` loops

10.18 Example: Fortran

```
% cat apo-example.f
      program listing
      real*4 a(0:9999), b(0:9999), total
      total = 0.0
      do i = 0, 9999
         b(i) = i
      enddo
      do i=0, 9999
         a(i) = b(i) + 1.0
      enddo
      do i=0, 9999
         a(i) = safety_unknown(a,i)
      enddo
      do i=0, 9999
         total = total + a(i)
      enddo
      write(*,*) total
      end
% efc -parallel -par_report3 -c apo-example.f
   program LISTING
   procedure: listing
   serial loop: line 13: not a parallel candidate due to statement at line 14
apo-example.f(5) : (col. 0) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 5
      shared: {"b"}
      private: {"i"}
      first private: { }
      reductions: { }
apo-example.f(9) : (col. 0) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 9
      shared: {"b", "a"}
      private: {"i"}
      first private: { }
      reductions: { }
apo-example.f(17) : (col. 0) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 17
      shared: {"a"}
      private: {"i"}
      first private: { }
      reductions: {"total"}
22 Lines Compiled
```

10.19 Example: C

```
% cat c-apo-example.c
main() {
        float a[10000], b[10000], total=0.0;
        int i;
        for (i=0; i <= 9999; i++)
                b[i] = i;
        for (i=0; i <= 9999; i++)
                a[i] = b[i] + 1.0;
        for (i=0; i <= 9999; i++)
                a[i] = safety_unknown(a,i);
        for (i=0; i <= 9999; i++)
                total = total + a[i-1];
        printf("%d\n", total);
}
% ecc -w -parallel -par_report3 -c c-apo-example.c
   procedure: main
   serial loop: line 13: not a parallel candidate due to statement at line 13
c-apo-example.c(6) : (col. 17) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 6
      shared: {"b"}
      private: {"i"}
      first private: { }
      reductions: { }
c-apo-example.c(9) : (col. 17) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 9
      shared: {"b", "a"}
      private: {"i"}
      first private: { }
      reductions: { }
c-apo-example.c(17) : (col. 17) remark: LOOP WAS AUTO-PARALLELIZED.
   parallel loop: line 17
      shared: {"a"}
      private: {"i"}
      first private: { }
      reductions: {"total"}
```

10.20 Strategy for Using `-parallel`

- Profile code to determine areas of most performance gain
- Look at output of `-par_report` for more parallelization opportunities
  - The auto-parallelizer may need more information
- Insert directives or enable compiler options to help the auto-parallelizer
  - Information to parallelize/optimize more
  - Indicate which loops are unimportant
- Repeat the above steps as many times as needed

10.21 Controlling the Analysis

- Directives to control parallelization:
  - Fortran

```
!DIR$ PARALLEL
CDIR$ PARALLEL
!DIR$ NOPARALLEL
CDIR$ NOPARALLEL
```

  - C/C++

```
#pragma ...
```

10.22 Debugging With `-parallel`

- Debug first running single-threaded

  ```
  % setenv OMP_NUM_THREADS 1
  ```

- Using debuggers on transformed code:
  - Compile with `-g`

    ```
    % efc -g -O0 -parallel file.f
    ```

- Avoid transformations that confuse the debugger (inlining and loop unrolling)
- Parallel regions, loops and sections get transformed into functions
  - __<*subprogram_name*>_<*line_no*>__par_region<*seq_no*>
  - __<*subprogram_name*>_<*line_no*>__par_loop<*seq_no*>
  - __<*subprogram_name*>_<*line_no*>__par_section<*seq_no*>

Lab: Analysis With `-parallel`

Use `-parallel` to analyze and parallelize a program.

Lab: Parallel Performance

Break data dependencies and check the answers.

Lab: Multiprocessing on the SGI Altix Systems

Change to the *Altix/Multiprocessing/labs/[fc]src* directory.

1. Run the `mat_dist.[fc]` code to get a single processor timing.
2. Add multiprocessing directives to the source code and recompile.
3. Run the code in 2, 4, and 8 processors.
4. Run the `mat_reduce.[fc]` code to get a single processor timing.
5. Add multiprocessing directives to the source code and recompile.
6. Run the code in 2, 4, and 8 processors.

Module 11

Message Passing Interface

11.1 Module Objectives

After completing this module, you will be able to

- Define MPI
- Describe why message passing is a viable parallel programming paradigm
- Explain why MPI is a popular message passing library
- Identify common MPI components
- Write simple parallel programs using MPI calls

11.2 Message Passing

- Explicit parallel programming
  - Programmer inserts communication calls into the program “manually”
  - All processors execute all the code
- Based on “message” transmittal
  - Message consists of status and, usually, data
- Offers point-to-point (process-to-process) or global (broadcast) messages
- Requires a sender and a receiver
  - Processes do not have direct access to each other’s memory

11.3 Why Message Passing?

- Only way to program parallel applications for non-shared memory systems
- Gives programmer 100% control about how to divide the problem
- Can perform better than implicit methods
- Portable — does not require a shared memory machine

11.4 What Is MPI?

- The de-facto standard message passing library
  - Similar functionality to PVM and other libraries
- Goals
  - Provide source-code portability
  - Allow efficient implementation
  - Functionality
- Callable from Fortran, C, C++
  - MPI 1.2 has 129 routines plus 13 “deprecated” ones (big!)
  - MPI-2 adds 157 routines (bigger!)
  - Subset of 6 is enough to do basic communications (small!)

11.5 MPI Header Files and Functions

- Header file
  - Fortran

    ```
    INCLUDE 'mpif.h'
    ```

  - C/C++

    ```
    #include <mpi.h>
    ```

- Function format
  - Fortran

    ```
    CALL MPI_xxx (...., ISTAT)
    ```

  - C/C++

    ```
    int stat = MPI_Xxx (....);
    ```

11.6 MPI Startup and Shutdown

- MPI initialization
  - Fortran

    ```
    CALL MPI_INIT (istat)
    ```

  - C/C++

    ```
    MPI_Init (int *argc, char ***argv);
    ```

  - Must be called *before* any other MPI calls
- MPI termination
  - Fortran

    ```
    CALL MPI_FINALIZE (istat)
    ```

  - C/C++

    ```
    int MPI_Finalize (void);
    ```

  - Must be called *after* all other MPI calls

11.7 Communicator and Rank

- Communicator
  - Group of processes, either system or user defined
  - Default communicator is `MPI_COMM_WORLD`
  - Use the function `MPI_COMM_SIZE` to determine how many processes are in the communicator
- Rank
  - Process number (zero based) within the communicator
  - Use the function `MPI_COMM_RANK` to determine which process is currently executing

11.8 Compiling MPI Programs

```
ecc prog.c -lmpi
```

```
ecc prog.C -lmpi
```

```
efc prog.f -lmpi
```

11.9 Launching MPI Programs

- On most machines, the `mpirun` command launches MPI applications:

  `mpirun -np` *num_Procs user_executable* [*user_args*]

- Example: Launching a program to run with five processes on one computer

  % **`mpirun -np 5 ./a.out`**

- Example: Launching a program to run with 64 processes on each of two systems

  % **`mpirun host1,host2 64 ./a.out`**

11.10 Example: simple1_mpi.c

```
#include <mpi.h>
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int num_procs;
   int my_proc;

/* Initialize MPI */
   MPI_Init(&argc, &argv);
/* Determine the size of the communicator */
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
/* Determine processor number */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_proc);

   if (my_proc == 0)
     printf("I am process %d. Total number of \
            processes: %d\n", my_proc,num_procs);

/* Terminate MPI */
   MPI_Finalize();
}
% ecc simple1_mpi.c -lmpi
% mpirun -np 5 ./a.out
I am process 0. Total number of processes: 5
```

11.11 Example: simple1_mpi.f

```
      program simple1
      include 'mpif.h'
C Initialize MPI
      call mpi_init(istat)
C Determine the size of the communicator
      call mpi_comm_size(mpi_comm_world, num_procs,
     &                   ierr)
C Determine processor number
      call mpi_comm_rank(mpi_comm_world, my_proc, jerr)
      if (my_proc .eq. 0)
     &   write(6,1) 'I am process ',myproc,
     &   '. Total number of processes: ',num_procs
1        format(a,i1,a,i1)
C Terminate MPI
      call mpi_finalize(ierr)
      end
% efc simple1_mpi.f -lmpi
% mpirun -np 5 ./a.out
I am process 0. Total number of processes: 5
```

11.12 MPI Basic (Blocking) Send Format

- C/C++ synopsis

```
int MPI_Send (void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

- Fortran synopsis

```
CALL MPI_SEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM, ISTAT)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, ISTAT
```

- Standard allows for implementation to choose buffering scheme
- `buf` contains the array of data to be sent
- `MPI_Datatype` is one of several predefined types or a derived (user-defined) type

11.13 MPI Basic (Blocking) Receive Format

- C/C++ synopsis

```
int MPI_Recv (void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

- Fortran synopsis

```
CALL MPI_RECV (BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
               STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS (MPI_STATUS_SIZE)
```

- `MPI_ANY_SOURCE` and `MPI_ANY_TAG` can be put in as wildcards when exact source/tag is not known or is not critical to the application
- `buf` contains the array of data to be received
- `MPI_Datatype` is one of several pre-defined types or a derived (user-defined) type

11.14 Elementary Data Types

MPI	Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

MPI	C/C++
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

11.15 Example: simple2_mpi.c

```
#include <mpi.h>
#include <stdio.h>
#define N 1000
main(argc, argv)
int argc;
char *argv[];
{
   int num_procs;
   int my_proc;
   int init, size, rank, send, recv, final;
   int i, j, other_proc, flag = 1;
   double sbuf[N], rbuf[N];
   MPI_Status recv_status;

/* Initialize MPI */
   if ((init = MPI_Init(&argc, &argv)) != MPI_SUCCESS) {
      printf("bad init\n");
      exit(-1);
   }
/* Determine the size of the communicator */
   if ((size = MPI_Comm_size(MPI_COMM_WORLD, &num_procs))
               != MPI_SUCCESS) {
      printf("bad size\n");
      exit(-1);
   }
/* Make sure we run with only 2 processes */
   if (num_procs != 2) {
      printf("must run with 2 processes\n");
      exit(-1);
   }
/* Determine process number */
   if ((rank = MPI_Comm_rank(MPI_COMM_WORLD, &my_proc))
               != MPI_SUCCESS) {
      printf("bad rank\n");
```

```
      exit(-1);
   }
   if (my_proc == 0) other_proc = 1;
   if (my_proc == 1) other_proc = 0;

   for (i = 0; i < N; i++)
      sbuf[i] = i;
/* Both processes send and receive data */
   if (my_proc == 0) {
      if ((send = MPI_Send(sbuf, N, MPI_DOUBLE, other_proc, 99,
                     MPI_COMM_WORLD)) != MPI_SUCCESS) {
         printf("bad send on %d\n",my_proc);
         exit(-1);
      }
      if ((recv = MPI_Recv(rbuf, N, MPI_DOUBLE, other_proc, 98,
                     MPI_COMM_WORLD, &recv_status))
                  != MPI_SUCCESS) {
         printf("bad recv on %d\n", my_proc);
         exit(-1);
      }
   } else if (my_proc == 1) {
      if ((recv = MPI_Recv(rbuf, N, MPI_DOUBLE, other_proc, 99,
                     MPI_COMM_WORLD, &recv_status))
                  != MPI_SUCCESS) {
         printf("bad recv on %d\n", my_proc);
         exit(-1);
      }
      if ((send = MPI_Send(sbuf, N, MPI_DOUBLE, other_proc, 98,
                     MPI_COMM_WORLD)) != MPI_SUCCESS) {
         printf("bad send on %d\n",my_proc);
         exit(-1);
      }
   }
/* Terminate MPI */
   if ((final = MPI_Finalize()) != MPI_SUCCESS) {
      printf("bad finalize \n");
```

```
      exit(-1);
   }
/* Making sure clean data has been transferred */
   for(j = 0; j < N; j++) {
      if (rbuf[j] != sbuf[j]) {
         flag = 0;
         printf("processor %d: rbuf[%d]=%f. Should be %f\n",
                my_proc, j, rbuf[j], sbuf[j]);
      }
   }
   if (flag == 1) printf("Test passed on processor %d\n",
                         my_proc);
   else printf("Test failed on processor %d\n", my_proc);
}
% ecc -w simple2_mpi.c -lmpi
% mpirun -np 2 ./a.out
Test passed on process 1
Test passed on process 0
```

11.16 Example: simple2_mpi.f

```
 program two_procs
 include 'mpif.h'
 parameter (n=1000)
 integer other_proc
 integer send, recv
 integer status(mpi_status_size)
 dimension sbuf(n), rbuf(n)

 call mpi_init(init)
 if (init .ne. mpi_success) stop 'bad init'
 call mpi_comm_size(mpi_comm_world, num_procs, ierr)
 if (num_procs .ne. 2) stop 'npes not 2'
 if (ierr .ne. mpi_success) stop 'bad size'
 call mpi_comm_rank(mpi_comm_world, my_proc, jerr)
 if (jerr .ne. mpi_success) stop 'bad rank'
 if (my_proc .eq. 0) other_proc = 1
 if (my_proc .eq. 1) other_proc = 0

 do i = 1, n
 sbuf(i) = i
 enddo

 if (my_proc .eq. 0) then
   call mpi_send(sbuf, n, mpi_real, other_proc, 99,
&                mpi_comm_world, send)
   if (send .ne. mpi_success) stop 'bad 0 send'
   call mpi_recv(rbuf, n, mpi_real, other_proc, 98,
&                mpi_comm_world, status, recv)
   if (recv .ne. mpi_success) stop 'bad 0 recv'
 else if (my_proc .eq. 1) then
   call mpi_recv(rbuf, n, mpi_real, other_proc, 99,
&                mpi_comm_world, status, recv)
   if (recv .ne. mpi_success) stop 'bad 1 recv'
   call mpi_send(sbuf, n, mpi_real, other_proc, 98,
```

```
     &                mpi_comm_world, send)
        if (send .ne. mpi_success) stop 'bad 1 send'
       endif

       call mpi_finalize(ierr)
       if (ierr .ne. mpi_success)stop 'bad final'
       iflag = 1
       do j = 1, n
         if (rbuf(j) .ne. sbuf(j)) then
           iflag = 0
           print*,'process ', my_proc, ':rbuf(', j, ')=',
     &           rbuf(j),'.Should be ',sbuf(j)
         endif
       enddo

       if (iflag .eq. 1) then
         print*,'Test passed on process ',my_proc
       else
         print*,'Test failed on process ',my_proc
       endif

       end
% efc -w simple2_mpi.f -lmpi
% mpirun -np 2 ./a.out
Test passed on process 0
Test passed on process 1
```

11.17 Additional MPI Messaging Routines

- Buffered messages

```
MPI_Bsend(buf, count, datatype, dest, tag, comm)
```

- Asynchronous messages

```
MPI_Isend (buf, length, data_type, destination,
           message_tag, communicator, &request)
MPI_Ibsend(buf, count, datatype, dest, tag, comm, &request)
```

- Return receipt messages

```
MPI_Ssend(buf, count, datatype, dest, tag, comm)
MPI_Issend(buf, count, datatype, dest, tag, comm, &request)
```

Notes:

When using buffered sends, the user must provide a usable buffer for MPI using an `MPI_Buffer_attach` command. Additional MPI calls are available to manage these buffers.

When using the asynchronous MPI calls, a handle is returned to the user. The user cannot modify/delete “data” until the message is completed or freed. See the next slide for checking the status of requests.

11.18 MPI Asynchronous Messaging Completion

`MPI_Wait(request, status)`

- Wait until request is completed

`MPI_Test(request, flag, status)`

- Logical flag indicates whether request has completed

`MPI_Request_free(request)`

- Removes request

Asynchronous Message Receipt

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request)
MPI_Iprobe(source, tag, comm, flag, status)
```

- Checks for messages without blocking
- Probe will check for messages without receiving them

11.19 Commonly Used MPI Features

- Point-to-point messages
- Collective operations
  - Broadcast
    - For example, one task reads in a data item and wants to send to all other tasks
  - Global reductions
    - Sums, products, minimums, maximums
- Derived data types
  - Necessary for noncontiguous patterns of data
- Functions to assist with topology grids
  - Convenience—no performance advantage

11.20 Collective Routines

- Called by all processes in the group
- Examples
  - Broadcast
  - Gather
  - Scatter
  - All-to-all broadcast
  - Global reduction operations (such as sums, products, max, and min)
  - Scan (such as partial sums)
  - Barrier synchronization

11.21 Synchronization

- Format
  - C/C++ synopsis

    ```
    int MPI_Barrier(MPI_Comm comm)
    ```

  - Fortran synopsis

    ```
    CALL MPI_BARRIER (COMM, ISTAT)
    INTEGER COMM, ISTAT
    ```

- Blocks the calling process until all processes have made the call
  - Ensures synchronization for time-dependent computations
  - Most commonly used synchronization routine

11.22 Broadcast

- Format
- C/C++ synopsis

```
int MPI_Bcast(void* buf, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

- Fortran synopsis

```
CALL MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT, COMM, ISTAT)
<TYPE> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, ISTAT
```

- Broadcasts a message from `root` to all processes in the group, including itself

11.23 Example: bcast.c

```
#include <mpi.h>
#include <stdio.h>
#define N 5
main(argc, argv)
int argc;
char *argv[];
{
   int num_procs, my_proc;
   int a[N];
   int i, j, root=0;
   for(i = 0; i < N; i++)
      a[i] = -11;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_proc);
   if (my_proc == root) {
      for(i=0;i<N;i++)
         a[i] = -20;
   }
   MPI_Bcast((void *)a, N, MPI_INT, root, MPI_COMM_WORLD);
   for (j = 0; j < N; j++)
      printf("%d ",a[j]);
   printf("\n");
   MPI_Finalize();
}
% ecc -w bcast.c -lmpi
% mpirun -np 7 ./a.out
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
-20 -20 -20 -20 -20
```

11.24 Example: bcast.f

```
      program cast
      include 'mpif.h'
      parameter (n=5)
      dimension buf(n)
      integer root
      parameter (root = 0)

      do i = 1, n
        buf(i) = -11.0
      enddo
      call mpi_init(ierr)
      call mpi_comm_size(mpi_comm_world, num_procs, ierr)
      call mpi_comm_rank(mpi_comm_world, my_proc, jerr)
      if (my_proc .eq. root) then
        do i=1, n
          buf(i) = -20.0
        enddo
      endif
      call mpi_bcast(buf,n,mpi_real,root,mpi_comm_world,
     &               istat)
      write(6,1) (buf(j), j=1,n)
    1 format(5(f5.1,1x))
      call mpi_finalize(ierr)
      end
% efc -w bcast.f -lmpi
% mpirun -np 7 ./a.out
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
-20.0 -20.0 -20.0 -20.0 -20.0
```

11.25 Reduction

- Format
  - C/C++ synopsis

    ```
    int MPI_Reduce (void* sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
    ```

  - Fortran synopsis

    ```
    CALL MPI_REDUCE (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT,
                     COMM, ISTAT)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, ISTAT
    ```

- Combines the elements of `sendbuf` in each process in the group, using the operation `op`, and returns the results in `root` process's `recvbuf`
- MPI provides predefined operations for `op`:

  ```
  MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND
  MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR
  MPI_MAXLOC, MPI_MINLOC
  ```

11.26 Example: reduce.c

```
#include <mpi.h>
#include <stdio.h>
#define N 5
main(int argc, char *argv[])
{
   int num_procs;
   int my_proc;
   int myarr[N];
   int global_res[N];
   int i;

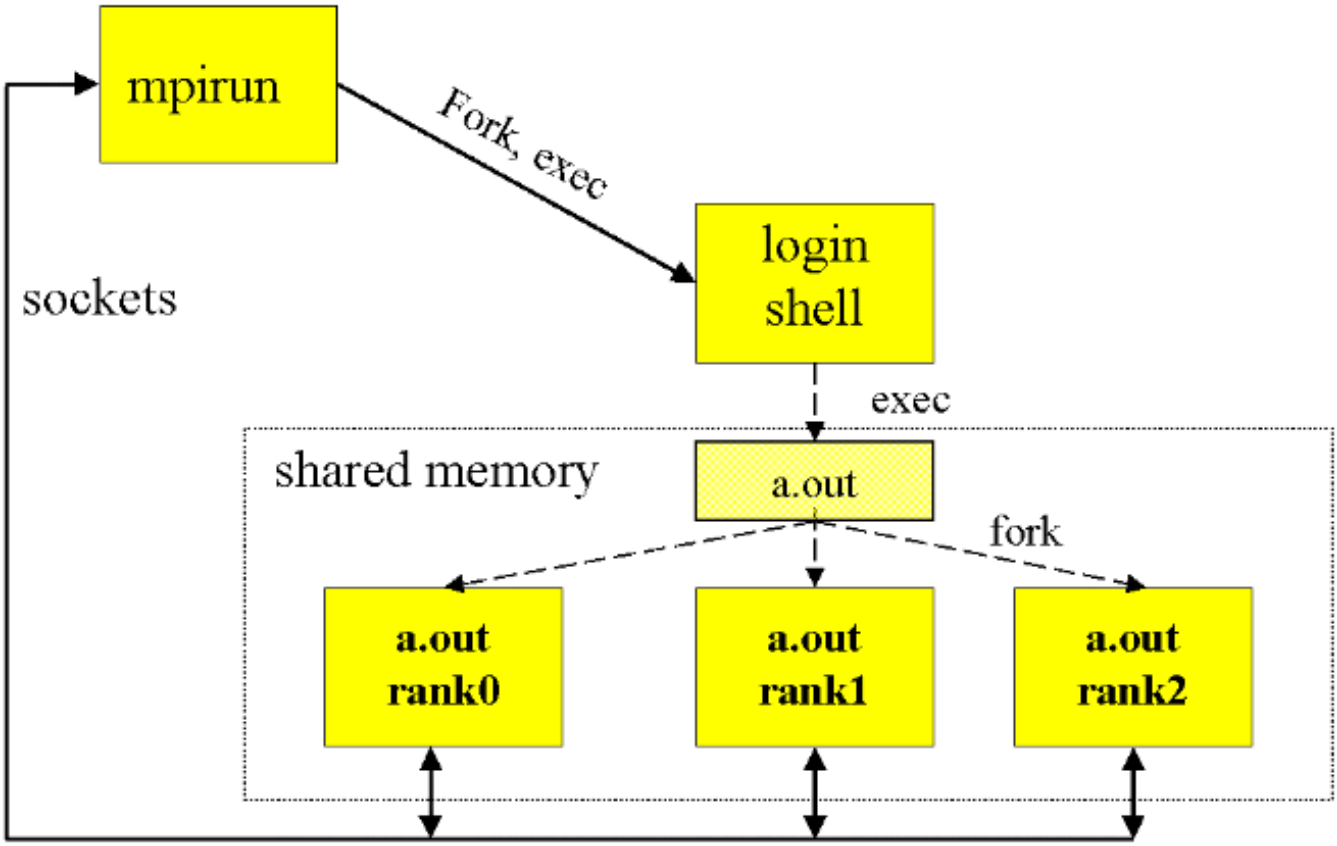
   MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_proc);
   for (i = 0; i< N; i++)
      myarr[i] = my_proc+i+1;
   MPI_Reduce((void *) myarr, (void *) global_res, N, MPI_INT,
              MPI_SUM, 0, MPI_COMM_WORLD);
   if (my_proc == 0) {
      for(i = 0; i< N ; i++)
         printf("%d\n",global_res[i]);
         printf("\n");
   }
   MPI_Finalize();
}
% ecc -w reduce.c -lmpi
% mpirun -np 5 ./a.out
15
20
25
30
35
```

11.27 Example: reduce.f

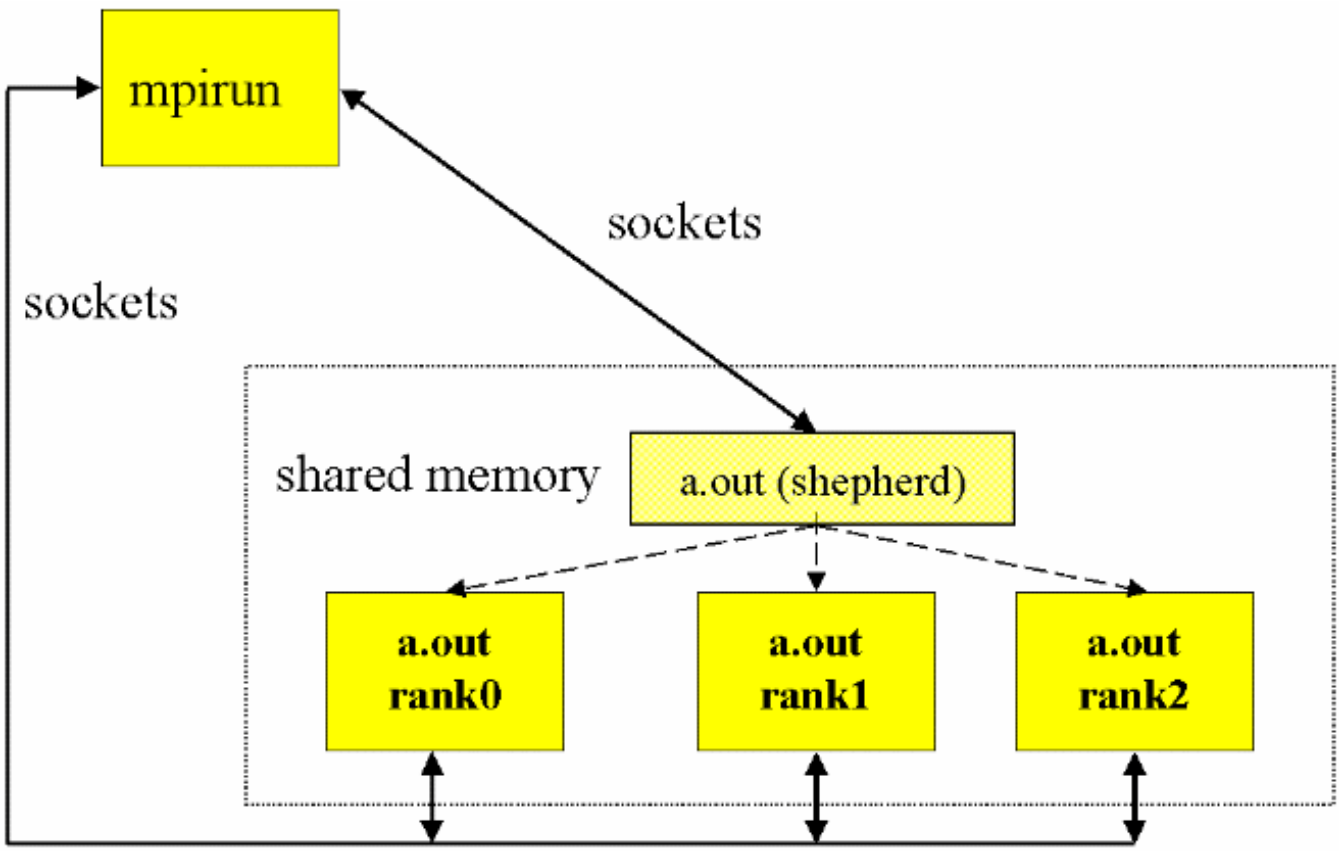
```
      program reduce
      include 'mpif.h'
      parameter (n = 5)
      integer myarr(n)
      integer global(n)

      call MPI_Init(ierr)
      call MPI_Comm_size(MPI_COMM_WORLD, num_procs, ierr)
      call MPI_Comm_rank(MPI_COMM_WORLD, my_proc, jerr)
      do i = 1, n
        myarr(i) = my_proc + i
      enddo
      call MPI_Reduce(myarr, global, n, MPI_INTEGER, MPI_SUM, 0,
     & MPI_COMM_WORLD, ierr)
      if (my_proc .eq. 0)
     &  write(6,1) (global(i), i=1, n)
    1 format(5(i2,1x))
      call MPI_Finalize(ierr)
      end
% efc -w reduce.f -lmpi
% mpirun -np 5 ./a.out
15 20 25 30 35
```

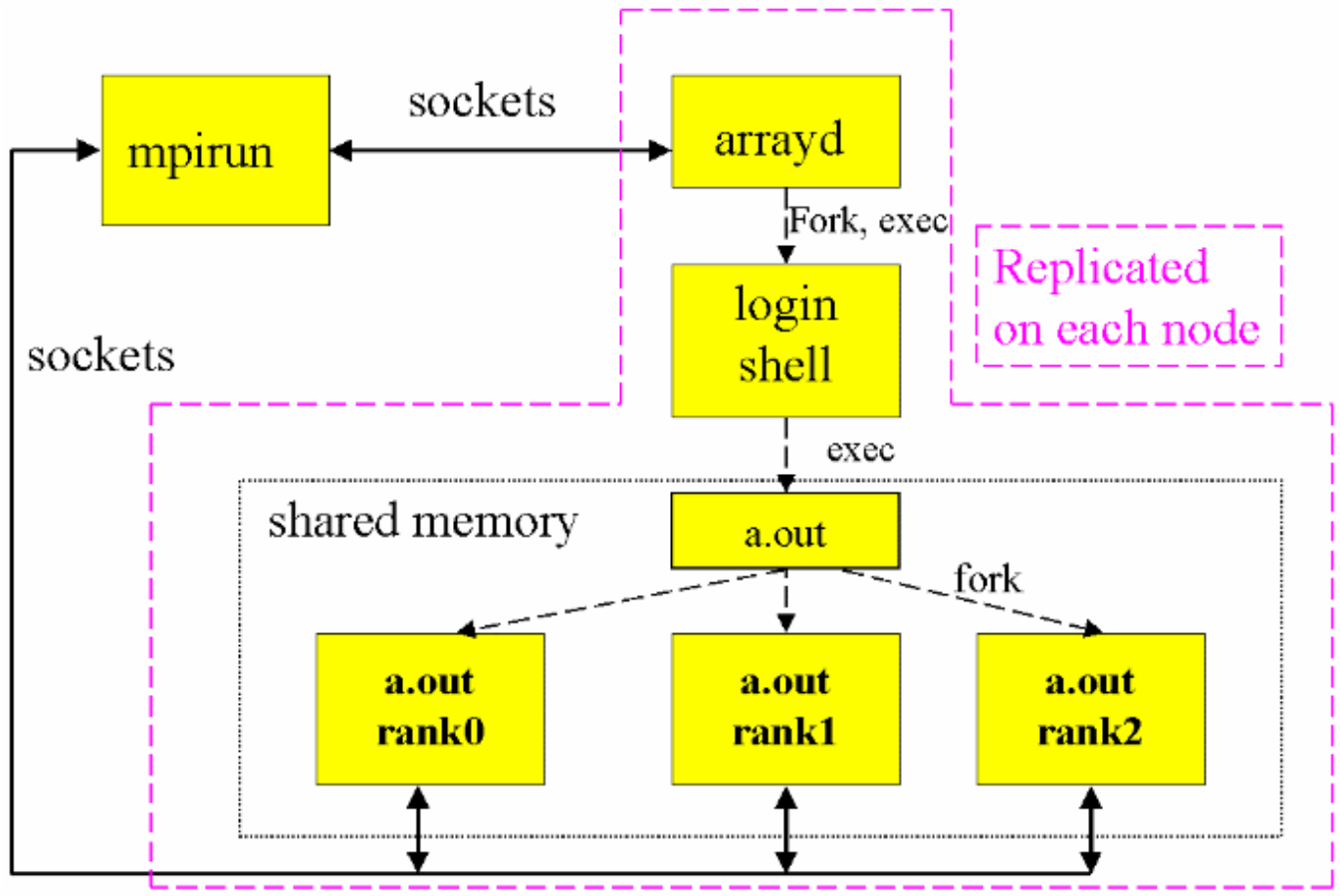
11.28 MPI Application on SGI Altix Systems



11.29 MPI Application After Startup



11.30 MPI Application on SGI Altix Cluster



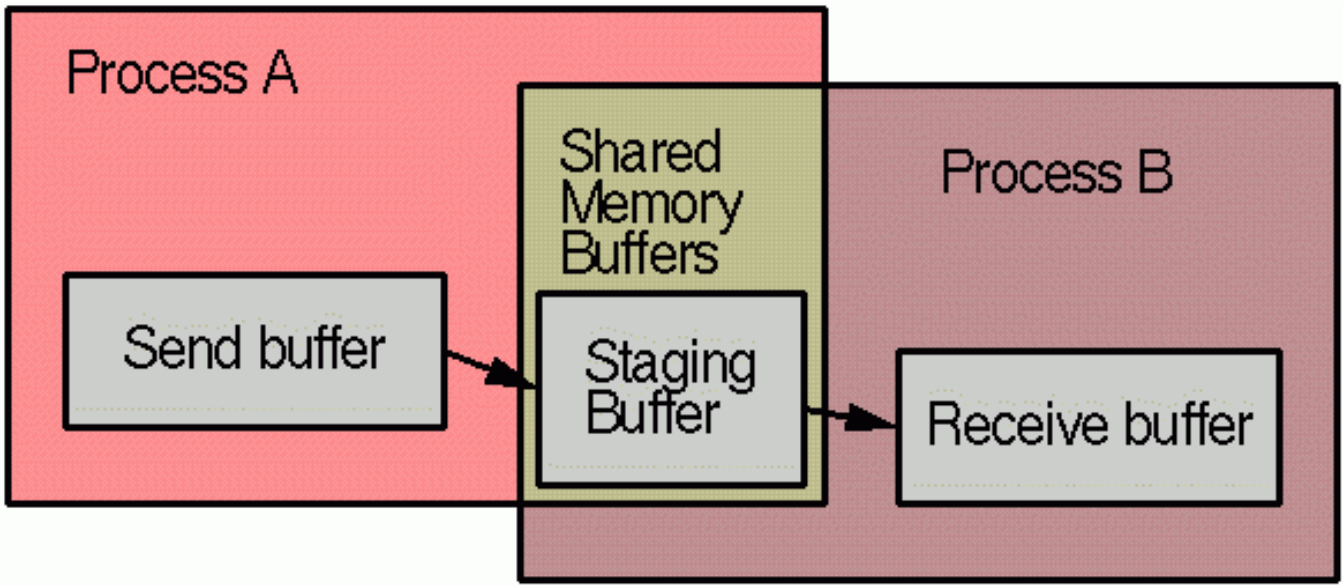
11.31 MPI Process Initiation

- `mpirun` requests the array services daemon, `arrayd`, to create processes
- Login shell is created—creates a single copy of program
- This copy becomes the “shepherd” or MPI daemon
    - Forks the MPI processes, does bookkeeping
    - One daemon per machine in a cluster

11.32 MPI Process Relationships

- Ancestor/descendent relationship is lost (true only on clusters)
- Extra shepherd process is introduced
- Most, but not all, job control functions are retained
  - Cntl C/interrupt works
  - Cntl Z/suspend works
  - `fg` works to resume
  - `bg` is not supported

11.33 MPI Messaging Implementation



11.34 MPI on SGI Altix Clusters

- Can communicate across machines in a supercluster of SGI Altix Systems
- Three interhost communication modes:
  - GSN (formerly called HiPPI 6400)
  - Myrinet
  - NUMAflex (XPMEM)

11.35 NUMA Memory Layout

- Explicit NUMA placement used for static memory and symmetric heap
- “First-touch” is used for heap and stack
- `dplace` may be used, but remember to skip the shepherd process

  ```
  mpirun -np 4 dplace -s1 -c0-3 a.out
  ```

- If cpusets are used, the numbers in the `-c` option refers to logical CPUs within the cpuset

11.36 Cluster Example

- See the */usr/lib/array/arrayd.conf* file

```
array goodarray
      machine fast.sgi.com
      machine faster.sgi.com
      machine another.sgi.com
```

- Sample command to run 128 processes across two of the clustered machines:

```
mpirun -a goodarray fast 64 a.out : faster 64 a.out
```

  - “64” is number of processors

11.37 Standard in/out/err Behavior

- All `stdout/stderr` from MPI tasks directed to `mpirun`
- `stdout` is line buffered
- Sent to `mpirun` as a message
- `stdin` is limited to MPI rank 0
- `stdin` is line buffered
- New line is needed for `mpirun` to process input from `stdin`

11.38 Debugging

- Etnus Totalview

```
totalview mpirun -a -np 4 a.out
```

11.39 Using Performance Tools

- `profile.pl`

  ```
  mpirun -np 4 profile.pl [options] a.out
  ```

11.40 Scheduling With cpusets

- In a time-shared environment, use cpusets to ensure that message passing processes are scheduled together:

  ```
  cpuset -q myqueue -A mpirun -np 100 a.out
  ```

- Dynamic cpuset creation is supported in Platform’s LSF and Altair Engineering’s PBSpro

11.41 MPI Optimization Hints

- Do not use wildcards, except when necessary
- Do not oversubscribe number of processors
- Collective operations are not all optimized
  - Use SHMEM to optimize bottlenecks
- Minimize use of `MPI_barrier` calls
- Optimized paths
  - `MPI_Send() / MPI_Recv()`
  - `MPI_Isend() / MPI_Irecv()`
- Less optimized:
  - `ssend, rsend, bsend, send_init`
- When using `MPI_Isend()/MPI_Irecv()`, be sure to free your request by either calling `MPI_Wait()` or `MPI_Request_free()`

11.42 Environment Variables

- `MPI_DSM_CPULIST`
  - Allows specification of which CPUs to use
  - If running within an n-processor cpuset, use 0-<$n-1$> rather than physical CPU numbers
  - Works like an implicit `dplace -s1`
- `MPI_BUFS_PER_PROC`
  - Number of 16-kB buffers for each processor (default 32)
  - For use within a host; they are assigned locally so copy into buffer is efficient and has no contention
- `MPI_BUFS_PER_HOST`
  - Single pool of buffers for interhost communication, 16 kB each (default 32)
  - Less memory usage but more contention
- Many others, see `man mpi`

11.43 Instrumenting MPI

- MPI has `PMPI*` names

```
int MPI_Send(args)
{
   sendcount++;
   return PMPI_Send(args);
}
```

  - "`MPI_Send`" is user defined send function; "`PMPI_Send`" is the actual `MPI_Send` function in the library

11.44 Message Passing References

- Man pages
  - `mpi`
  - `mpirun`
  - `shmem`
- Release notes
  - `relnotes mpt`
- MPI Standard

  `http://www.mpi-forum.org/docs/docs.html`

11.45 General MPI Issues

- Most programs use blocking calls
- Use of nonblocking and synchronization calls can lead to faster codes
- Synchronization (barrier, wait, and so on) calls are often overused; algorithm rethinking often eliminates the unnecessary calls
- Instead of writing your own procedures, investigate whether MPI has one and use it (for example, reductions, synchronization, broadcast)
- URLs `http://www.mcs.anl.gov/mpi/index.html` and `http://www.mpi-forum.org/docs/docs.html` are good starting points to learn more about the standard

Lab: MPI and Explicit Data Decomposition

For the following lab exercises, make sure the `mpt` module is loaded.

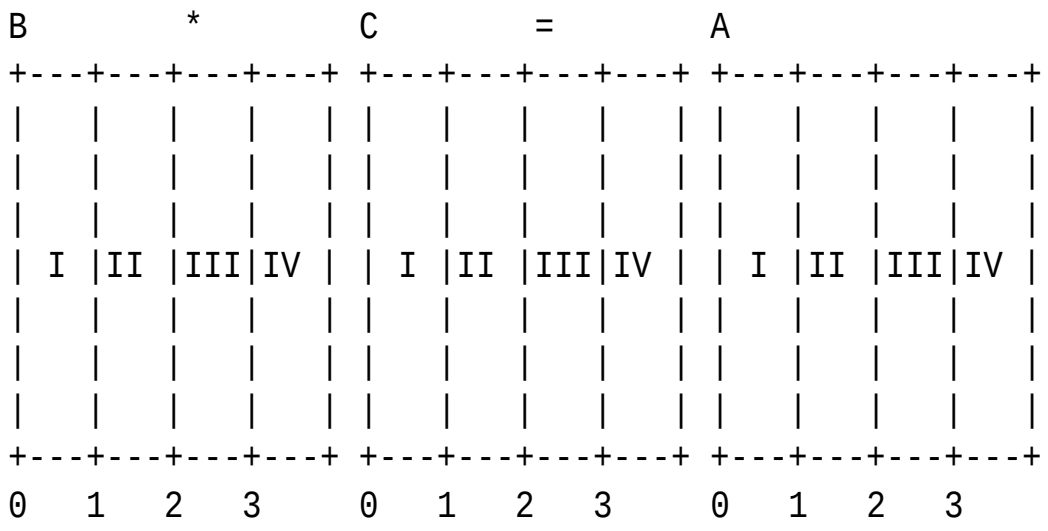
1. Write a program to distribute the work of initializing an array, A, where
   A(i) = i as i = 1 to 1024
   Share the work across four PEs. Have each processor calculate a partial sum for A.
   Use MPI calls to pass all the partial sums to one processor.
   What is the final sum? _____
   How well does your program "scale"? _____

2. Try using an MPI call to do the reduction.
   How many processors now have the final sum?

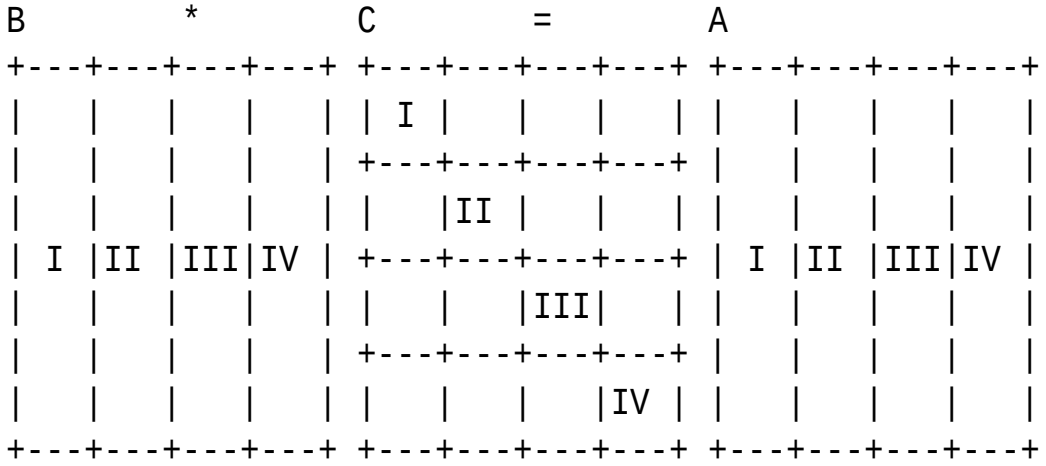
   _____

3. Use another MPI call to get the sum to all the processors.

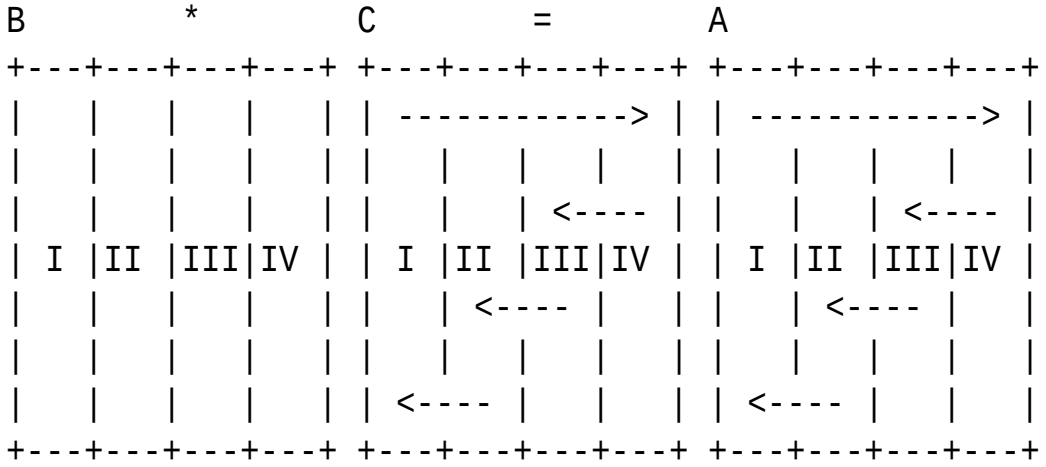
4. Compile and run the program `Altix/MPI/mxm4.mpi.[fc]`. This program must be run on four PEs.
   `mxm4.mpi.[fc]` is a distributed 256×256 matrix multiply, where
   B (i , j) = i - j
   C (i , j) = 2i - j
   The initial data layout is as follows:



The first pass of the outer loop uses:



The first data transfer moves:



How many passes does it take to complete the multiply?

Which pieces of data are used on each pass of the outer loop? How much data is moved to complete the multiply?

---

5. Add calls to time the multiplication phase and the data passing stages of this program.
   How many MFLOPs does each processor achieve in the multiplication phase?

How much is the performance reduced if the overhead of data passing is considered?

---

6. Modify the data passing routines to reduce the data passing time.
   How much does this improve the overall performance of this program?

   What is the trade-off for this improvement?

7. Can you reduce the data passing time further by changing the data decomposition?
   What about replicating some of the data?

Module 12

Shared Memory Access Library (libsma)

12.1 Module Objectives

After completing this module, you will be able to

- Understand one-sided, data-passing concepts
- Use data-passing calls to parallelize applications

12.2 Shared Memory Access Library

- “Global memory access” library developed by Cray Research for the Cray T3D/T3E
- Library routines operate on remote and local memory
- Under Altix, part of Message Passing Toolkit (MPT) package
- Unlike message passing, `shmem` routines do not require the sender-receiver pair
- Minimal overhead and latency and maximum data bandwidth
- Available on all SGI systems

12.3 Shared Memory Access Library (continued)

Supported operations

- Remote data transfer
- Atomic swap
- Atomic increment and add
- Work-shared broadcast and reduction
- Synchronization

12.4 Shmem Parameters

Some `shmem` routines have predefined parameters as argument(s)

- C/C++

  ```
  #include <mpp/shmem.h>
  ```

- Fortran

  ```
  INCLUDE 'mpp/shmem.fh'
  ```

12.5 Data Addresses

- C, C++, or Fortran data objects are passed by address to `SHMEM` routines
- Target or source arrays that reside on the remote PE must be “symmetric”
- `SHMEM` routines that operate on the same data object on multiple PEs require that symmetric data objects be passed (collective routines)
- Symmetric
  - Data object whose local address is exactly the same as a corresponding data object on remote PEs
  - Remote data objects are accessed by using the address of the corresponding data object on the local PE
- The following data objects are symmetric on Altix systems:
  - Fortran PE-private data objects in common blocks or with the SAVE attribute
    - These data objects must not be defined in a DSO
  - Non-stack C and C++ variables
    - These data objects must not be defined in a DSO
  - Objects allocated from the symmetric heap using `shmalloc`, `shpalloc`, or related routines

12.6 Individual Routines

- Called by one PE regardless of how many PEs are allocated to a job
- “Owner” of the remote memory is not involved in the transfer
- User is responsible for any necessary synchronization
- Most commonly used individual routines:
  - `shmem_get`*n*: calling PE gets data from a specified PE
    - Uses simple loads
    - Blocks execution — does not return until data has arrived in local memory
  - `shmem_put`*n*: calling PE puts data to a specified PE
    - Uses simple stores
    - Function returns when last store instruction is issued; data may not have arrived at destination yet
    - Order of arrival in remote memory for multiple calls is not guaranteed

12.7 GET Operations

- C/C++ data transfer

```
void shmem_get<32|64|128> (void *target, const void *source,
                           size_t len, int pe)
```

- Fortran data transfer

```
integer len, pe
call shmem_get<4|8|32|64|128> (target, source, len, pe)
```

- Transfer `len` words from address `source` on process `pe` to address `target` on local processing element
- C/C++ strided data transfer

```
void shmem_iget<32|64|128> (void *target, const void *source,
                            ptrdiff_t tst, ptrdiff_t sst,
                            size_t len, int pe)
```

- Fortran strided data transfer

```
integer tst, sst, len, pe
call shmem_iget<4|8|32|64|128> (target, source, tst, sst,
                                len, pe)
```

- Transfer strided data from a specified processing element (PE)

12.8 PUT Operations

- C/C++ data transfer

```
void shmem_put<32|64|128> (void *target, const void *source,
                           size_t len, int pe)
```

- Fortran data transfer

```
integer len, pe
call shmem_put<4|8|32|64|128> (target, source, len, pe)
```

- Transfer `len` words to address `target` on process `pe` from address `source` on local processing element
- C/C++ strided data transfer

```
void shmem_iput<32|64|128> (void *target, const void *source,
                            ptrdiff_t tst, ptrdiff_t sst,
                            size_t len, int pe)
```

- Fortran strided data transfer

```
integer tst, sst, len, pe
call shmem_iput<4|8|32|64|128> (target, source, tst, sst,
                                len, pe)
```

- Transfer strided data to a specified processing element (PE)

12.9 Get and Put

- `target` Array to be updated
  - For get routines, `target` resides on the local PE. For put routines, `target` resides on the remote PE and must be remotely accessible.
  - Data types are as follows:
    - `shmem_get`*n*, `shmem_iget`*n*, `shmem_put`*n*, `shmem_iput`*n*
      Any non-character type that has a storage size equal to n bytes
- `source` Array to be copied.
  - The `source` argument can have any data type permitted for `target`
  - For put routines, `source` resides on the local PE For get routines, `source` resides on the remote PE and must be remotely accessible
- `len` Number of elements in the target and source arrays
- `pe` PE number of the remote PE

12.10 Starting Up Virtual PEs

- You must start PEs for a data passing (`shmem`) program
- Synopsis:
  - C/C++:

    ```
    void start_pes(int npes)
    ```

  - Fortran:

    ```
    integer npes
    call start_pes(npes)
    ```

- `npes` identifies the total number of PEs desired; however, since `shmem` is layered on the MPI library, the `mpirun` command's option `-np` will determine the number of processes created.
- Routine can be called only once, preferably before any other calls by the main program

12.11 Determining Processing Element Number

- Synopsis
  - C/C++:

    ```
    int _my_pe (void);
    ```

    or

    ```
    int shmem_my_pe(void);
    ```

  - Fortran:

    ```
    MYPE = MY_PE ()
    ```

    or

    ```
    INTEGER SHMEM_MY_PE
    MYPE = SHMEM_MY_PE()
    ```

- These functions return the number of the calling processing element (zero-based)

12.12 Determining the Total Number of PEs

- Synopsis
  - C/C++:

    ```
    int _num_pes (void);
    ```

  - Fortran:

    ```
    NPES = NUM_PES()
    ```

- These functions return the total number of processing elements in the current application

12.13 Compiling and Running SHMEM Programs

- To compile

```
cc prog.c -lsma
g77 prog.f -lsma
ecc prog.c -lsma
efc prog.f -lsma
```

- If using SHMEM and MPI together, add the `-lmpi` loader option
- On the SGI Altix, SHMEM runs on top of MPI, so the use of `mpirun` is necessary:

`mpirun -np` *no_of_processes executable*

12.14 Example: GET (Fortran)

```
      PROGRAM GET ! SAMPLE 2-PE CODE USING SHMEM_GET32
      INTEGER, PARAMETER :: SIZE = 10
      COMMON /BLK/ SRC
      REAL(KIND=4), DIMENSION(SIZE) :: SRC, TARGET
      INTEGER :: OTHER_PE, SHMEM_MY_PE
      CALL START_PES(2) ! start 2 virtual PE's
      IF (SHMEM_MY_PE() == 0) OTHER_PE = 1
      IF (SHMEM_MY_PE() == 1) OTHER_PE = 0
      SRC = SHMEM_MY_PE()
      CALL SHMEM_BARRIER_ALL() ! MAKE SURE EVERYONE HAS
                               ! INITIALIZED SRC
      CALL SHMEM_GET32(TARGET, SRC, SIZE, OTHER_PE)
      WRITE(6,1) 'PE',MY_PE(),': ', (TARGET(I), I=1, SIZE)
    1 FORMAT(A,I2,A,10(F3.1,1X))
      END PROGRAM GET
% efc shmemget32.f90 -lsma
% mpirun -np 2 ./a.out
PE 0: 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
PE 1: 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

12.15 Example: GET (C/C++)

```
#include <stdio.h> /* sample 2 PE code using shmem_get32 */
#define SIZE 10
main()
{
   static float src[SIZE];
   float target[SIZE];
   int other_pe;
   int i;
   start_pes(2); /* start 2 virtual PE's */
   if (shmem_my_pe() == 0) other_pe = 1;
   if (shmem_my_pe() == 1) other_pe = 0;
   for (i = 0; i < SIZE; i++) src[i] = shmem_my_pe();
      shmem_barrier_all(); /* make sure everyone has
                              initialized src */
   shmem_get32(target, src, SIZE, other_pe);
   printf("PE %d: ",_my_pe());
   for (i = 0; i < SIZE; i++) printf("%3.1f ",target[i]);
   puts("\n");
}
% cc shmemget32.c -lsma
% mpirun -np 2 ./a.out
PE 1: 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
PE 0: 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
```

12.16 Completing a Single put Operation

Since `put` is asynchronous, it is the programmer's responsibility to guarantee the arrival (put) of data to the remote PE. This can be done via:

- `shmem_barrier_all`
  (guarantees memory and remote store completions)
- `shmem_wait`
  (event driven wait routine)
- `shmem_wait_until`
  (event driven wait routine)
- `shmem_quiet`
  (guarantees completion of remote stores)
- `shmem_fence`
  (guarantees network quiet and completion of remote stores)

12.17 Example: PUT (Fortran)

```
      PROGRAM PUT ! SAMPLE 2-PE CODE USING SHMEM_PUT32
      INTEGER, PARAMETER :: SIZE = 10
      COMMON /BLK/ TARGET
      REAL(KIND=4), DIMENSION(SIZE) :: SRC, TARGET
      INTEGER :: OTHER_PE, SHMEM_MY_PE
      CALL START_PES(2) !START 2 VIRTUAL PE'S
      IF (SHMEM_MY_PE() == 0) OTHER_PE = 1
      IF (SHMEM_MY_PE() == 1) OTHER_PE = 0
      SRC = SHMEM_MY_PE()
      CALL SHMEM_PUT32(TARGET, SRC, SIZE, OTHER_PE)
      CALL SHMEM_BARRIER_ALL() ! MAKE SURE DATA HAS BEEN
                               ! DELIVERED
      WRITE(6,1) 'PE',MY_PE(),': ', (TARGET(I), I=1, SIZE)
    1 FORMAT(A,I2,A,10(F3.1,1X))
      END PROGRAM PUT
% efc shmemput32.f90 -lsma
% mpirun -np 2 ./a.out
PE 0: 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
PE 1: 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

12.18 Example: PUT (C/C++)

```
#include <stdio.h> /* sample 2 PE code using shmem_put32 */
#define SIZE 10
main()
{
   static float target[SIZE];
   float src[SIZE];
   int other_pe;
   int i;
   start_pes(2); /* start 2 virtual PE's */
   if (shmem_my_pe() == 0) other_pe = 1;
   if (shmem_my_pe() == 1) other_pe = 0;
   for (i = 0; i < SIZE; i++) src[i] = shmem_my_pe();
   shmem_put32(target, src, SIZE, other_pe);
   shmem_barrier_all(); /* make sure data has been delivered */
   printf("PE %d: ",_my_pe());
   for (i = 0; i < SIZE; i++)
      printf("%3.1f ",target[i]);
   puts("\n");
}
% ecc shmemput32.c -lsma
% mpirun -np 2 ./a.out
PE 0: 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
PE 1: 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

12.19 Multiple PUT Calls

`shmem_fence` assures ordering of delivery of puts

- Synopsis
  - C/C++:

    ```
    void shmem_fence(void);
    ```

  - Fortran:

    ```
    CALL SHMEM_FENCE
    ```

- All `put` operations issued to a particular processing element prior to the call to `shmem_fence` are guaranteed to be delivered before any `put` operations to the same processing element that follow the call to `shmem_fence`

12.20 Example: Multiple PUT Calls (Fortran)

```
PROGRAM MULTI_PUT
! SAMPLE 2-PE CODE USING MULTIPLE SHMEM_PUT64 CALLS
INTEGER, PARAMETER :: SIZE = 500000
COMMON /BLK/ TARGET1, IFLAG
REAL(KIND=8), DIMENSION(SIZE) :: SRC, TARGET1
INTEGER :: OTHER_PE, SHMEM_MY_PE
CALL START_PES(2)
IF (SHMEM_MY_PE() == 0) OTHER_PE = 1
IF (SHMEM_MY_PE() == 1) OTHER_PE = 0
DO I = 1, SIZE
   SRC(I) = SIN(FLOAT(I))
ENDDO
IFLAG=999
IF (SHMEM_MY_PE() == 1) THEN
   CALL SHMEM_WAIT (IFLAG, 999)
   ...
ELSE     ! PE 0
   CALL SHMEM_PUT64(TARGET1, SRC, SIZE, OTHER_PE)
   CALL SHMEM_FENCE() ! GUARANTEES TARGET1 IS PUT
   IFLAG = -999
   CALL SHMEM_PUT64(IFLAG, IFLAG, 1, OTHER_PE)
   ...
ENDIF
...
```

12.21 Example: Multiple PUT Calls (C/C++)

```
#include <stdio.h>
/* sample 2 PE code using multiple shmem_put64 calls */
#include <math.h>
#define SIZE 500000
main()
{
   static double, target[SIZE];
   double src[SIZE];
   static long flag;
   int other_pe;
   int i;
   start_pes(2);
   if (shmem_my_pe() == 0) other_pe = 1;
   if (shmem_my_pe() == 1) other_pe = 0;
   for (i = 0; i < SIZE; i++) src[i] = sin((float) i);
   flag = 999;
   if (_my_pe() == 1) {
      shmem_wait (& flag, 999)
      ...
   }
   else {
      shmem_put64(target, src, SIZE, other_pe);
      shmem_fence(); /* guarantees target is put */
      shmem_put64(& flag, -999, 1, other_pe);
      ...
   }
   ...
}
```

12.22 Atomic Swap Operations

- Synopsis
  - C/C++:

    ```
    long shmem_swap (long *target, long value, int pe)
    ```

  - Fortran:

    ```
    integer shmem_swap, pe
    ires = shmem_swap(target, value, pe)
    ```

- Writes value `value` into `target` on processing element `pe` and returns the previous contents of `target` as an atomic operation

12.23 Collective Routines

The following are SHMEM collective routines:

- `shmem_and`
- `shmem_barrier`
- `shmem_broadcast`
- `shmem_collect`
- `shmem_max`
- `shmem_min`
- `shmem_or`
- `shmem_prod`
- `shmem_sum`
- `shmem_xor`

12.24 Collective Routines (continued)

- Collective routines need the include file *shmem.h* or *shmem.fh*
- All participating PEs must make the call; syntax is generally of the form:

```
shmem_func(*target, *source, nitems, PE_start,
           log2PE_stride, PE_size, *pWrk, *pSync)
```

  - `target`
    target address (symmetric)
  - `source`
    source address (symmetric)
  - `nitems`
    number of items (32-bit or 64-bit) to be transferred
  - `PE_start`
    base PE number
  - $\log_2$`PE_stride`
    $\log_2$ (stride between PEs)
  - `PE_size`
    number of PEs to make the call (the active set)
  - `pWrk`
    scratch work array used internally (symmetric)
  - `pSync`
    scratch synchronization array used internally (symmetric)

12.25 Collective Routines (continued)

- Example:

```
shmem_func ( *t, *s, n, 5, 1, 4, *pWork, *pSync)
```

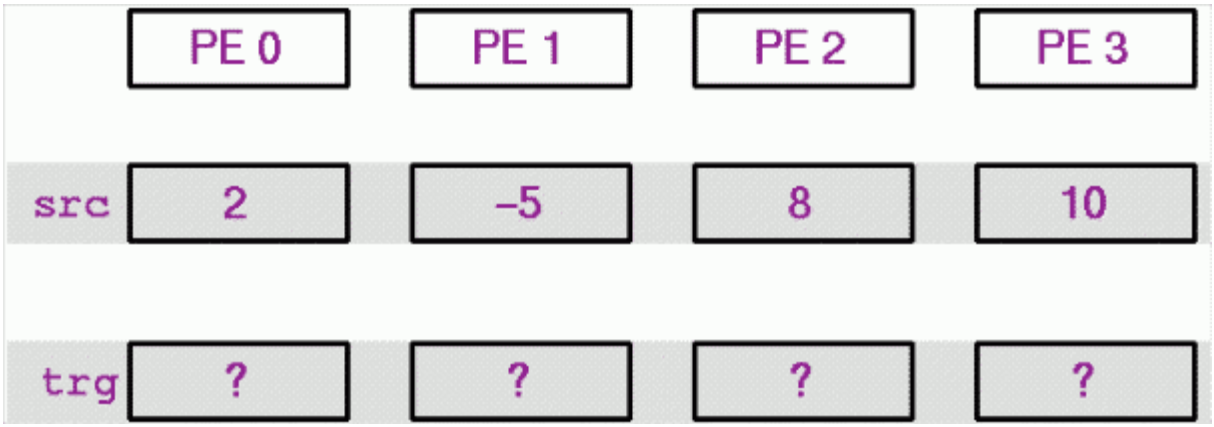
- The active set of PEs defined by (5, 1, 4) is: 5, 7, 9, 11
- All collective routines must initialize and use a `pSync` array
  - Using the `pSync` array
    - Size of the `pSync` array is different for different operations
      - Defined in the header file
    - All of `pSync` array should be initialized to `_shmem_sync_value` (`shmem_sync_value` for Fortran)
    - All PEs in the active set must initialize `pSync` before any PE calls a collective routine
    - Multiple `pSync` arrays are often needed if a particular PE calls a collective `SHMEM` routine more than once
      - Some PEs in the active set for the second call arrive at that call before processing of the first call is complete. There are two special cases:
      - The `shmem_barrier_all` routine allows the same `pSync` array to be used on consecutive calls as long as the active PE set does not change
      - If the same collective routine is called multiple times with the same active set, the calls may alternate between two `pSync` arrays. The `SHMEM` routines guarantee that a first call is completely finished by all PEs by the time processing of a third call begins on any PE
      - Because the `SHMEM` routines restore `pSync` to its original contents, multiple calls that use the same `pSync` array do not require that `pSync` be reinitialized after the first call
  - All collective routines must allocate a `pWrk` array
    - Size of the `pWrk` array is different for different operations
      - Defined in the header file

12.26 Reduction Routines

- `SHMEM` reduction routines perform reductions on symmetric arrays:
  - `shmem_and`
  - `shmem_broadcast`
  - `shmem_collect`
  - `shmem_max`
  - `shmem_min`
  - `shmem_or`
  - `shmem_prod`
  - `shmem_sum`
  - `shmem_xor`
- These routines perform a reduction across a set of PEs
- Return the reduction to all PEs
- Routines exist for `short`, `int`, `float`, `double`, `complexf`, `complexd` or in Fortran 90 `INT8`, `REAL8`, `COMP8`, `INT4`, `REAL4`, `COMP4`

12.27 Reduction Example

- Assume four PEs



12.28 Reduction Example (continued)

- Fortran:

```
npes = num_pes()
Call Shmem_int8_sum_to_all(trg, src, 1, 0, 0, npes, wrk, sync)
```

- C/C++:

```
npes = _num_pes();
shmem_int_sum_to_all(&trg, &src, 1, 0, 0, npes, wrk, sync);
```

12.29 Reduction Example (continued)

- Fortran:

```
npes =shmem_n_pes()
Call Shmem_int8_sum_to_all(trg, src, 3, 0, 0, npes, wrk, sync)
```

- C/C++:

```
npes = shmem_n_pes();
shmem_int_sum_to_all(trg, src, 3, 0, 0, npes, wrk, sync);
```

	PE 0	PE 1	PE 2	PE 3
src	2	–5	8	10
src	7	–2	–4	11
src	10	–8	6	8
trg	15	15	15	15
trg	12	12	12	12
trg	16	16	16	16

12.30 Example: Reduction (Fortran)

```
      PROGRAM TOALL
      INCLUDE 'mpp/shmem.fh'
      COMMON /BLK/ PWK(SHMEM_REDUCE_MIN_WRKDATA_SIZE), &
                   PSYNC(SHMEM_REDUCE_SYNC_SIZE), &
                   SHARED_SRC, SHARED_TGT
      INTEGER*8 :: PSYNC
      REAL*8 :: PWK
      REAL*8 :: SHARED_SRC, SHARED_TGT
      INTEGER*8, PARAMETER :: N = 256
      INTEGER*4 :: SHMEM_MY_PE
      REAL*8, DIMENSION(N) :: A
      REAL*8 :: GLOBAL_SUM
      CALL START_PES(0) !number of PE's determined at run-time
      NUMPES = NUM_PES()
      SHARED_SRC = 0.0
! INITIALIZE THE SYNC ARRAY
      DO I = 1, SHMEM_REDUCE_SYNC_SIZE
         PSYNC(I) = SHMEM_SYNC_VALUE
      ENDDO
      CALL SHMEM_BARRIER_ALL() ! must sync after initializing psync
! INITIALIZE THE LOCAL ARRAY AND PRIVATE SUM
      DO I = 1, N
         A(I) = SHMEM_MY_PE() * 256 + I
         SHARED_SRC = SHARED_SRC + A(I)
      ENDDO
! WAIT FOR ALL PES' TO COMPLETE INITIALIZATION
      CALL SHMEM_REAL8_SUM_TO_ALL(SHARED_TGT, SHARED_SRC, 1, &
                                  0, 0, NUMPES, PWK, PSYNC)
      GLOBAL_SUM = SHARED_TGT
      IF (SHMEM_MY_PE() == 0) &
         PRINT*,'GLOBAL SUM = ', GLOBAL_SUM
      END PROGRAM TOALL
```

```
% efc shmemall.f90 -lsma
% mpirun -np 4 ./a.out
GLOBAL SUM = 524800.
```

12.31 Example: Reduction (C/C++)

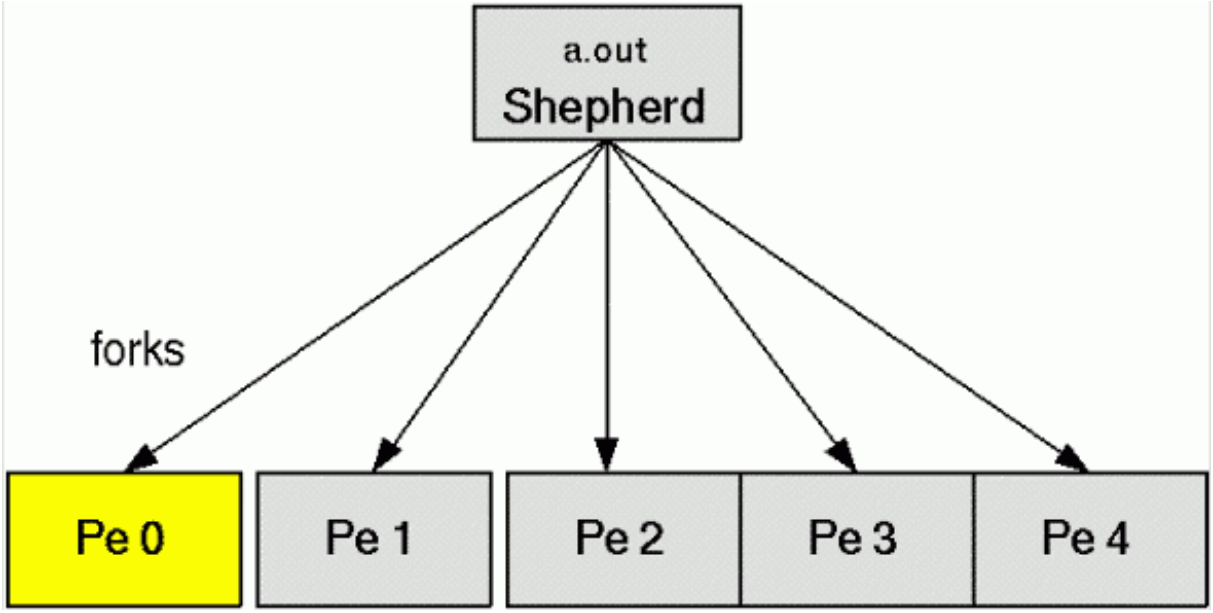
```
#include <stdio.h>
#include <mpp/shmem.h>
main()
{
   /* Static work space for shared memory library */
   static double shared_source = 0.0;
   static double shared_target;
   static double pWrk[_SHMEM_REDUCE_MIN_WRKDATA_SIZE];
   static long pSync[_SHMEM_REDUCE_SYNC_SIZE];
   /* Local variables for main */
   double A[256];
   int i;
   double global_sum;
   int mype, npes;
   start_pes(0); /* number of PE's determined at run-time */
   mype = shmem_my_pe();
   npes = _num_pes();
   /* Initialize the sync array */
   for (i=0; i < _SHMEM_REDUCE_SYNC_SIZE; i++)
      pSync[i] = _SHMEM_SYNC_VALUE;
   shmem_barrier_all(); /* must sync after initializing psync */
   /* Initialize the local array & sum */
   for(i = 0 ; i < 256; i++) {
      A[i] = (mype * 256) + i + 1;
      shared_source += A[i];
   }
   /* calculate the global sum */
   shmem_double_sum_to_all (&shared_target, &shared_source,
                            1,0,0,npes,pWrk,pSync);
   global_sum=shared_target;
   if (mype ==0 )
      printf("global sum = %f \n",global_sum);
}
```

```
% ecc shmemall.c -lsma
% mpirun -np 4 ./a.out
global sum = 524800.000000
```

12.32 Inside the Implementation

- Application launch and helper processes
- MPI message passing implementation
- `SHMEM` data-passing and synchronization implementation

12.33 SHMEM Application Running on SGI Altix Systems



12.34 SHMEM Process Relationships

- Startup initiated by `start_pes()` function
- Extra shepherd process is used
- Ancestor-descendent relationship and all job control is intact

12.35 SHMEM PUT/GET Implementation

- Memory copy via `fastbcopy()`, a fast version of `bcopy()` provided in the SGI MPI library
- Symmetric data objects must be used
    - Static memory
    - Symmetric heap

12.36 SHMEM Optimization Hints

- Gets are better than puts because of cache side-effects
- Reduce the frequency of barriers if possible
    - Remove extra barriers that often creep in
    - Consider using flag words and `shmem_wait()`
- Avoid use of `shmem_swap` for implementing locks. Use `shmem_lock`/`shmem_unlock` instead.

12.37 Instrumenting SHMEM

SHMEM has `_shmem*` names:

```
shmem_put32(args)          User-defined function
{
   datasent += len;
   _shmem_put32(args);     Actual shmem_put32 function
}
```

Lab: Using the shmem library

1. Write a program to distribute the work of initializing an array, A, where A(i) = i as i = 1 to 1024 (you can reuse the code from the MPI module).
   Share the work across four PEs. Have each processor calculate a partial sum for A.
   Use `shmem` calls to pass all the partial sums to one processor.
   What is the final sum?

   How well does your program "scale"?

   ---

2. Try using a shmem call to do the reduction.
   How many processors now have the final sum?

   ---

3. Compile and run the program *Altix/SHMEM/mxm4.mpi.[fc]*. This program must be run on four PEs.
   Convert the MPI calls to shmem calls. Run the code and make sure you get the same answers.
   How do the timings compare to the MPI code?

Module 13

Tuning Parallel Applications

13.1 Module Objective

After completing the module, you will be able to recognize performance problems introduced by parallel processing.

13.2 Prescription for Performance

- First, tune single-processor performance
  - Cache management is the most important factor in single processor performance on SGI Altix Systems (and most microprocessor-based systems)
- Has the program been properly parallelized?
  - Is enough of the program parallelized (Amdahl's law)?
  - Is the load well-balanced?
- OpenMP programs
  - Cache-friendly programs: no data placement tuning
  - Noncache-friendly programs: Is false sharing a problem?
    - Determine using `profile.pl`, `histx`, `Vtune`
    - Fix by modifying data structures

Notes:

The first step in tuning a parallel application is making sure that it has been properly parallelized. This means that enough of the code needs to have been parallelized to allow the program to attain the desired speedup. The Amdahl's law extrapolation shown below helps determine if this is a source of performance problems. Amdahl's law states that the speedup on N processors, S_N, for a program with a parellelizable fraction p of its running time, is:

$$S_N = \frac{1}{(1-p) + (p/N)}$$

From Amdahl's law you can calculate the value of p to extrapolate the potential speedup with higher numbers of processors. If N and M are the two processor counts for which speedups are known, then

$$p = \frac{S_N - S_M}{(1 - \frac{1}{N})S_N - (1 - \frac{1}{M})S_M}$$

13.3 Has the Program Been Properly Parallelized?

- Tuning data placement
  - Currently, first-touch is the only page-placement policy available on the Altix
  - Initialize data in parallel
- Amdahl's law: Sequential component needs to be small enough to allow desired scaling
- Workload needs to be balanced
  - Use `profile.pl` to measure each process's load
  - Use proper scheduling types and algorithms

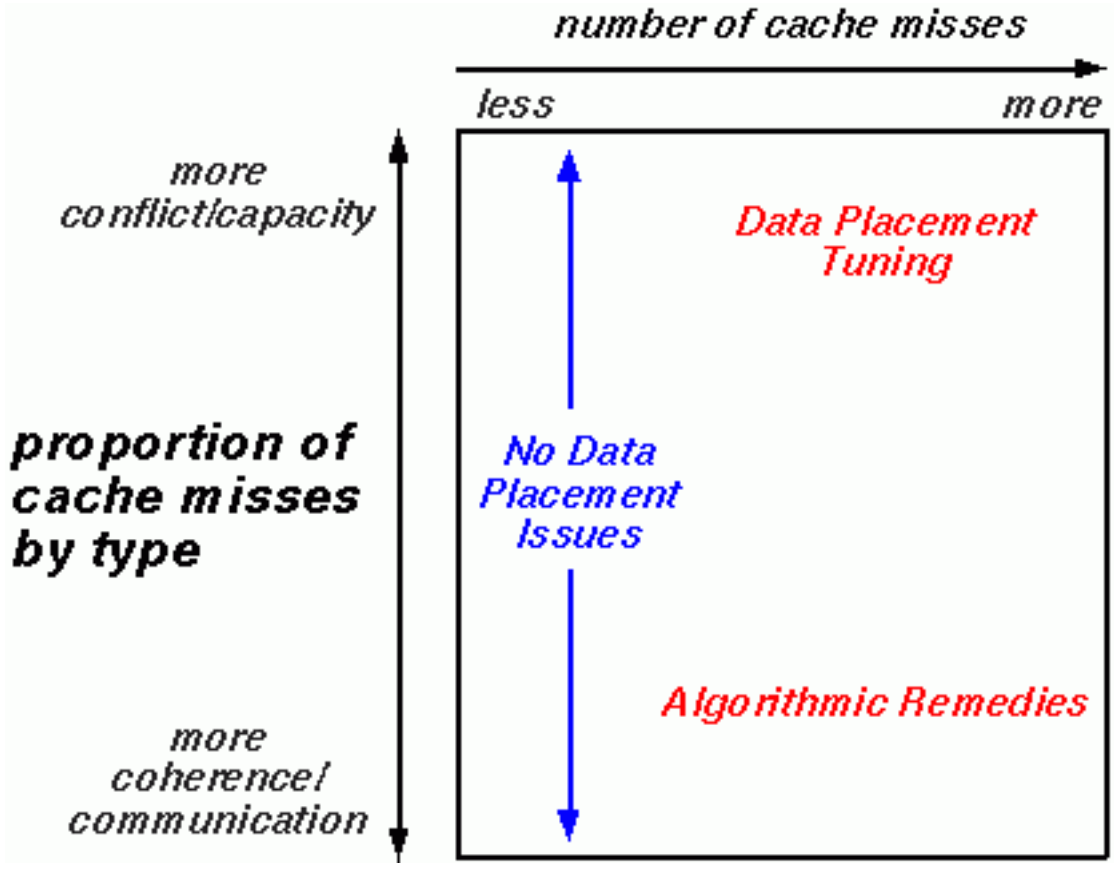
13.4 OpenMP Programs

- Platform-independent performance criteria
  - Amdahl’s law
  - Load balance
- Platform-dependent performance criteria
  - Eliminate cache coherency contention (shared memory)
  - Data placement (distributed shared memory)
- Data placement is not a problem for
  - Single-processor programs
  - Cache friendly programs

13.5 Non-Cache-Friendly Programs

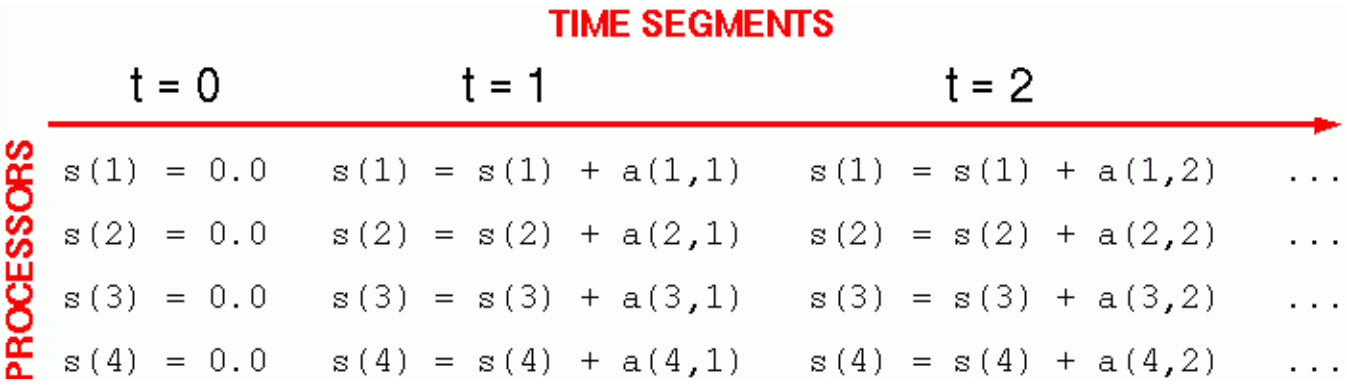
If scaling is poor, the first thing to check for is cache coherency contention

- Memory contention
  - One CPU repeatedly updates a cache line that other CPUs use for input
- False sharing
  - Two or more CPUs repeatedly update the same cache line
- Identified by high number of cache invalidations
  - Use `lipfpm(1)` and other `histx+` tools with event



13.6 False Sharing

```
      integer m, n, i, j
      real a(m,n), s(m)
c$omp parallel do private(i,j) shared(s,a)
      do i = 1, m
         s(i) = 0.0
         do j = 1, n
            s(i) = s(i) + a(i,j)
         enddo
      enddo
```



Notes:

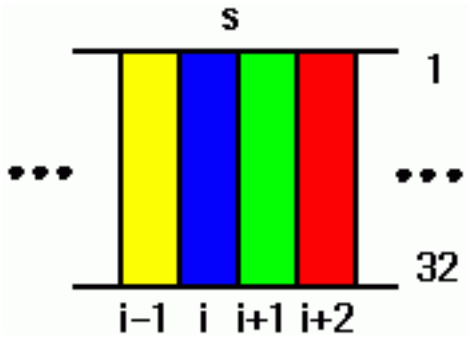
This code calculates the sums of the rows of a matrix. For simplicity, assume `m=4` and that the code is run on up to four processors. What you observe is that the time for the parallel runs is longer than when just one processor is used. The reason is that at each stage of the calculation, all four processors attempt to concurrently update one element of the sum array, `S`. For a processor to update one element of `S`, it needs to gain exclusive access to the cache line holding the element it wants to update. But because `S` is only four words in size, it is likely that `S` is contained entirely in a single cache line, so each processor needs exclusive access to all of `S`. Thus, only one processor at a time can update an element of `S`.

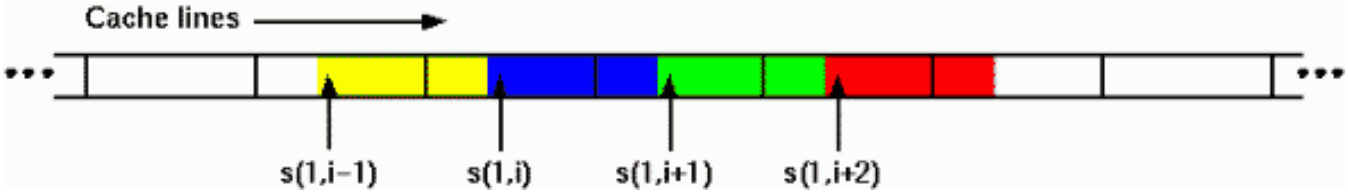
Actually, for a processor to gain exclusive access to a cache line, it first needs to invalidate any cached copies of `S` that might reside in the other processors. Then it needs to read a fresh copy of the cache line from memory, because the invalidations will have caused data in some other processor's cache to be written back to memory.

So, whereas in a sequential version, the element of `S` being updated can be kept in a register, in the parallel version, false sharing forces the value to continually be reloaded from memory, in addition to serializing the updates.

13.7 False Sharing Fixed

```
      integer m, n, i, j
      real a(m,n), s(32,m)
c$omp parallel do private(i,j) shared(s,a)
      do i = 1, m
         s(1,i) = 0.0
         do j = 1, n
            s(1,i) = s(1,i) + a(i,j)
         enddo
      enddo
```





Notes:

If each element of `S` was in a separate cache line, then each processor could keep a copy of the appropriate line in its cache, and the calculations could be done perfectly in parallel. One way to do this is to convert `S` to a two-dimensional array with the first dimension one (L3) cache line in size. Thus, the elements `S(1,1)`, `S(1,2)`, `S(1,3)`, and `S(1,4)` are guaranteed to be in separate cache lines.

13.8 Correcting Cache Coherency Contention

- Minimize the number of variables that are accessed by more than one CPU
- Segregate non-volatile (rarely updated) data items into cache lines different from volatile (frequently updated) items
- Isolate unrelated volatile items into separate cache lines to eliminate false sharing
- When volatile items are updated together, group them into single cache lines
    - Update all the data objects in a single guarded region

Notes:

Carefully review the design of data collections that are used by parallel code. For example, the root and the first few branches of a binary tree are likely to be visited by every CPU that searches that tree, and they will be cached by every CPU. However, elements at higher levels in the tree may be visited by only a few CPUs. One option is to pre-build the top levels of the tree so that these levels never have to be updated once the program starts. Also, before you implement a balanced tree algorithm, consider that tree-balancing can propagate modifications all the way to the root. It might be better to cut off balancing at a certain level and never disturb the lower levels of the tree. (Similar arguments apply to B-trees and other branching structures: the "thick" parts of the tree are widely cached, while the twigs are less so.)

Other classic data structures can cause memory contention, and algorithmic changes are needed to cure it:

- The two basic operations on a heap (also called a priority queue) are "get the top item" and "insert a new item." Each operation ripples a change from end to end of the heap-array. The same operations on a linked list are read-only at all nodes except for the one node that is directly affected.
- A hash table can be implemented compactly, with only a word or two in each entry. But that creates false sharing by putting several table entries (which are logically unrelated by definition) into the same cache line. Avoid false sharing: make each hash table entry a full 128 bytes, cache-aligned. Take advantage of the extra space to store a list of overflow hits in each entry. Such a list can be scanned quickly because the entire cache line is fetched as one operation.

13.9 Scalability and Data Placement

- Consider data placement tuning if your program
  - Is properly parallelized
  - Has no cache coherency contention
  - Shows less scaling than expected
- Data placement tuning is programming with data access patterns in mind

Notes:

- Optimized program's topology:
  Processes making up the parallel program should be run on nearby nodes to minimize access costs for data they share.
- Optimized page placement:
  Memory required for the data a processor accesses (most) should be allocated from its own node.

Accomplishing these two tasks automatically for all programs is virtually impossible. The operating system simply does not have enough information to do a perfect job.

13.10 Tuning Data Placement for OpenMP Library Programs

- Turn to data placement tuning after eliminating other possibilities
  - Parallelize data initializations
  - Modify algorithms

13.11 Programming for the Default First-Touch Policy

```
      integer i, j, n, niters
      parameter (n = 8*1024*1024, niters = 1000)
      real a(n), b(n), q
c initialization
      do i = 1, n
         a(i) = 1.0 - 0.5*i
         b(i) = -10.0 + 0.01*(i*i)
      enddo
c real work
      do it = 1, niters
         q = 0.01*it
         do i = 1, n
         a(i) = a(i) + q*b(i)
         enddo
      enddo
```

Notes:

If there is a single placement of data that is optimal for your program, you can use the first-touch policy to cause each processor's share of the data to be allocated from memory local to its node. As a simple example, consider parallelizing the above vector operation. This vector operation is "embarrassingly parallel," so the work can be divided among the processors of a shared memory computer any way you want. For example, if `p` is the number of processors, the first processor can carry out the first `n/p` iterations, the second processor the next `n/p` iterations, and so on. (This is called a *simple* schedule type.) Alternatively, each thread can perform one of the first `p` iterations, then one of the next `p` iterations, and so on. (This is called an *interleaved* schedule type.)

But for cache-based machines, not all divisions of work produce the same performance. The reason for this is that if a processor accesses the element `a(i)`, the entire cache line containing `a(i)` is moved into its cache. If the same processor works on the rest of the elements in the cache line, they will not have to be moved into cache, because they are already there. But if different processors work on the other elements, the cache line must be loaded into some processor's cache for each element. Even worse, false sharing is likely to occur. Thus, the performance is best for work allocations in which one processor is responsible for each element of the same cache line.

This brings up a subtle point involving the parallelization. In programming a typical distributed memory computer, you use data distribution to parallelize a program. That is, because a processor can only operate on data that are stored in its local memory, you break the arrays into pieces and store each piece in the memory of a single processor. This partitioning of the data then dictates what work a processor does: namely, it does the work that involves the data stored locally. Contrast this with a shared memory computer. In a shared memory parallelization, you distribute work (for example, iterations of a loop) to the processors, not data. Because the memory is shared, there really is no notion of data distribution: all data are equally accessible by all processors.

In practice, however, this distinction is generally lost, because it is usually easy to find a correspondence between the work and the data. For example, in the vector operation above, the processor that carries out iteration `i` is the only one that touches the data elements `a(i)` and `b(i)`, so these data have been effectively distributed to that processor. The distinction is further blurred by the presence of caches, because to achieve the best performance, the programmer needs to account for which cache lines are touched in carrying out a particular piece of work. Thus, even though the shared memory parallelization directives distribute work, programmers often think in terms of data distribution.

13.12 Programming for the Default First-Touch Policy (continued)

```
      integer i, j, n, niters
      parameter (n = 8*1024*1024, niters = 1000)
      real a(n), b(n), q
c initialization
c$omp parallel do private(i) shared(a,b)
      do i = 1, n
         a(i) = 1.0 - 0.5*i
         b(i) = -10.0 + 0.01*(i*i)
      enddo
c real work
      do it = 1, niters
         q = 0.01*it
c$omp parallel do private(i) shared(a,b,q)
         do i = 1, n
            a(i) = a(i) + q*b(i)
         enddo
      enddo
```

Notes:

This is exactly the same shared memory parallelization that is used on a bus-based machine. Note that because the schedule type is not specified, it defaults to simple: in other words, process 0 performs iterations `1` to `n/p`, process 1 performs iterations `n/p+1` to `2*n/p`, and so on. Use of the static schedule type is important. Because the initialization takes a small amount of time compared with the "real work," parallelizing it does not reduce the sequential portion of this code by much, so some programmers will not bother to parallelize the first loop for a traditional shared memory computer. However, if you are relying on the first-touch policy to ensure a good data placement, parallelizing the initialization code in the same way as the "real work" is critical.

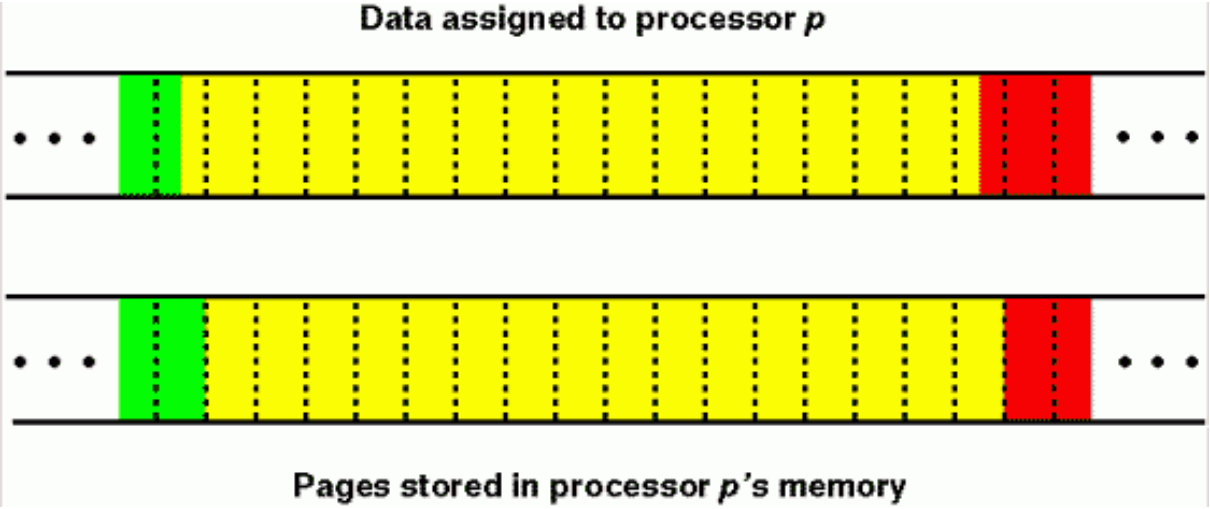
Due to the correspondence of iteration number with data element, the parallelization of the "real work" loop means that elements `1` to `n/p` of the vectors `a` and `b` are accessed by process 0. To minimize memory access times, you would like these data elements to be allocated from the memory of the node running process 0. Similarly, elements `n/p+1` to `2*n/p` are accessed by process 1, and you would like them allocated from the memory of the node running it, and so on. This is accomplished using the first-touch policy. The processor that first touches a data element causes the page holding that data element to be allocated from its local memory. Thus, if the data is to be allocated so that each processor can make local accesses during the "real work" section of the code, each processor must be the one to initialize its share of the data. This means that the initialization loop is parallelized the same way as the "real work" loop.

Now consider why the simple schedule type is important. Data is placed using a granularity of one page, so they will only be placed in their optimal location if the same processor initializes all the data elements in a page. This

is just like optimizing the parallelization to account for cache lines as was discussed earlier, only now you need to block the data into pages rather than lines. The default page size is 16 kB, or 4096 data elements, which is a fairly large number. Because the simple schedule type blocks together as many elements as possible for a single processor to work on (`n/p`), it creates more optimally allocated pages than any other work distribution.

13.13 Simple Schedule Type Is Important

Memory is allocated in pages ($\geq$ 16 kB)



Notes:

For the example above, `n` = 8388608. If the program is run on 128 processors, `n/p` = 65536, which means that each processor's share of each array fills 16 pages (65536 elements$\times$4B/element / 16 kB/page). It is unlikely that an array begins exactly on a page boundary, so you would expect 15 of a processor's 16 pages to contain only "its elements" and one page to contain some of its elements and some of another processor's elements. Although for the optimal data layout no pages would share elements from multiple processors, this small imperfection will have a negligible effect on performance.

On the other hand, if you use an interleaved schedule type, all processors repeatedly attempt to concurrently touch 128 consecutive data elements. Since 128 consecutive data elements are almost always on the same page, the resulting data placement could be anything from a random distribution of the pages to one in which all pages end up in the memory of a single processor. This initial data placement will affect the performance. In general, you should try to arrange it so that each processor's share of a data structure exceeds the size of a page. If you need finer granularity than this, you may need to consider using the reshaped directives.

13.14 Non-OpenMP Library Programs and dplace

- Message-passing: MPI, SHMEM
- pthreads
- `fork(2)`
- Control data placement via first-touch
- Using `fork(2)`, each process allocates its own memory
- Use `dplace(1)` either explicit or implicit (e.g., using `MPI_DSM_CPULIST` for MPI programs)

13.15 Summary

- First tune single-processor performance
- Tuning OpenMP parallel programs
  - Modify code if necessary to take advantage of the first-touch policy
- Non-OpenMP programs
  - Rely on first touch
- Use `dplace` and cpusets whenever possible

Module 14

Performance Analysis

14.1 Module Objective

After completing the module, you will be able to identify and use timing tools to profile single-CPU codes.

14.2 Sources of Performance Problems

- CPU-bound processes
  - Performing many “slow” operations
    - `sqrt`, fp divides
  - Non-pipelined operations
    - Switching between adds and mults
- Memory-bound processes
  - Poor memory strides
  - Page thrashing
  - Cache misses
  - Poor data placement (in NUMA systems)
- I/O bound processes
  - Performing synchronous I/O
  - Performing formatted I/O
  - Library and system level buffering

14.3 Profiling Tools

- `lipfpm`, `pfmon`: First-pass detection of program-wide performance issues
  - What is the problem?
- `pfmon`, `profile.pl`, `histx+`: Source-procedure annotation for focused experiments
  - Which procedures have the problems?

14.4 Hardware Counter Registers — Itanium 2

- Over a 100 countable events, counted in four 48-bit performance counters
  - Each counter is separately configurable
- May be used for
  - Workload characterization (system-wide tuning)
  - Overall program performance
  - Profiling (single-application performance analysis)

14.5 Event Categories

Performance-related events are classified into ten categories:

- Basic Events
- Instruction Dispersal Events
- Instruction Execution Events
- Stall Events
- Branch Events
- Memory Hierarchy
- System Events
- TLB Events
- System Bus Events
- Register Stack Engine Events

14.6 Basic Events

- Clock cycles

  `CPU_CYCLES`

- Retired instructions

  `IA64_INST_RETIRED`
  `IA32_INST_RETIRED`
  `IA32_ISA_TRANSITIONS`

14.7 Instruction Execution Events

- 18 events; some of the more interesting are:

```
FP_OPS_RETIRED
FP_FLUSH_TO_ZERO
LOADS_RETIRED
MISALIGNED_LOADS_RETIRED
STORES_RETIRED
MISALIGNED_STORES_RETIRED
```

14.8 Stall events

- 9 events; large numbers of stalls indicate major bottlenecks

```
BACK_END_BUBBLE
BE_EXE_BUBBLE
BE_FLUSH_BUBBLE
BE_L1D_FPU_BUBBLE
BE_LOST_BW_DUE_TO_FE
BE_RSE_BUBBLE
FE_BUBBLE
FE_LOST_BW
IDEAL_BE_LOST_BW_DUE_TO_FE
```

14.9 Memory Hierarchy Events

- Events related to cache and prefetch activity
  - 14 L1 data cache events divided into 5 mutually exclusive sets
  - 23 L2 cache events divided into 6 mutually exclusive sets
  - 5 L3 cache events

14.10 Other Events

- 4 Instruction Dispersal events
- 7 Branch events
- 6 System events
- 7 TLB events
- 30 System Bus events
- 8 Register Stack Engine (RSE) events

14.11 histx+

Set of tools for performance analysis and bottleneck identification

- Three data collection programs
  - `lipfpm`, `samppm`, `histx`
- Three filters for performance data postprocessing and display
  - `iprep`, `csrep`, `dumppm`

14.12 `lipfpm`: Linux IPF Performance Monitor

- Reports counts of desired events for entire run of a program

  `lipfpm` [*options*] *command* [*arguments*]

- Options
  - `-e` *cnt0* [`-e` *cnt1* `...`] specific counts (up to four)
  - `-i` interactive selection of events
  - `-f` follow forks
  - `-h` display a help message
  - `-k` include counts at kernel level
  - `-o` *path* send output to *path.command*.PID

Notes:

The subject program and its arguments are given. `lipfpm` sets up the counter interface and forks the subject program. `lipfpm` gathers its information with no modifications to the program. It generates a separate report for each process of an application that uses pthreads, OpenMP or MPI.

The `-i` option is handy if you want to choose interactively from the enormous list of countable events, but it precludes using standard input redirection for input read by the application.

One of the most useful features of `lipfpm` is its ability to account properly for applications that `fork`, pthreaded applications, and applications that `exec`. For example, an application called `ptprog` forks once, the fork child calls `exec`, and all instances create an extra thread for computation. When we run:

```
% lipfpm -f -e LOADS_RETIRED -e STORES_RETIRED -o cnts ptprog
[...normal ptprog output...]
% ls cnts.*
cnts.ptprog.16196   cnts.ptprog.16198   cnts.ptprog.16201
cnts.ptprog.16197   cnts.ptprog.16199   cnts.ptprog.16202
cnts.ptprog.16197-1 cnts.ptprog.16200   cnts.ptprog.16203
```

we get counts for each thread. (Note that `libpthread` creates an extra helper thread, so there are 9 total.) Since the `fork()` child has `exec()`ed the same ptprog, and it will have the same PID, `-1` is appended to the previously-used name to distinguish it. Subsequent `exec()`s would result in further suffixes: `-2`, `-3`,...

The command to use for MPI applications has the form

`mpirun -np` *n* `lipfpm -f` [*more_opts*] *command* [*args*]

`lipfpm` does *not* work with statically-linked programs, and neither does it exhaustively check the set of events requested for correctness. The user must only supply sets of events which the processor is capable of monitoring correctly. Performance monitoring events are discussed in Chapters 10 and 11 of the *Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*.

14.13 `samppm`

- Samples selected counter values at a rate the user may specify

  `samppm` [*options*] `-o` *path* [`-r` *n*] *command* [*arguments*]

- Options are the same as for `lipfpm`, except for `-i`, which is not available in `samppm`
- `-o` *path* is mandatory,
- `-r` specifies sampling rate in "ticks" (1 tick $\approx$ 0.977ms); default is 10
- Data is recorded in per-thread binary files called *path*`.`*command*`.`PID
- Can be used to provide time-varying performance metrics
- Load balance issues may be determined from inter-thread comparisons
- Readable data generated from binary files by `dumppm`

Notes:

At each sample time, `samppm` reads the counts and stores them in binary form in the output file along with a time stamp. The `dumppm` filter is then used to convert the binary file to a human or script readable tabular format. The notes for `lipfpm` apply to `samppm` as well.

14.14 `dumppm`: Dump PM results from `samppm`

- Generates a human- or script-readable tabular listing from binary files produced by `samppm`

  `dumppm [-c] [-d] [-h] [-l` *n1*`,`*n2*`,...] [[<] `*infile*`] [[>] `*outfile*`]`

- Options:

  `-c` only list events
  `-d` print differences and relative times
  `-h` print help
  `-l` print only events *n1*, *n2*, … (event 0 is time)

- `dumppm` can operate as a filter:

```
% cat file.prog.1234 | dumppm -d | more
% dumppm -d < file.prog.1234 | more
% dumppm -l 2,3 file.prog.1234 > results
% dumppm -d file.prog.1234 results
```

Notes:

The `-c` flag is used to quickly tell one what is contained in the file, and shows the corespondence between event and output column. For example:

```
% samppm -e CPU_CYCLES -e LOADS_RETIRED -r 2 -o dat myprog
  <normal program output here>
% dumppm -c dat.myprog.3214
dumppm: event 1 is: CPU_CYCLES
dumppm: event 2 is: LOADS_RETIRED
%
```

If we now drop the `-c`, the output will consist of 3 columns: time of day, sampled `CPU_CYCLES` value, and sampled `LOADS_RETIRED` value, e.g.:

```
1042470662.010721             2059042              420371
1042470662.012656             3454206              711769
1042470662.014595             4866156             1006722
. . .
```

If we add the `-d` flag, time becomes relative, and differences between counts are output instead:

```
0.000000            2059042             420371
0.001935            1395164             291398
0.003874            1411950             294953
. . .
```

If we want a different order of, and possibly a different number of columns, we can use the `-l` flag to specify which of the original columns to output, in which order. For instance, to output the change in `CPU_CYCLES` followed by relative time we use `-d -l 1,0`:

```
2059042         0.000000
1395164         0.001935
1411950         0.003874
. . .
```

14.15 `histx`: Histogram Execution

- Profiling tool, it can sample either the IP (instruction pointer, aka program counter) or the call stack

  `histx [`*options*`] -o` *path* `[-s` *type*`]` *command*

- Options:

`-b`	specify bin bits when using ip sampling: 16, 32 or 64 (default: 16)
`-e`	specify event source (default: `timer@1`)
`-f`	follow fork
`-h`	this message (command not run)
`-k`	also count kernel events for PM source
`-l`	include line level counts in IP sampling report
`-o`	send output to file *path*`.`*command*`.`PID
`-s`	type of sampling

- Event sources:

`timer@`*n*	profiling timer events. A sample is recorded every *n* ticks. (One tick is about 0.977 ms.)
`pm:<`*event*`>@`*n*	performance monitor events. A sample is recorded when the counter associated with `<`*event*`>` increases by *n* or more.

- Types of sampling:

`ip`	Sample instruction pointer
`callstack[`*N*`]`	Sample the call stack. *N*, if given, specifies the maximum number of frames to record. The default is 5 frames.

Notes:

For long running programs where any particular instruction bundle could be sampled more that 65535 times, the counts attributed to a function may not be accurate. The `-b` flag can be used in such cases to switch to accumulators

that can handle counts of up to 2^{32}-1 or even 2^{64}-1 before overflowing. However, these require the use of more memory per thread.

`histx` produces a separate output file for each thread in a program. It also handles programs that `exec`, and, similarly to `lipfpm` and `samppm`, in cases where the `exec`'d program name is the same as the original, will produce multiple reports in files named *path*`.`*prog*`.`PID`-`*exec_instance*.

The program to be analyzed must not be compiled with the `-p` compiler flag.

14.16 `histx`: IP Sampling

```
% histx -o out prog
% sort -r out.prog.* | head
                  586: *Total for thread 18458*
                  454: libm.so.6.1:*Total*
                  279: libm.so.6.1:cos
                  118: a.out:*Total*
                  109: libm.so.6.1:asin
                   66: a.out:f1 [prog.c:19]
                   45: libm.so.6.1:acos
                   32: a.out:f2 [prog.c:29]
                   21: libm.so.6.1:sin
                   20: a.out:_init
```

Notes:

The default type of sampling is IP sampling, and the default event source is a timer event at a rate of ~1 kHz. What we see in each line of output is

```
<count>: <library>:<function> [<file>:<line>]
```

`<count>` is the number of events that occurred within the body of `<function>` contained in `<library>`, starting at `<line>` within `<file>`. In the case of the main executable, `<library>` is "`a.out`". If line information is not available, the "`[<file>:<line>]`" output will not appear.

Totals are also accumulated on a per library, and per thread basis. Note that `<line>` corresponds to the first source line of `<function>`, and *not* the line where the sample was taken. The count attributed to `<function>` is the sum of counts in all bins interior to `<function>`.

If it is desired to see counts attributable to individual lines, and if the executable contains line information (produced with `-g` compiler flag), one can add the -l flag, e.g.:

```
% histx -l -o out prog
% sort -r out.prog.* | head
                580: *Total for thread 3704*
                399: libm.so.6.1:*Total*
                203: libm.so.6.1:cos
                167: a.out:*Total*
                 98: libm.so.6.1:acos
                 89: libm.so.6.1:asin
                 82: a.out:f1 [prog.c:19]
                 82: [prog.c:23] (a.out)
                 67: a.out:f2 [prog.c:29]
                 67: [prog.c:33] (a.out)
```

Now we see additional lines in the report of the form

```
<count>: [<file>:<line>] (library)
```

Here, `<count>` is the number of events that occurred on the instructions generated by the compiler for `<line>` within `<file>` contained in `<library>`. From the report above, we can see that all 82 events recorded in function `f1()` were recorded on line 23 of *prog.c*.

Function level counts are computed by summing over all bins interior to a function. Line level counts are computed by summing over all bins intersecting instructions for a particular source line. The two are different as instructions from multiple lines can fall into a single bin. Note that compilers and linkers are known to occasionally generate incorrect or invalid line information. This can confuse `histx`, or even cause the application to crash.

`histx` can do IP sampling at very high rates when using performance monitor register overflows. For example, to sample the IP every 10,000 CPU cycles (i.e. around 100 kHz on a 1 GHz processor system):

```
% histx -l -o out -e pm:CPU_CYCLES@10000 prog
% sort -r out.prog.* | head
               41675: *Total for thread 3711*
               27342: libm.so.6.1:*Total*
               13382: a.out:*Total*
               12072: libm.so.6.1:cos
                7796: libm.so.6.1:acos
                6709: libm.so.6.1:asin
                6456: a.out:f1 [prog.c:19]
                6456: [prog.c:23] (a.out)
                5753: a.out:f2 [prog.c:29]
                5753: [prog.c:33] (a.out)
```

The IP sampling report is deliberately as simple as possible. It is envisioned that filters (e.g., `iprep`, see below) will be used to transform `histx` output into something more intuitive.

14.17 `histx`: Call Stack sampling

```
% histx -o cs -s callstack10 prog
% more out.prog.*
. . .
101 5
          libm.so.6.1:cos
          a.out:f1 [prog.c:23]
          a.out:main [prog.c:70]
          libc.so.6.1:__libc_start_main [../sysdeps/generic/libc-start.c:129]
          a.out:_start
106 5
          libm.so.6.1:asin
          a.out:f1 [prog.c:23]
          a.out:main [prog.c:70]
          libc.so.6.1:__libc_start_main [../sysdeps/generic/libc-start.c:129]
          a.out:_start
. . .
```

Notes:

While IP sampling can tell us which function events are occuring in, it is often important to understand the control flow that led to the function being called. This is done by not sampling just the IP when an event occurs, but rather the call stack. To activate call stack sampling, use `-s callstack`*N* where *N* is the maximum number of frames to record. (Larger values require more memory.)

Each sampled call stack is reported in the form:

```
<count> <N>
        <library1>:<function1> [<file1>:<line1>]
        <library2>:<function2> [<file2>:<line2>]
        . . .
        <libraryN>:<functionN> [<fileN>:<lineN>]
```

The most sampled call stacks are those with the largest `<count>` value.

A filter (`csrep`) is provided to generate a "butterfly" report from raw callstack reports.

14.18 `iprep`: IP Sampling Report

- Generates a report from one or more raw IP sampling reports produced by `histx`

```
% iprep ip.prog.* > report.all
% more report.all
       Count           Excl. %   Incl. %   Name
--------------------   -------   -------   --------------------
               12362    29.730    29.730   libm.so.6.1:cos
                7716    18.557    48.286   libm.so.6.1:acos
                6533    15.712    63.998   libm.so.6.1:asin
                6338    15.243    79.241   a.out:f1 [prog.c:19]
                5655    13.600    92.840   a.out:f2 [prog.c:29]
                1401     3.369    96.210   a.out:_init
                 625     1.503    97.713   libm.so.6.1:sin
. . .
       Count           Excl. %   Incl. %   Name
--------------------   -------   -------   --------------------
                6345    48.989    48.989   [prog.c:23] (a.out)
                5401    41.700    90.689   [prog.c:33] (a.out)
                  96     0.741    91.430   [elf/elf.h:80] (ld-linux-ia64.so.2)
. . .
```

Notes:

Output from `iprep` consists of one or two sections. The first section is a listing of summed functions counts, sorted from largest to smallest, along with their exclusive and inclusive percentages. If line level data is available, the second section presents it in a similar format.

14.19 `csrep`: Call Stack Sampling Report

- Generates a “butterfly” report from one or more raw call stack sampling reports produced by `histx`

```
% csrep cs.prog.* > butterfly.all
% more butterfly.all
 98.1% libc.so.6.1:__libc_start_main [libc-start.c:129]
 97.5% a.out:_start
 48.7% a.out:main [prog.c:93]
 48.7% a.out:main [prog.c:70]
 36.8% a.out:f1 [prog.c:23]
 34.4% a.out:f2 [prog.c:33]
 31.7% libm.so.6.1:cos
 17.9% libm.so.6.1:asin
 14.8% libm.so.6.1:acos
. . .
  -------------------------------------------------------------------------------
  100.00% ( 48.74%) libc.so.6.1:__libc_start_main [libc-start.c:129]
  ....... ( 48.74%) a.out:main [prog.c:70]
   75.43% ( 36.76%) a.out:f1 [prog.c:23]
   24.57% ( 11.97%) a.out:f1 [prog.c:19]
. . .
```

Notes:

Output from `csrep` consists of 2 sections. The first one is a listing of the frequency of appearance of each function found in a callstack. The second section consists of a “butterfly” report for each function. Each has the form:

```
  -------------------------------------------------------------------------------
  xxx.xx% (yyy.yy%) <caller_1 info>
  xxx.xx% (yyy.yy%) <caller_2 info>
  xxx.xx% (yyy.yy%) <caller_3 info>
  . . .
  xxx.xx% (yyy.yy%) <caller_M info>

  ....... (yyy.yy%) <function info>
  xxx.xx% (yyy.yy%) <callee_1 info>
  xxx.xx% (yyy.yy%) <callee_2 info>
  xxx.xx% (yyy.yy%) <callee_3 info>
  . . .
  xxx.xx% (yyy.yy%) <callee_N info>
```

This conveys a lot of information. Above the function of interest, there is a line for each caller of the function. Each `xxx.xx` percentage is computed from the formula:

```
<# callstacks containing caller_j followed by function> /
<# callstacks containing function> * 100
```

so the sum for all callers is 100%. Each `yyy.yy` percentage is given by:

```
<# callstacks containing caller_j followed by function> / <total # of callstacks> * 100
```

Below the function of interest, there is a line for each callee of the function. The `xxx.xx` percentages are now given by the equation:

```
<# callstacks containing function followed by callee_j> /
<# callstacks containing function> * 100
```

and again sum to 100%, while the `yyy.yy` percentages are given by

```
<# callstacks containing function followed by callee_j> /
<total # of callstacks> * 100
```

This type of report allows one to not only quickly identify the functions of interest in an application, but also identify the must likely paths through the program by which such functions were called.

14.20 `pfmon`: Performance Monitor

- Performance monitoring tool — interface to the PMU
- Two types of sessions
  - Per process
  - System-wide
- User and/or kernel activity may be monitored in either type of session

14.21 `profile.pl`

- Perl script interface to `pfmon`
- Uses `dplace` to bind the application to spoecific processors
- Invokes other Perl scripts to generate a readable report

```
% profile.pl -c1 ./adi2
profile.pl: Parsing arguments and setting defaults.
profile.pl: Samples/tick defaults to: 9000000 for event CPU_CYCLES.
profile.pl: Program to profile is: ./adi2.
profile.pl: Running the program under pfmon control:
profile.pl: pfmon --system-wide --smpl-outfile=sample.out --smpl-entries=100000
  -u -k --short-smpl-periods=9000000 --smpl-output-format=compact --events=CPU_CYCLES
  --cpu-mask=2 /usr/bin/dplace -c1     ./adi2
================================================================================
 Time:     16.465 seconds
 Checksum:  4.6021112569E+07
================================================================================
profile.pl: Program has completed.
profile.pl: Checking the profile results.
profile.pl: cpu 1: 1733 samples.
profile.pl: Merging sample files into a single file. profile.pl: cat sample.out.cpu1  > sample.out
profile.pl: Removing the per-processor sample files.
profile.pl: rm -f sample.out.cpu1
profile.pl: Creating a program map file.
makemap.pl: Read     45 symbols from ./adi2.
makemap.pl: Read  14099 symbols from /sw/com/scsl/1.4.1-2/lib/libscs.so.
makemap.pl: Read    590 symbols from /lib/libm.so.6.1.
makemap.pl: Read   1121 symbols from /sw/com/intel-compilers/7.1.017/compiler70/ia64/lib/libcxa.so.4.
makemap.pl: Read   2587 symbols from /lib/libc.so.6.1.
makemap.pl: Read    224 symbols from /lib/ld-linux-ia64.so.2.
makemap.pl: Sorting symbols.
makemap.pl: Wrote 18670 symbols to   adi2.map
profile.pl: Running the profile analyzer.
profile.pl: analyze.pl adi2.map sample.out  > profile.out
analyze.pl: Read 18671 symbols from adi2.map.
analyze.pl: No System.map file found; kernel analysis will be skipped.
analyze.pl: total observations:        1685
analyze.pl: Sorting the user observations
profile.pl: Profile results are in file: profile.out.
```

14.22 `profile.pl` (continued)

```
% cat profile.out
total observations:   1685
user ticks:          1685          100 %
=============================================================================
                           User
      Ticks     Percent  Cumulative   Routine
                          Percent
-----------------------------------------------------------------------------
       1182       70.15    70.15      zsweep_
        196       11.63    81.78      ysweep_
        172       10.21    91.99      xsweep_
         92        5.46    97.45      drand64_
         32        1.90    99.35      main
         10        0.59    99.94      _init
          1        0.06   100.00      _dl_relocate_object
=============================================================================
```

Lab: Using the histx+ and profile.pl tools

Profiling a program

- Change to the *Altix/Performance_analysis/labs/[f|c]src* directory.
- Type **`make mat_dist`** to build the executable `mat_dist`.
- Analyze the performance of `mat_dist` with `histx` and `profile.pl` (such as pc sampling, ideal time, user time).
- Determine which line[s] is most time-consuming in `mat_dist`.
- Can you suggest any techniques to make `mat_dist` run faster?

Module 15

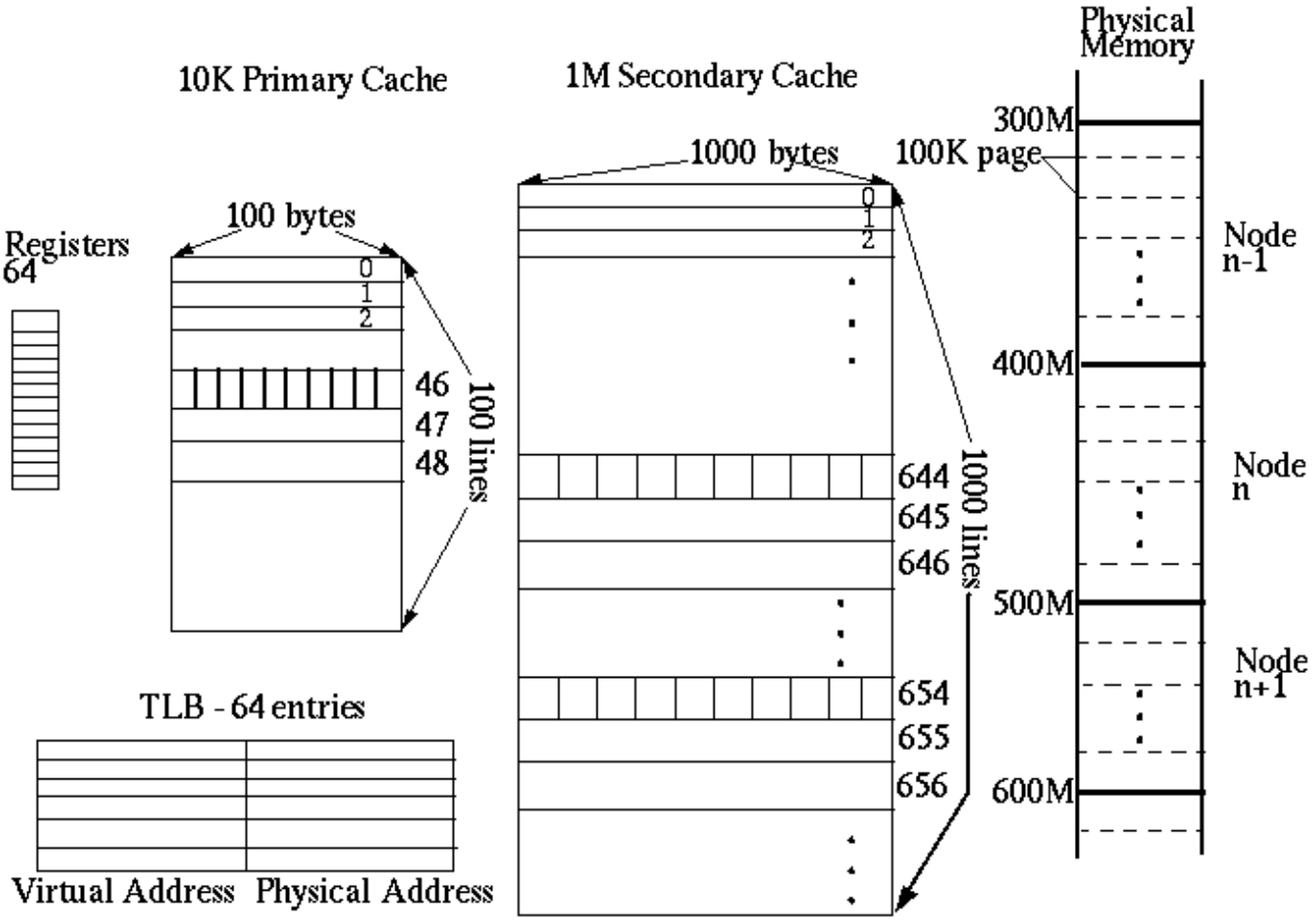
Single-CPU Optimization Techniques

15.1 Module Objectives

After completing the module, you will be able to

- Understand the relationship between the SGI Altix systems hardware and the compilers
- Invoke and understand different compiler optimization options to enhance performance of single-CPU codes

15.2 Memory Hierarchy



Notes:

The processors in SGI Altix systems will first look in the primary cache to satisfy a load request. For ease of understanding, we are assuming a direct mapped cache, and only two levels of cache. Any requested address can reside in the primary cache in exactly one location. All data is moved in and out of cache one cache line at a time. In our example, this would be 100 contiguous bytes of storage. So if our data is in cache, it will be in one of the 100 possible cache lines. We will use a digit mask (bit map on the SGI Altix Systems) to determine which cache line to look at to see if it contains our data.

Once we determine which cache line to look at, we check the cache tag on that line to see if it matches the rest of our address. If it does, this line contains our data, we have a primary cache hit and we have minimal latency. If the cache tag does not match the rest of our address, we have a cache miss and we will look at the secondary cache for our data.

For the ease of understanding let's assume a machine similar in architecture to the SGI Altix Systems, but we will use a base 10 addressing scheme. We will use digit masks to locate address quickly in the cache, in the SGI Altix Systems these would be bit masks.

Because this is a RISC architecture, data must be loaded into registers before any work can be performed on it. To achieve maximum performance on this chip we need to minimize the latency of the load. The latency to load from primary cache is one or two clock periods(int/float). The latency to load from secondary cache is 8 clock periods. The latency to load from memory varies depending on the speed of the processor chip and the location of the data in physical memory. These latencies range from 60–200+ clock periods.

We will also use a direct mapped cache for our secondary cache. Each cache line here is 1000 bytes of contiguous memory and we have 1000 lines of cache. The digit mask here will be different.

Once we determine which cache line our data could reside in, we check the cache tag on that line to see if it matches the rest of our address. If it matches, we have a cache hit, and we will move 100 bytes of data into the primary cache. If the tag doesn't match, we have a cache miss and we will have to go to memory to find our data.

15.3 Example Code

```
DIMENSION /ABC/ A(1000,1000),B(1000,1000),C(1000,1000)
DO I = 1 , 1000
  DO J = 1 , 1000
     A(I,J) = A(I,J) + B(I,J) * C(I,J)
  ENDDO
ENDDO
```

Notes:

This code would generate a series of instruction along these lines:

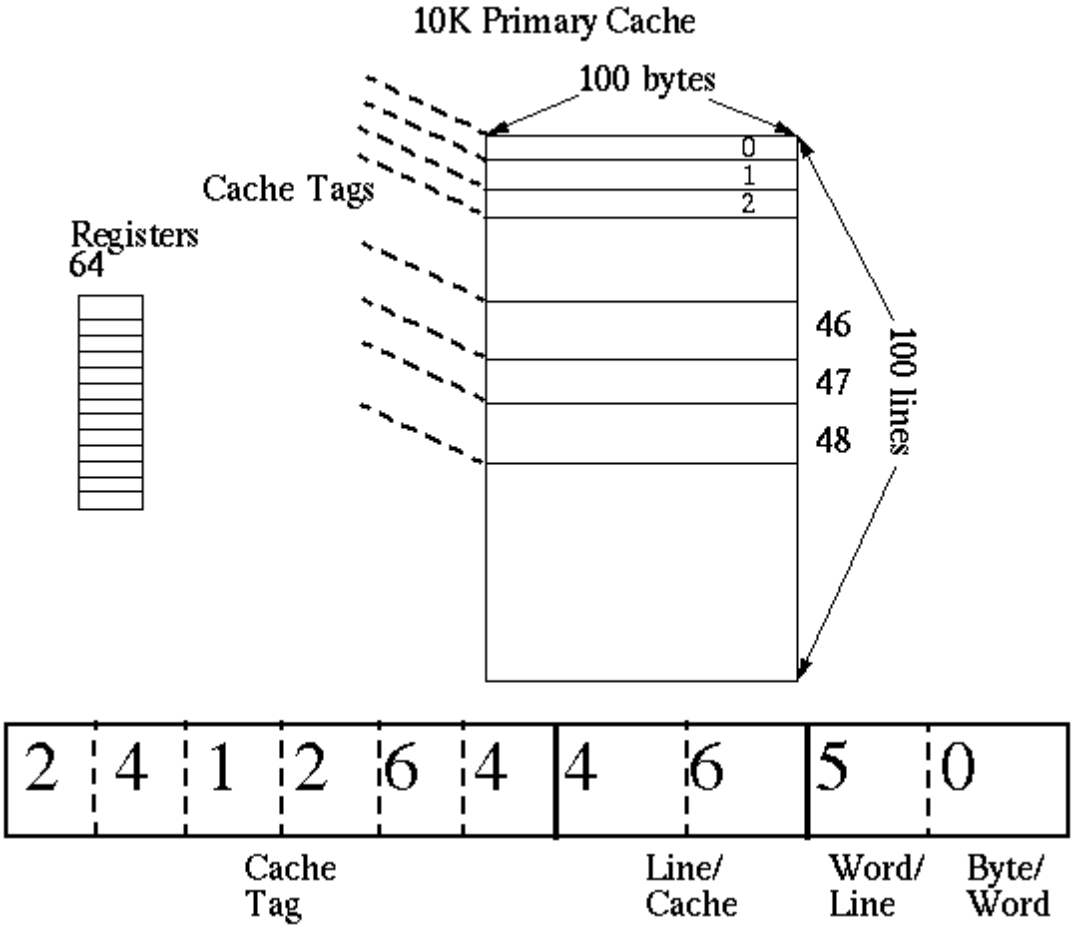
```
Load A(1,1) into R1
Load B(1,1) into R2
Load C(1,1) into R3
Multiply R2 and R3 giving R4
Add R4 and R1 giving
R5 Store R5 into A(1,1)
Increment/Test/Jump
```

In assembly code `A(1,1)`, `B(1,1)`, and `C(1,1)` would be offsets from the base addresses of `A`, `B` and `C`. Let's look at these loads.

A C code with a similar problem:

```
double a[1000][1000],b[1000][1000],c[1000][1000];
int i,j;
. . .
for (j=0,j<1000,j++)
   for (i=0,i<1000,i++)
      a[i,j] += b[i][j] * c[i][j];
```

15.4 Primary Cache



Notes:

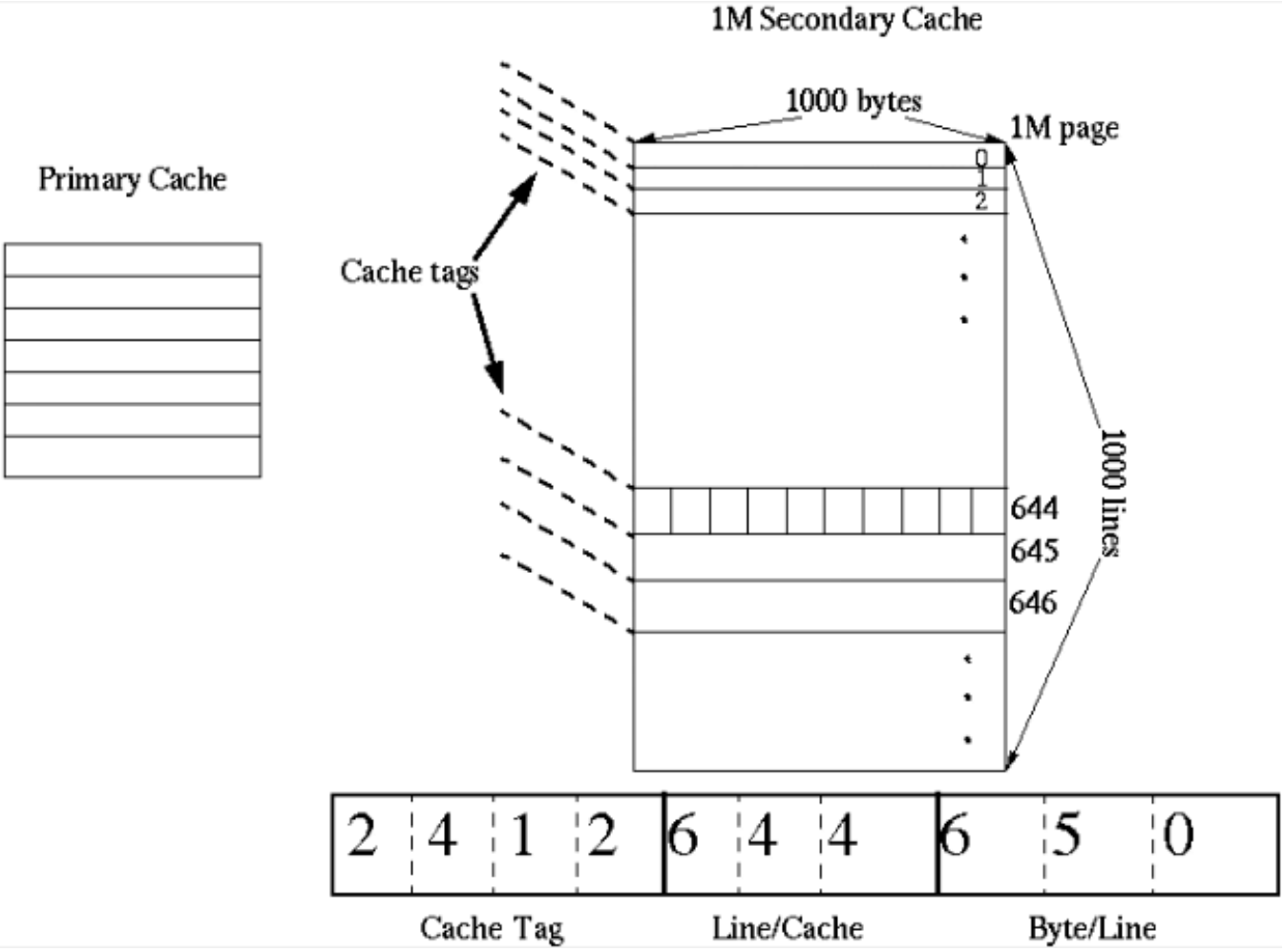
Let's look at the load of `A(1,1)`.

We will use 2412644650 as the base address of `A`.

Although all or pieces of `A` may reside in cache from data initialization, let's assume no data from `A`, `B` or `C` is in either cache.

The first place we will look for `A(1,1)` is in primary cache. We apply the primary cache mask to the address of `A(1,1)`. This tells us that if `A(1,1)` is in primary cache, it must be at line number 46. We compare the cache tag at line 46 to see if it matches the cache tag portion of our address, 241264. Here we assume the tags don't match. This gives us a primary cache miss.

15.5 Secondary Cache



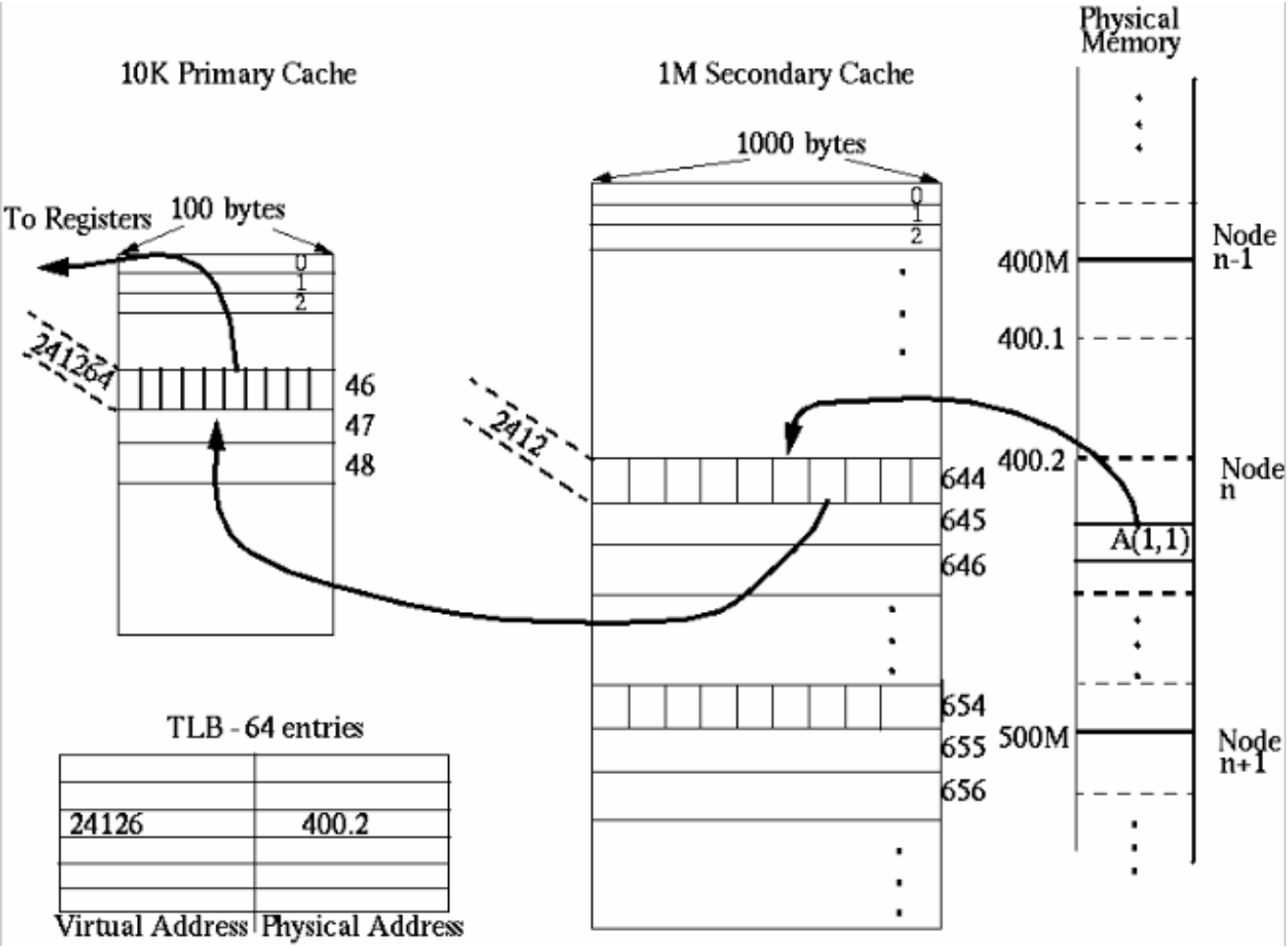
Notes:

We will now try secondary cache.

We apply the secondary cache mask to the address of `A(1,1)`. This tells us that if `A(1,1)` is in secondary cache, it must be at line number 644. We compare the cache take at line 644 to see if it matches the cache tag portion of our address, 2412. Again we assume it does not match. We now have a secondary cache miss.

We now must load our data from memory.

15.6 First Data Movement



Notes:

We are using a virtual addressing scheme for our main memory. This means that the virtual address we are attempting to load could be located anywhere in memory. Although the physical memory is distributed across the nodes, all processors have access to all of memory. The address we are looking for may or may not be on this node.

Physical memory is allocated in pages, on our machine 100K bytes of contiguous memory. The OS maps and remaps virtual addresses to these physical blocks. To find the true physical address of the virtual address we are looking for we use the on-chip Translation Look-aside Buffer. This table has a limited number of entries, 64 on our machine. This means on our chip we can keep track of 6.4 MB of memory. In the Itanium 2 processors, there are two levels of TLB for data and instructions.

If the virtual address we are looking for has an entry in the TLB, we use the offset from the beginning of the page and read a secondary cache line from that offset in the physical page. The latency here depends on where this page is in physical memory.

If the virtual address we are looking for is not in the TLB, this is a page fault. We must ask the OS to update our TLB with the entry we are looking for. The OS keeps track of the location of all pages of memory. If the page is loaded in physical memory, the OS updates our TLB and returns control to the program to read the data. This is a minor page fault. If the OS has moved the page to swap space, then it must reload it into physical memory, then update our TLB. This is a very expensive event called a major page fault.

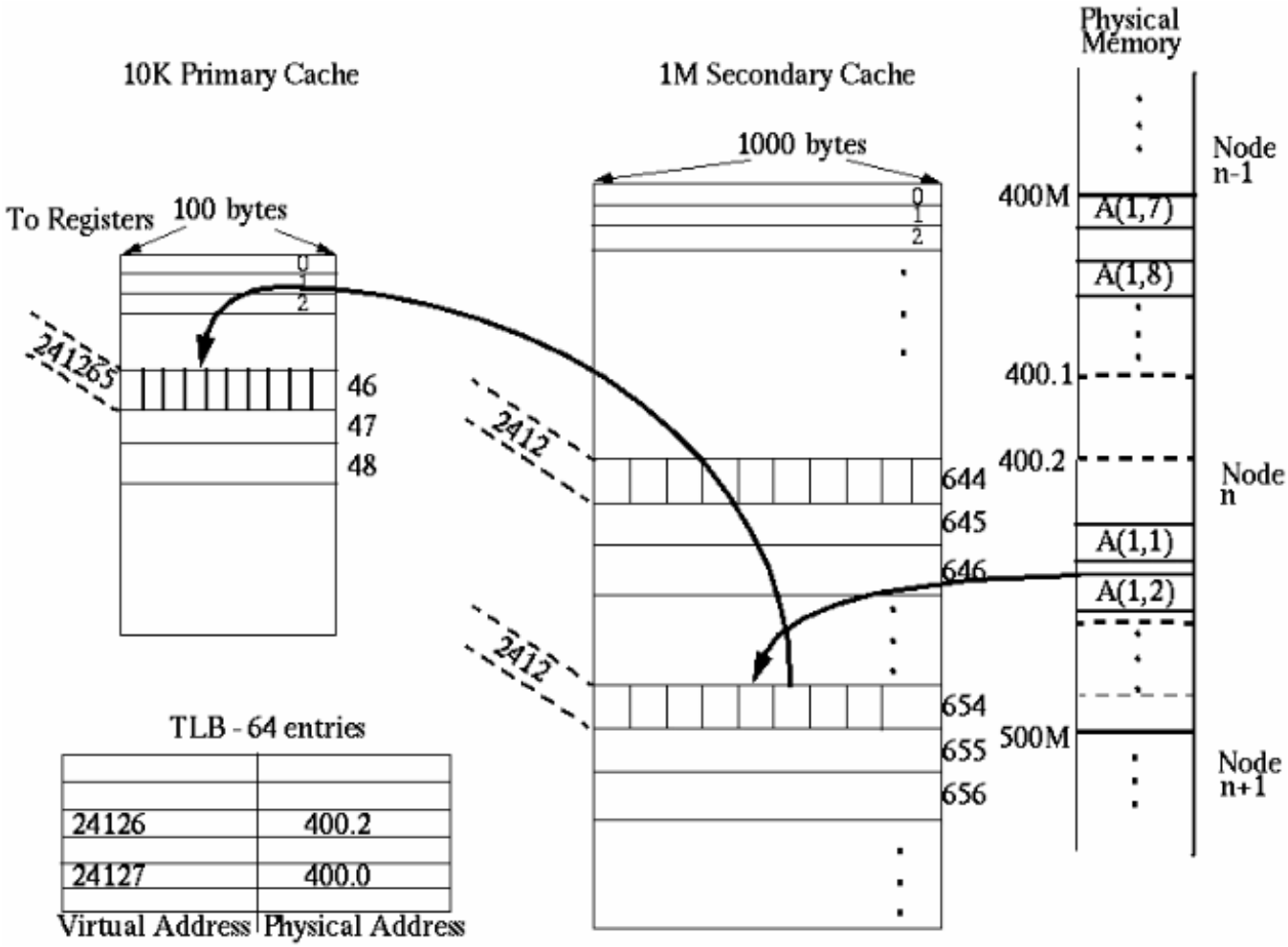
To find our data in memory, we look for the base address of the page we wish to load in the TLB. The base address of the virtual page containing `A(1,1)` is 2412600000, assuming a 100-kB page size.

We find virtual page 24126 maps to physical page 400.2.

We now load 1000 bytes of data starting at 400264000 into secondary cache line 644, and change the cache tag for line 644 to 2412.

We are looking for the 650th byte of this cache line to move into primary cache. Since we always move data on cache line boundaries, we will load bytes 600–699 from secondary cache line 644 into primary cache line 46. We then update the cache tag on line 46 of the primary cache to 241264.

15.7 Second Data Movement



Notes:

Let's skip the rest of the first pass of the loop and consider what happens on the second pass. On the second pass of the loop, we need to load `A(1,2)`. The address of `A(1,2)` is 10,000 bytes after the address of `A(1,1)` or 2412654650.

We look in primary cache line 46 for cache tag 241265, we get a cache miss. We look in secondary cache line 654 for cache tag 2412, we get a cache miss. We look in the TLB for 24126, and go load 1000 bytes into secondary cache line 654 starting from 400254000. We change the cache tag on line 654 to 2412, move 100 bytes starting at byte 600 into primary cache line 46 and change the cache tag to 241265.

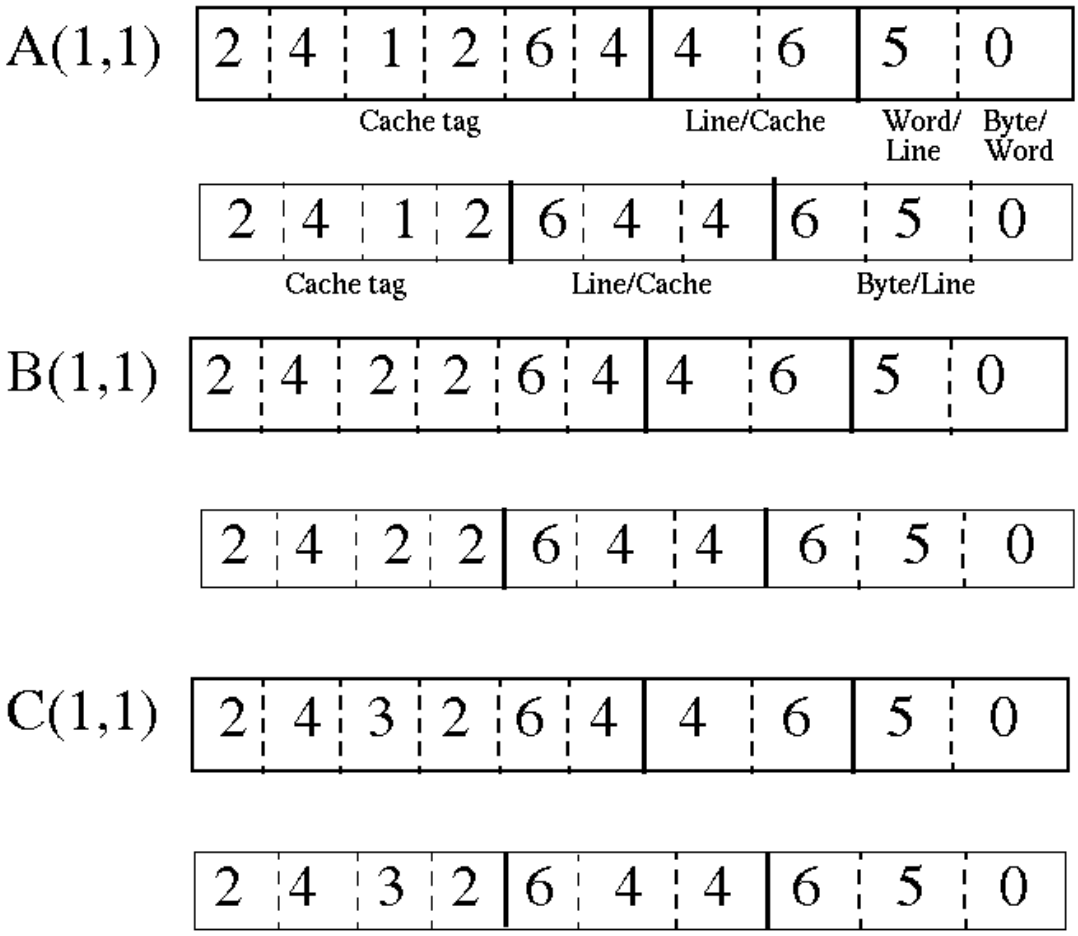
The load of `A(1,j)` will exhibit similar behavior for the next four loads. When we attempt to load `A(1,7)` from address 2412704650, we will need to look up the physical address for 24127 in the TLB. Due to the natural of virtual memory, it is possible that 24127 may reside in physical memory before 24126, for example at 400.0.

The next nine loads would come from this page. This pattern will repeat, one TLB lookup for every 10 loads. User code has less than 64 entries to use in the TLB table, so after approximately 600 loads, we will start getting TLB misses. Because the TLB is LRU, we will get minor page faults on every 10th load for the rest of this code.

Because we are only using every 10th line of the secondary cache, when we load `A(1,100)`, address 2413644650, we will begin overwriting everything we have loaded into secondary cache.

By the time we return to load `A(2,1)`, which was in primary cache line 46 and secondary cache line 644 at the beginning of the code, it will no longer be loaded in either cache, and we will need to load it again from memory.

15.8 Array Mappings

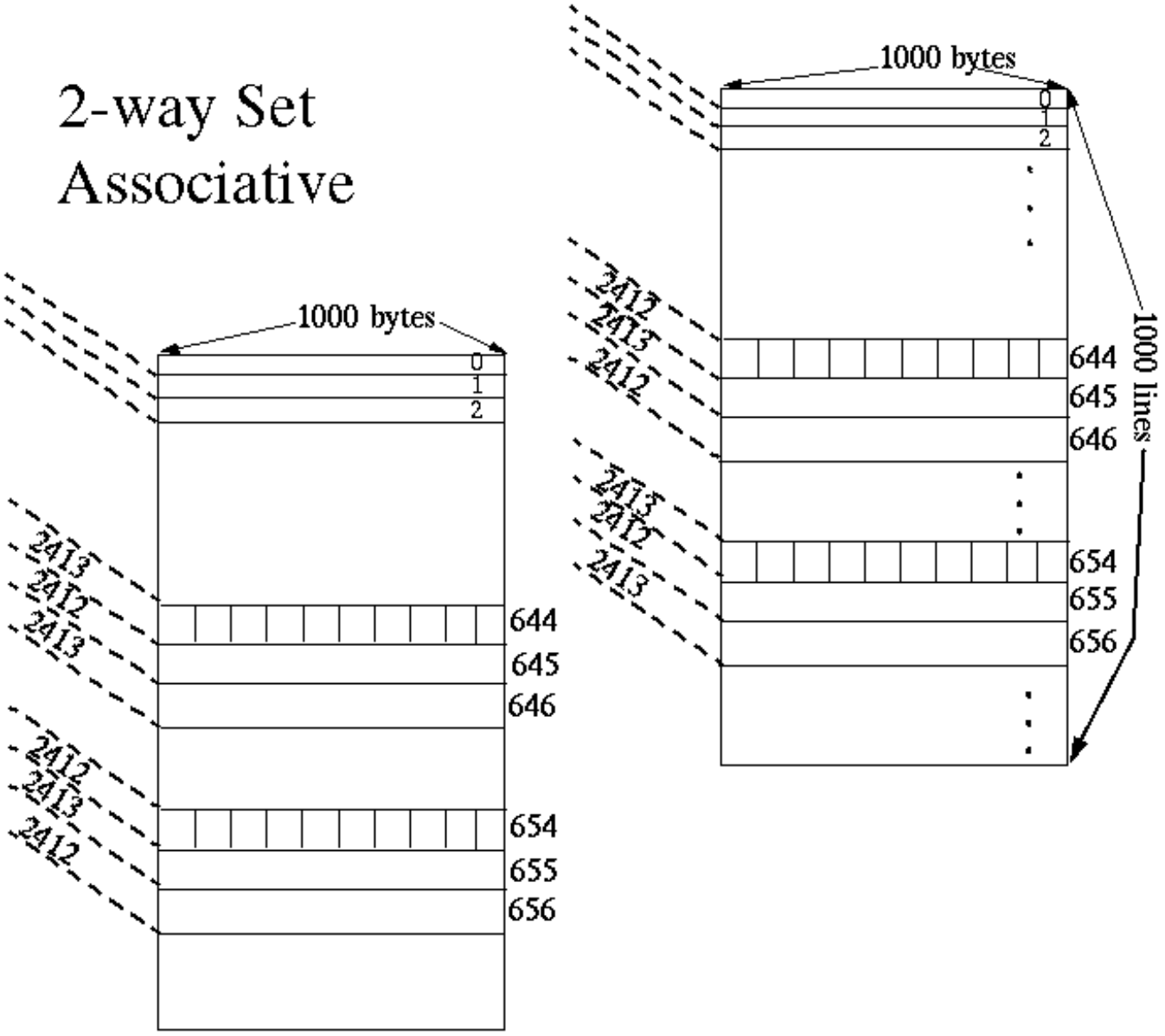


Notes:

If we go back and look at the other loads in the loop, we see that `B(i,j)` and `C(i,j)` both map to the same primary and secondary cache line as `A(i,j)` for all `i` and `j`. This mapping causes all loads of `A`, `B`, and `C` to constantly write over each other in cache, this is called thrashing. To eliminate thrashing, we can add a pad between the data objects, which has the effect of mapping the starting addresses of `A`, `B`, and `C` to different cache lines. Two secondary cache lines of padding should be enough to ensure that we have completely used all the data in the cache line before it is overwritten by a later load.

We know the exact addresses of `B` and `C` because of the nature of Fortran common blocks. In the `C` code, with `a`, `b`, and `c` being automatic arrays, their storage allocation may not be as bad as the Fortran allocation, but then again it might be as bad. Any Fortran data objects not allocated in the same common block would have the same unknown behavior. Because cache thrashing has such an adverse affect on performance, we may want to be more certain of our code's addressing scheme. One method would be to allocate all major data structures in one large block, then partition the data block so that the individual data objects have a starting address that is offset in cache.

15.9 2-Way Set Associativity



Notes:

In our example, we used a direct map cache. The Itanium 2 processors in SGI Altix systems use a four-way set associative primary cache, an eight-way set associative secondary cache and a 6- or 12-way (depending on size) set associative tertiary cache. These are many-to-*n* mappings. The mappings are the same as the direct mapped cache, but when the hardware looks to load an address, it must look in *n* places. On a cache miss in L3 cache, the hardware picks the least recently used (LRU) cache line for the load of new data.

15.10 Cache-Based Architecture

- Memory hierarchy:
  - Main memory: ≈130–700 nanoseconds
  - L2 DTLB miss →HPW (Hardware Page Walker): 25 or 31 cycle penalty, or OS trap
  - L1 DTLB miss: 4 cycle penalty if L2 DTLB hit
  - L3 cache: 12–17 clock cycles
  - L2 cache: 5–6 clock cycles
  - L1 cache: 1 clock cycle
  - Registers: 0 cycles
- Perform as many operations as are possible on a piece of data once it is in the registers
- Perform as many operations as are possible on a piece of data once it is in the cache
- Use the concepts of temporal and spatial locality

Notes:

A cycle is the reciprocal of the clock frequency; for the SGI Altix systems, it can be the reciprocal of 900, 1000, 1300 or 1500 MHz, i.e., 1.1111, 1, 0.7692 or 0.6666 nanoseconds, respectively.

15.11 Cache Design Details and Terminology

- Cache line: Discrete unit of data or instructions that is transferred, and into which cache and memory is segmented
- Cache hit: Data or instruction reference to a cache line that is currently in the cache
- Cache miss: Reference to a cache line that is not currently in cache and must be brought in
- Write-through: Cache change immediately updates memory
- Write-back: Memory is not changed until absolutely necessary (cache line is clean if not modified, dirty if modified but not written back)
- Prefetch: Ability to initiate a fetch prior to time data is actually needed
- Set associativity: Defines how memory is mapped into cache
    - One-way: Address in memory has one target cache location (also called direct-mapped cache)
    - N-way: Address in memory has N target cache locations

15.12 Libraries

- SCSL (Scientific Computing Software Library)

  BLAS, LAPACK, FFT and signal-processing routines, Sparse solvers
  `cc|efc|ecc -o program <objs> -lscs [-lm]`
  `cc|efc|ecc -i8 -o program <objs> -lscs_i8 [-lm]`
  `man intro_[scsl|blas|lapack|fft|solvers]`

- MKL (Intel Math Kernel Library)

  BLAS (`-lmkl_itp`), LAPACK (`-lmkl_lapack32`, `-lmkl_lapack64`)
  FFT (`-lmkl_itp`), Vector math library (`-lmkl_vml_itp`)

- Standard math libraries (`-lm` and `-limf`)

15.13 Some Intel Compiler Flags

- Relatively few tuning flags exposed
- `-O2` is default, which exposes some software pipelining
- `-O3` triggers more SWP, prefetching, loop reorganization, etc.
- `-ip[o]`: intra-file/inter-file interprocedural optimization
- `-prof_gen/-prof_use`: feedback training; useful if you have a good training set
- `-fno-alias, -fno-fnalias`: to indicate the use of pointers with no aliasing in C
- `-ftz`: flush underflows to zero, avoid kernel traps (enabled by default at `-O3`)

15.14 Other Useful Flags

- Generate an optimization report: `-opt_report`
- Generate assembly listing: `-S`
- Enable OpenMP support: `-openmp`
- Automatic parallelization: `-parallel`
- Verbose compiler driver output: `-v`
- For Fortran:
  - `-safe_cray_ptr` : no aliasing for Cray pointers
  - `-auto`: all local variables are automatic (`-auto_scalar` is default); required for thread safety
  - `-stack_temps`: allocate arrays on the stack, if possible; can sometimes help OpenMP performance; use with care and watch for stacksize problems

15.15 Let the Compiler Do the Work

- Ideally, a compiler should do all the work automatically
  - Cannot optimize for both in-cache and out-of-cache data
- Optimizations
  - Software pipelining
  - Inter-procedural analysis
  - Loop nest optimizations

Notes:

In reality, these options are not independent of one another. Some loop nest optimizations can enhance software pipelining, and interprocedural analysis can give the loop nest optimizer information enabling it to do its job better.

15.16 Software Pipelining

- Vectorizable loops well-suited to Software Pipelining (SWP)
- SWP may be slower for loops with only a few iterations
- SWP cannot be done if:
  - There are function calls in loop: use `-ip` or `-ipo` to foster inlining
  - Loop contains complicated conditionals or branching
- SWP impeded by:
  - Recurrences between iterations: use `IVDEP` directive
  - Loop body too large: solved by loop fission
  - Register overflow: solved by loop fission
- SWP algorithms are heuristic:
  - Schedules are not unique
  - Finding a schedule can be computationally intensive
- Use `-O2` or `-O3` to enable SWP (more SWP is attempted at `-O3`)

Notes:

The Intel Itanium 2 processors have a superscalar architecture capable of executing 6 instructions per cycle. Execution units may be filled by combinations of the following:

- Up to 6 instructions per cycle into 6 ALU, 2 Integer and 1 ISHIFT units
- Up to 2 loads and 2 stores or 4 loads, in the data cache unit
- Up to 6 instructions per cycle into 6 multimedia units
- Up to 1 instruction per cycle into 2 parallel shift, 1 parallel multiply or 1 popcnt units
- Up to 2 instructions per cycle into 2 FP multiply-add or 2 FP other operations units
- Up to 3 instructions per cycle into 3 branch units

In addition, all instructions are pipelined and have associated pipeline latencies ranging from one to dozens of cycles. It is the job of the compiler, through software pipelining, to determine how to best schedule instructions to fill up as many superscalar slots as possible.

15.17 Compiler Directives

Fortran:

`cdir$ ivdep`	no aliasing
`cdir$ swp`	try to software-pipeline
`cdir$ noswp`	disable software-pipelining
`cdir$ loop count (NN)`	hint for SWP
`cdir$ distribute point`	split this large loop
`cdir$ unroll (`*n*`)`	unroll *n* times
`cdir$ nounroll`	do not unroll
`cdir$ prefetch a`	prefetch array `a`
`cdir$ noprefetch c`	do not prefetch array `c`

15.18 Compiler Directives

C/C++:

`#pragma ivdep`	no aliasing
`#pragma swp`	try to software-pipeline
`#pragma noswp`	disable software-pipelining
`#pragma loop count (NN)`	hint for SWP
`#pragma distribute point`	split this large loop
`#pragma unroll (`*n*`)`	unroll *n* times
`#pragma nounroll`	do not unroll
`#pragma prefetch a`	prefetch array `a`
`#pragma noprefetch c`	do not prefetch array `c`

15.19 IVDEP Directive

Here, compiler flags are not enough:

```
void idaxpy(int n, double a, double *x, double *y, int *index)
{
   int i;
   for (i=0; i<n; i++) y[index[i]] += a * x[index[i]];
}
```

Use the `ivdep` (ignore vector dependencies) directive

```
#pragma ivdep
   for (i=0; i<n; i++) y[index[i]] += a * x[index[i]];
```

Without `ivdep`, the loop will have a poor SWP schedule or no SWP at all

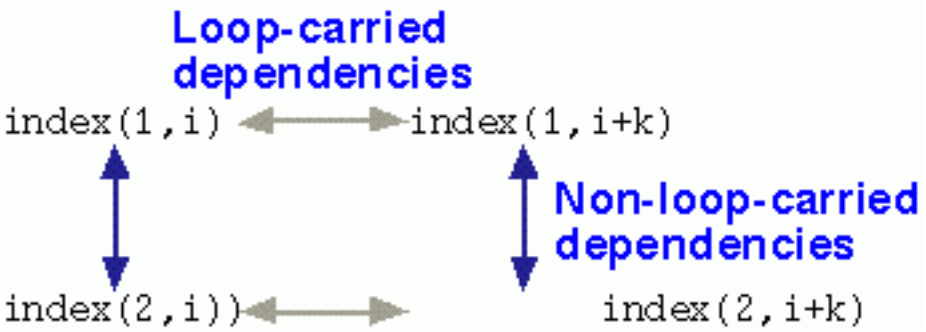
Notes:

When the code contains user-level indirection (the index array) whose values (memory addresses) are not known to the compiler, the `restrict` and `disjoint` flags do not help much. You must tell the compiler that there are actually no dependencies between iterations.

15.20 IVDEP Varieties

- Loop-carried dependencies:

```
do i = 1, n
   a(index(1,i)) = b(i)
   a(index(2,i)) = c(i)
enddo
```



- `-ivdep_parallel` tells the compiler that there are no loop-carried dependencies

```
for (i=0; i<n; i++)
   y[index[i]] += a * x[index[i]];
```

- Cray `ivdep` applies to `IDAXPY, but not here:`

```
for (i=0; i<=n; i++)
   a[i] = a[i-1] + 3;
```

Notes:

The `ivdep` directive applies to inner loops only. The directive also applies only to "loop-carried" dependencies. If `index(1,i)` = `index(1,i+k)` or `index(2,i)` = `index(2,i+k)` for some value of `k` such that `i+k` is in the range `1` to `n`, then there is a loop-carried dependency.

If `index(1,i)` = `index(2,i)` for some value of `i`, then there is a nonloop-carried dependency.

To determine whether a loop has lexically backwards loop-carried dependencies, the following checklist should be used. If for a given array referenced within a loop all four answers are "yes", then there is a lexically backwards loop-carried dependence with respect to that array:

- Does the loop contain two references to the same array?
- Is one of the references a store (i.e., does it appear to the left of an assignment)?
- Do index values of the references overlap in memory?
- Does the earlier reference have an earlier index with respect to the loop induction variable?

15.21 Inter-Procedural Analysis

- Analyzes the entire program
- Precedes other optimizations
- Performs optimizations across procedure boundaries
- Provides results to subsequent optimization phases so their optimizations benefit
- Easy to use, just add `-ip` or `-ipo` to both the compile step (`cc -ipo -c *.c ...`) and link step (`cc -ipo *.o ...`)
- Compile step will finish faster, link step will take longer
- If you change one file, you must relink (recompile)

15.22 Inter-Procedural Optimizations

- Function inlining
- Inter-procedural constant propagation of parameters and global variables
- Dead function elimination
- Dead variable elimination
- Automatic common block padding

15.23 Inlining

- Benefits
  - Exposes larger context to later optimization phases
  - Eliminates call overhead
  - Eliminates aliases and resolves otherwise unknown side-effects
- Disadvantages
  - Compilation time increases
  - Creates harder register allocation problem
  - Increases text size
- Restrictions
  - No `varargs` routines
  - No mismatched parameter types
  - No static (`save`) local variables
  - No recursive routines

Notes:

`varargs` routines are those with a variable argument list, as in

```
int printf (char *format, ...)
{
   ...
}
```

The `...` inside the parentheses specifies that the function takes a variable number of arguments.

15.24 Cache Basics

- Stride-1 accesses for block reuse

```
do i = 1, n                          do j = 1, m
   do j = 1, m                         do i = 1, n
      a(i,j) = b(i,j)        ⟶            a(i,j) = b(i,j)
   enddo                               enddo
enddo                                enddo
```

```
for (j=0; j<m; j++)                  for (i=0; i<n; i++)
   for (i=0; i<n; i++)        ⟶           for (j=0; j<m; j++)
      a[i][j] = b[i][j];                      a[i][j] = b[i][j];
```

Notes:

In the first loop, array `b` is copied to array `a` one row at a time. Because Fortran stores matrices in column-major and C/C++ store in row-major order, a new cache line must be brought in for each element of row `i`; this applies to both `a` and `b`. If `n` is large, the cache lines holding elements at the beginning of the row may be displaced from the cache before the next row is processed. In the second loop, the data is copied one column (Fortran) or row (C/C++) at a time. Thus, all data in the same cache line is accessed before the line can be displaced from the cache, so each line only needs to be loaded once.

15.25 Cache Basics (continued)

- Group together data used at the same time

```
 d = 0.0                          d = 0.0
 do i = 1, n                      do i = 1, n
   j=ind(i)                         j=ind(i)
   d=d+sqrt(x(j)*x(j)+    ⟶        d=d+sqrt(r(1,j)*r(1,j)+
&           y(j)*y(j)+           &           r(2,j)*r(2,j)+
&           z(j)*z(j))           &           r(3,j)*r(3,j))
 enddo                            enddo
```

```
d=0.0                             d=0.0
for (i=0; i<n; i++) {             for (i=0; i<n; i++) {
   j=ind[i];                         j=ind[i];
   d+= sqrt(x[j]*x[j]+    ⟶         d+= sqrt(r[j][1]*r[j][1]+
            y[j]*y[j]+                        r[j][2]*r[j][2]+
            z[j]*z[j]);                       r[j][3]*r[j][3]);
}                                 }
```

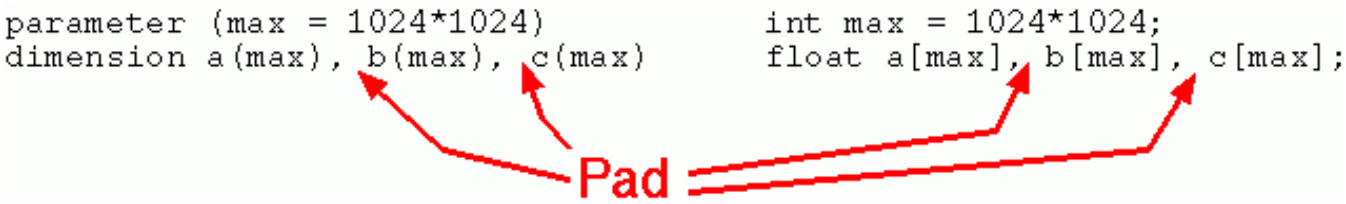
Notes:

The first loop accesses three arrays: `x`, `y`, and `z`. Access for these elements of the vectors is through the index array `ind`, so it is not likely that the accesses are stride 1. Thus, each iteration of the loop causes a new line to be brought into the cache for each of the three vectors.

In the second loop, since `r(1,j)`, `r(2,j)`, and `r(3,j)` (`r[j][1]`, `r[j][2]`, and `r[j][3]`) are contiguous in memory, it is likely that all three will fall in the same cache line. Thus, one cache line, rather than three, needs to be brought into the cache.

15.26 Cache Basics (continued)

- Avoid array sizes that are multiples of the cache size
- Use dummy pads



Notes:

Allocation of variables on the stack may vary from run to run, but the potential is here that these variables could be allocated in consecutive memory. Each vector is 4 MB in size; thus, the low 22 bits of the addresses of a_i, b_i, and c_i are the same, and the vectors map to the same cache line. To remedy this, introduce pads between the vectors to space their beginning addresses. Ideally, each padding variable should be at least the size of a full cache line. Thus, each pad should be 32 elements long.

Because the stack allocation is not predictable this solution may not work if the arrays are automatic, i.e., stack-allocated.

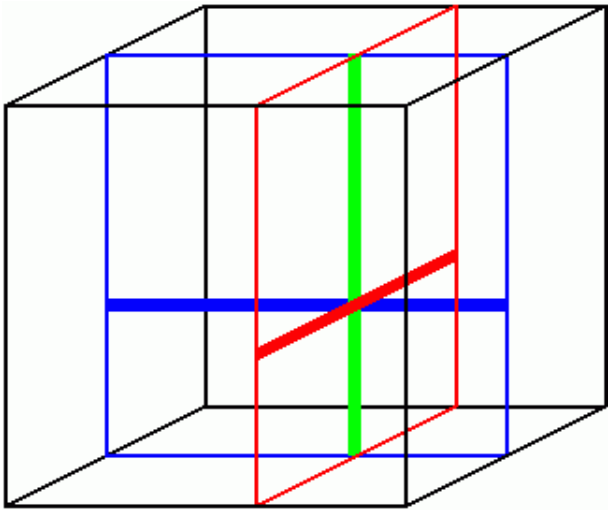
It is safer to allocate one large block of data for all the data arrays and the pad, then use pointers to make sure the starting address of the arrays don't line up on cache boundaries.

In Fortran, a common block will guarantee consecutive allocation of storage, but the compiler may decide to automatically insert padding between arrays in a given common block.

15.27 Use the Performance Monitor tools to Identify Cache Problems

- Example: ADI code

```
double precision a(256,256,256)
do k = 1, nz
  do j = 1, ny
    call xsweep(a(1,j,k),1,nx)
  enddo
enddo
do k = 1, nz
  do i = 1, nx
    call ysweep(a(i,1,k),ldx,ny)
  enddo
enddo
do j = 1, ny
  do i = 1, nx
    call zsweep(a(i,j,1),ldx*ldy,nz)
  enddo
enddo
```



- Poor cache performance

```
% histx -o out ./adi2
 Time:     12.771 seconds
 Checksum:  4.6021112569E+07
% sort -rn out.adi2.27974
              13998: *Total for thread 27974*
              13292: a.out:*Total*
               9773: a.out:zsweep_
               1709: a.out:ysweep_
               1594: a.out:xsweep_
                706: libscs.so:*Total*
                706: libscs.so:drand64_
                164: a.out:main
                 52: a.out:_init
```

Notes:

Operations are performed along “pencils” in each of the three dimensions. The PC sampling output is presented. This shows that the vast majority of time is spent in the routine zsweep. However, an analysis of the source code would indicate that xsweep, ysweep, and zsweep should ideally all take the same amount of time.

The difference between PC sampling and such ideal time is that, by taking samples of the program counter position as the program runs, PC sampling includes an accurate measure of the time associated with memory accesses. Ideal time simply counts the instructions that access memory and thus must tally the same amount of time for a cache hit as a cache miss. This means that the difference between PC sampling and ideal profiling results indicate which routines have cache problems.

15.28 Example: ADI Code

- First, try padding

```
dimension a(256,256,256)    ⟶    dimension a(257,257,256)
```

```
% histx -o out ./adi2
 Time:      5.463 seconds
 Checksum:  4.6021112569E+07
% sort -rn out.adi2.28114
                6506: *Total for thread 28114*
                5816: a.out:*Total*
                2388: a.out:zsweep_
                1613: a.out:ysweep_
                1592: a.out:xsweep_
                 689: libscs.so:*Total*
                 689: libscs.so:drand64_
                 147: a.out:main
                  76: a.out:_init
```

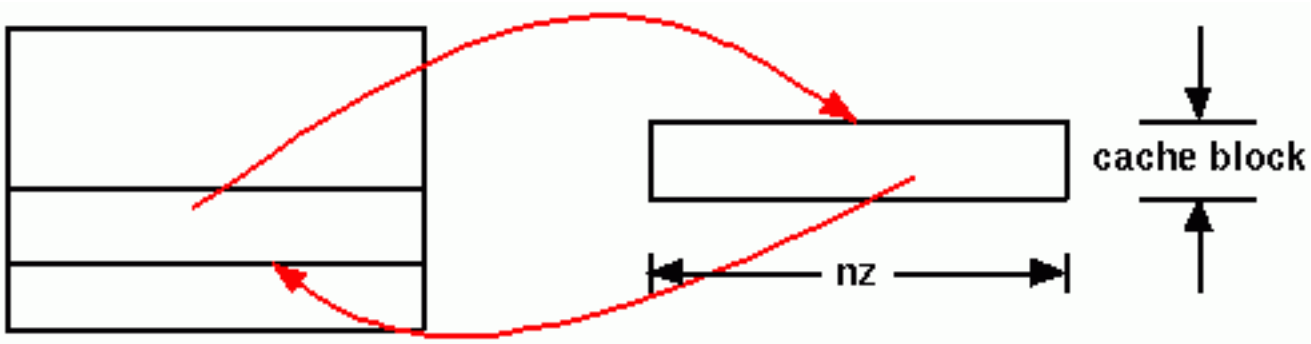
Notes:

The routine `zsweep` performs calculations along a pencil in the *z*-dimension. This is the index of the `a` array that varies most slowly in memory. The declaration for `a` shows that there are 256*256 = 65536 array elements, or 128 kB, between successive elements in the *z* dimension. Thus, each element along the pencil maps to one of two lines in the primary cache (because of associativity), and to one of 8 lines in the secondary cache.

In addition, every other element maps to one of 6 places in a 1.5-MB L3 cache, or one of 12 places in a 3 MB L3 cache. Since the associativity of the caches is limited, there will be cache thrashing, which you can fix by introducing (internal) padding—increase the size of the array to (257,257,256).

This fix provides sufficient padding with these dimensions, array elements along *z*-pencil all map to different secondary cache lines. The `histx` numbers above show that zsweep now takes just about 50% longer to execute as xsweep and yzweep (compared to a factor of 6 longer in the original version).

15.29 Example: ADI Code (continued)




```
parameter (nx=256, ny=256, nz=256)
parameter (ldx=257, ldy=257, ldz=256)
real*8 a(ldx,ldy,ldz), temp(ldx,ldz)
. . .
do j = 1, ny
  call copy(a(1,j,1),ldx*ldy,temp,ldx,nx,nz)
  do i = 1, nx
    call zsweep(temp(i,1),ldx,nz)
  enddo
  call copy(temp,ldx,a(1,j,1),ldx*ldy,nx,nz)
enddo
subroutine copy(from,lf,to,lt,nr,nc)
real*8 from(lf,nc), to(lt,nc)
do j = 1, nc
  do i = 1, nr
    to(i,j) = from(i,j)
  enddo
enddo
end
```

```
% histx -o out ./adi2
 Time:      5.197 seconds
 Checksum:  4.6021112569E+07
% sort -nr out.adi2.28372
               6235: *Total for thread 28372*
               5540: a.out:*Total*
               1613: a.out:ysweep_
               1593: a.out:xsweep_
               1387: a.out:zsweep_
                726: a.out:copy_
                695: libscs.so:*Total*
                694: libscs.so:drand64_
                166: a.out:main
                 55: a.out:_init
```

Notes:

Although TLB misses are not as big an issue as they are on SGI Origin systems, their effects may also be reduced. In the case of zsweep, where the large array is being traversed in the worst possible dimension, this can be accomplished by copying *z* pencils to a scratch array that is small enough to avoid TLB misses, carrying out the *z* sweeps on the scratch array, and then copying the results back to the `a` array.

15.30 Cache Blocking

- Example: matrix multiply

```
do j = 1, n
  do i = 1, m
    do k = 1, l
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

- Performance (on a SGI Origin 2000 with R10000@195MHz processors):

```
  m     n     l    lda   ldb   ldc    Seconds   MFLOPS
=======================================================
   30    30    30    30    30    30    0.000162  333.9
  200   200   200   200   200   200    0.056613  282.6
 1000  1000  1000  1000  1000  1000   25.431182   78.6
```

- Perfex output for 1000×1000 case:

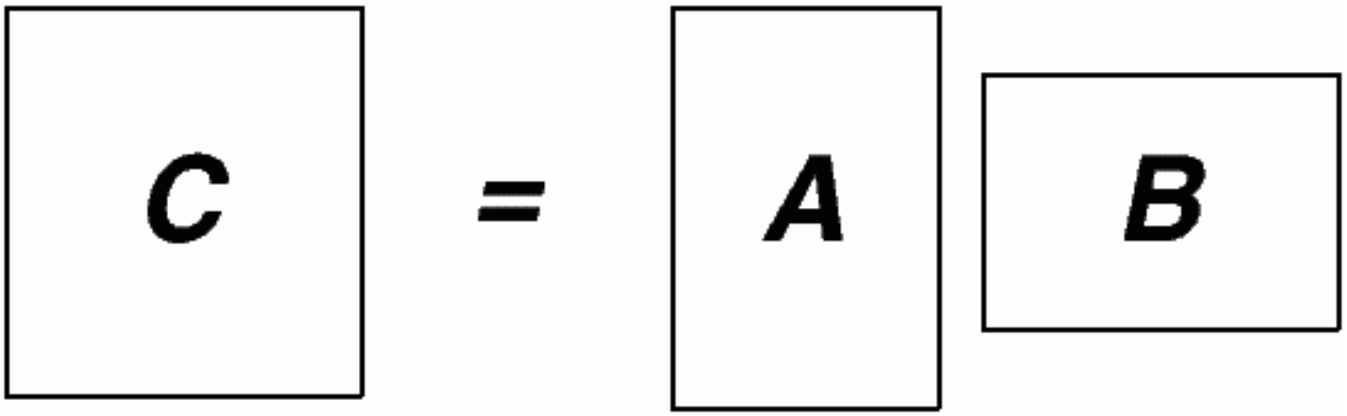
```
 0 Cycles..................... 5460189286 27.858109
23 TLB misses.................   35017210 12.164907
25 Primary data cache misses.....  200511309  9.217382
26 Secondary data cache misses...   24215965  9.141769
```

Notes:

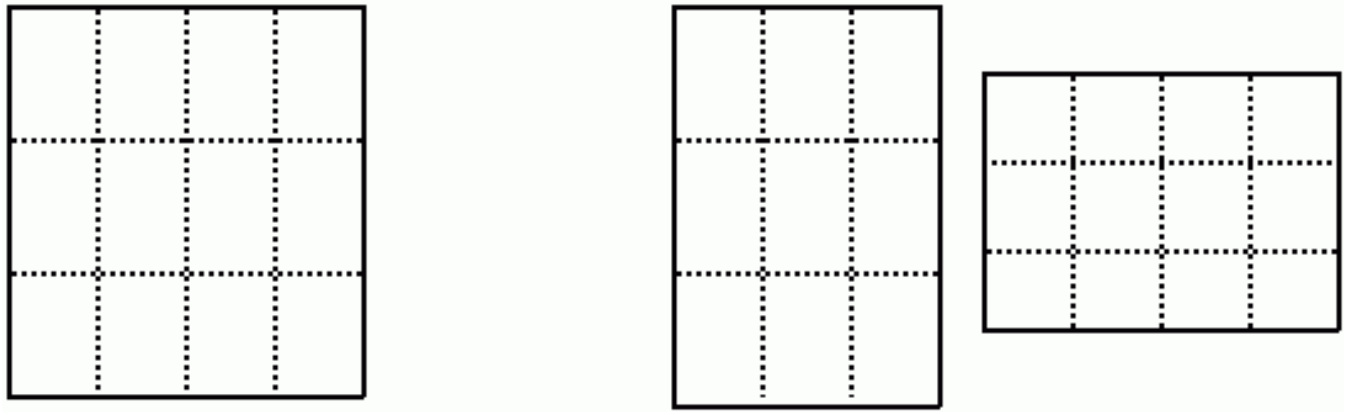
Cache blocking or tiling is the concept of breaking up data structures that are too big to fit in the cache. The example shows that when the matrices `A`, `B`, and `C` get bigger, the performance of the matrix multiply routine drops significantly. This problem with cache misses can be fixed by blocking the matrices.

15.31 Cache Blocking (continued)

- Example: matrix multiply



- Block matrices into cache-sized pieces

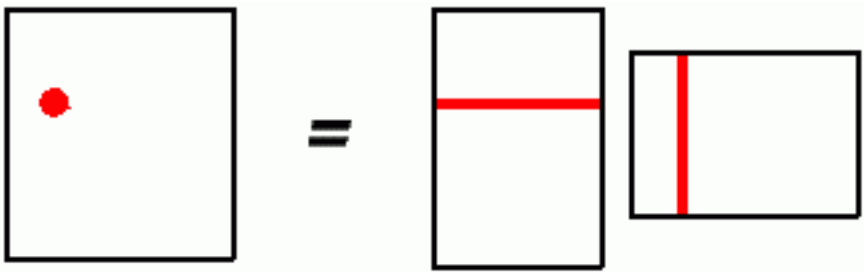


Notes:

Here, one block of C is updated by multiplying together a block of A and a block of B. The blocks must be small enough that one from each matrix can fit into the cache at the same time.

15.32 Why Does Blocking Work?

```
do j = 1, n
  do i = 1, m
    do k = 1, l
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    enddo
  enddo
enddo
```



Each element of C is touched once A is touched n times: once for each column of B B is touched m times: once for each row of A

If the matrices are large, each touch can cause a read from main memory, because previously read data will have been flushed from the cache.

If the matrices are blocked, and each block fits in the cache, the number of main memory accesses is reduced from the number of rows or columns to the number of blocks.

Notes:

When we count the number of lines each matrix is touched, we see that each `C(i,j)` is touched once; it is read from memory. Then the dot product of the row vector `A(i,:)` with the column vector `B(:,j)` is added to `C(i,j)`, and the result is stored back to memory. But calculating an entire column of `C` requires reading all the rows of `A`, so `A` is touched *n* times, once for each column of `C`, or equivalently, each column of `B`. Similarly, every column vector `B(:,j)` must be reread for each dot product with a row of `A`, so `B` is touched *m* times. If `A` and `B` don't fit in the cache, there is likely to be little reuse between touches, so these accesses will require streaming data in from memory.

When the matrices are blocked, a block of `C` is calculated by taking the dot product of a block-row of `A` with a block-column of `B`. The dot product consists of a series of submatrix multiplies. If three blocks—one from each matrix—fit in cache simultaneously, the elements of those blocks only need to be reading from memory once for each submatrix multiply. Thus, `A` will now only need to be touched once for each block-column of `C`, and `B` will only need to be touched once for each block-row of `A`. This reduces memory traffic by the size of the blocks.

15.33 Cache Blocking: 1000x1000 Matrix Multiply

For example, 65 columns fit in one set of a 1-MB cache:

- Blocked performance

```
                          block
  m    n    l   lda  ldb  ldc  order mbs nbs lbs Seconds MFLOPS
=============================================================
 1000 1000 1000 1000 1000 1000  ijk   65  65  65 6.837563 292.5
 1000 1000 1000 1000 1000 1000  ijk  130 130 130 7.144015 280.0
 1000 1000 1000 1000 1000 1000  ijk  195 195 195 7.323006 273.1
```

- Perfex output for 65×65 blocks:

```
0 Cycles................................ 1406325962 7.175132
21 Graduated floating point instructions.. 1017594402 5.191808
25 Primary data cache misses.............   82420215 3.788806
26 Secondary data cache misses...........    1082906 0.408808
23 TLB misses............................     208588 0.072463
```

- Loop Nest Optimizer:

```
% f77 -n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 matmul.f
  m    n    l   lda  ldb  ldc  Seconds   MFLOPS
=============================================
 1000 1000 1000 1000 1000 1000 7.199939   277.8
```

Notes:

The bigger the blocks, the greater the reduction in memory traffic, but they must be able to fit into cache simultaneously. Furthermore, the blocks cannot conflict with themselves; in other words, different elements of the same block cannot map to the same cache line. Conflicts can be reduced or eliminated by proper padding of the leading dimension of the matrices, careful alignment of the matrices in memory with respect to one another, and, if necessary, further limiting of the block size(s).

The larger the blocks, the greater the number of TLB entries that will be required to map their data. If the blocks are too big, the TLB cache will thrash. For large matrices, this means that the width of a block will be limited by the number of available TLB entries. The submatrix multiplies can be carried out so that the data accesses go down

the columns of two of the three submatrices and across the rows of only one submatrix. Thus, the number of TLB entries only limits the width of one of the submatrices.

For block sizes that limit cache reuse (1000×1000), the performance of the blocked algorithm matches the performance of the unblocked algorithm on a size that fits entirely in the secondary cache (200×200).

15.34 Loop Nest Optimizations (LNO)

- Improves cache and instruction scheduling by performing high-level transformations on loops
  - Loop interchange
  - Padding
  - Loop fusion
  - Cache blocking
  - Prefetching
  - + others
- Improves performance of numerical programs
- Default behavior is designed to be beneficial for most programs
  - Can be fine-tuned

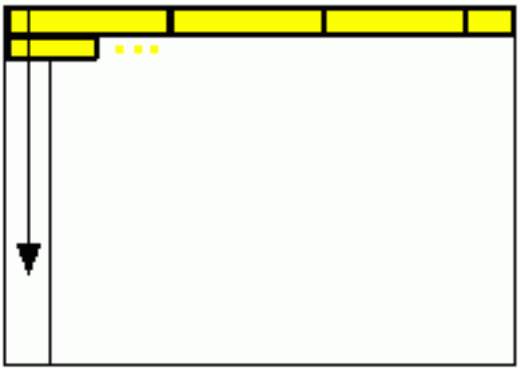
15.35 How to Run LNO

- Run at `-O3`
- Example: Matrix multiplication with wrong index order:

```
line 3: do j = 1, n
line 4:   do k = 1, n
line 5:     do i = 1, n
line 6:       c( i, j)= c( i, j)+ a( k, i)* b( k, j)
. . .
% efc -O3 -opt_report mtm.f
LOOP INTERCHANGE in mm_ at line 4
LOOP INTERCHANGE in mm_ at line 5
Block, Unroll, Jam Report:
Loop at line 3 unrolled and jammed by 4
Loop at line 4 unrolled and jammed by 4
```

15.36 Loop Interchange

```
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      a[j][i] = 0.0;
   }
}
```



- Interchange can minimize cache misses

```
for (j=0; j<n; j++) {
   for (i=0; i<n; i++) {
      a[j][i] = 0.0;
   }
}
```

- LNO at `-O3` can interchange loops automatically

15.37 Loop Interchange (continued)

- Need to consider more than caches

```
for (j=0; j<m; j++) {
   for (i=1; i<n; i++) {
      a[j][i]+ = a[j][i-1];
   }
}
```

  Nice cache behavior, but there is a recurrence

- What else can go wrong?

```
for (i=0; i<n; i++) {
   for (j=0; j<m; j++) {
      a[j][i] = 0.0;
   }
}
```

- What if `n`=2 and `m`=100?
  - Entire nest fits in cache
  - Loop interchange increases loop overhead

Notes:

When `n`=2 and `m`=100, the entire array fits in cache, and with the original loop order, the code achieves full cache reuse. Interchanging the loops, however, puts the shorter `i` loop inside. Software pipelining that loop incurs substantial overhead. So the compiler has made the wrong choice in this case.

15.38 Loop Interchange and Outer Loop Unrolling

```
for (j=0; j<m; j++) {
   for (i=0; i<n; i++) {
      a[j][i]+ = a[j][i-1];
   }
}
```

- Unrolling `j`-loop mitigates recurrence and preserves good cache behavior

Notes:

Loop interchange and outer loop unrolling can be combined to solve some performance problems that neither technique can solve on its own. Interchanging the above loop improves performance, but the recurrence on `i` limits software pipelining performance. But if the `j` loop is unrolled after interchange, the recurrence will be mitigated, because there will be several independent streams of work that can be used to fill up the processor's functional units.

```
for (j=0; j<m; j++ {
   for (i=0; i<n; i++) {
      a[j+0][i]+ = a[j+0][i-1];
        .
        .
        .
      a[j+3][i]+ = a[j+3][i-1];
   }
}
```

15.39 Example: Matrix Multiplication

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++)
      C[i*ldc+j] -= A[i*K+k] * B[j*ldb+k];
```

- Both A and B arrays accessed with unit stride in innermost loop, should get good performance
- Compile with `-O3 -fno-alias`
- Time for M = 10000 , N = 200 , K = 200 case: 19.1 sec, 402 MFLOPS

15.40 Matrix Multiplication: Hand-Unrolled Loops

```
for (i = 0; i < M- 3; i+= 4) {
  for (j = 0; j < N- 3; j+= 4) {
    t00 = 0; t10 = 0; t20 = 0; t30 = 0;
    t01 = 0; t11 = 0; t21 = 0; t31 = 0;
    t02 = 0; t12 = 0; t22 = 0; t32 = 0;
    t03 = 0; t13 = 0; t23 = 0; t33 = 0;
    for (k = 0; k < K; k++) {
      t00 += A[(i+0)*K+k] * B[(j+0)*ldb+k];
      t10 += A[(i+1)*K+k] * B[(j+0)*ldb+k];
      t20 += A[(i+2)*K+k] * B[(j+0)*ldb+k];
      t30 += A[(i+3)*K+k] * B[(j+0)*ldb+k];
    . . .}
    C[(i+0)*ldc+j+0] -= t00; C[(i+1)*ldc+j+0] -= t10;
    C[(i+2)*ldc+j+0] -= t20; C[(i+3)*ldc+j+0] -= t30;
  . . .}
}
```

- Compile with `-O3 -fno-alias`
- Time = 4.4 sec, 1820 MFLOPS

15.41 Matrix Multiplication: Fortran Version

```
do j = 1, n-3, 4
   do i = 1, m-3, 4
      t00 = 0.0
      t10 = 0.0
      t20 = 0.0
      t30 = 0.0
      . . .
         do k = 1, p
            t00 = t00 + A(k,j) * B(k,i)
            t10 = t10 + A(k,j+1) * B(k,i)
            t20 = t20 + A(k,j+2) * B(k,i)
            t30 = t30 + A(k,j+3) * B(k,i)
            . . .
         end do
      c(i,j) = c(i,j) - t00
      c(i,j+1) = c(i,j+1) - t10
      c(i,j+2) = c(i,j+2) - t20
      c(i,j+3) = c(i,j+3) - t30
   end do
end do
```

- Compile O3
- Time = 4.0 sec, 2000 MFLOPS

15.42 Stretching the Pipeline

- Compiler assumes loads will be satisfied from lowest level cache with minimal latency (e. g., 6 cycles for floating point)
  - Achievable with good prefetching
  - Good prefetching not always possible
- Can use `pfmon` to examine stall and latency events
  - e. g., `BE_EXE_BUBBLE.FRALL, DATA_EAR_CACHE_LAT8`
- Can try stretching the pipeline using *undocumented*, *unsupported* flag
  - `-mP3OPT_ecg_mm_fp_ld_latency=##`
  - Time for previous example with 20-cycle latency is 2.56 sec, 3123 MFLOPS
- Or use previously tuned code (a general rule of thumb)
  - Using SCSL DGEMM, example time is 2.2 sec, 3670 MFLOPS

15.43 Loop Fusion

- LNO automatically fuses loops
  - In general, may need to peel iterations:

```
do i = 1, n
  a(i) = 0.0
enddo
do i = 2, n-1
  c(i) = 0.5 * (b(i+1) + b(i-1))
enddo
```

↓ LNO does this automatically

```
a(1) = 0.0
do i = 2, n-1
  a(i) = 0.0
  c(i) = 0.5 * (b(i+1) + b(i-1))
enddo
a(n) = 0.0
```

Notes:

The first and last iterations of the first loop are peeled off so that you end up with two loops that run over the same iterations; these two loops may then trivially be fused.

15.44 Loop Fission

- Fission relieves register pressure:

```
for (j=0; j<n; j++) {
   ...
   s = ...
   ...
   ... = s
   ...
}
```

$\longrightarrow$

```
for (j=0; j<n; j++) {
   ...
   se[j] = ...
   ...
   }
for (j=0; j<n; j++) {
   ...
   ... = se[j]
   ...
}
```

Notes:

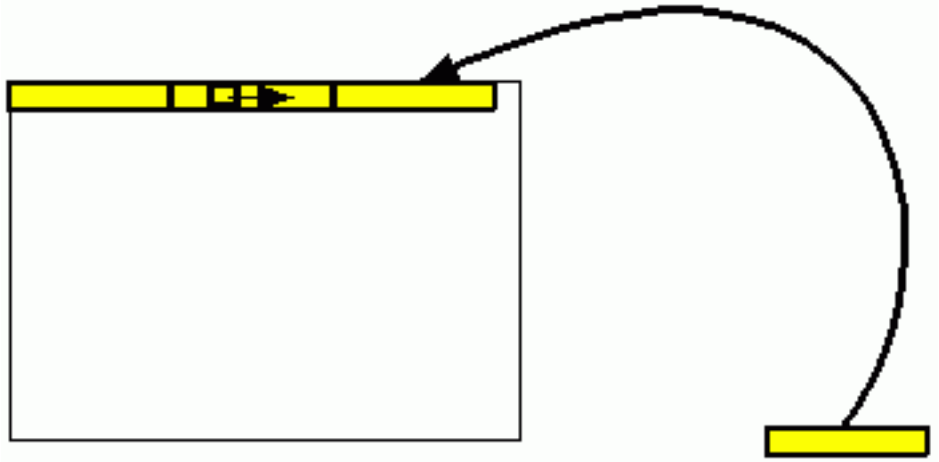
Although loop fusion generally improves cache performance, it has one potential drawback: it makes the body of the loop larger. Large loop bodies place greater demand on registers making software pipelining more difficult.

15.45 Prefetching

```
for (i=0; i<n; i++) { /* assume b is double */
   a += b[i];
}
```

Every secondary miss will stall the machine completely

```
for (i=0; i<n; i++) {
   prefetch b[i+16];
   a += b[i];
}
```



Tolerates much of the miss latency, but doubles the number of loads

Notes:

The Itanium 2 processor instruction set contains prefetch instructions that can move data from main memory into cache in advance of their use. This movement allows some or all of the latency required to access the data to be hidden behind other work. One optimization the LNO performs is to insert prefetch instructions into a program. In the modified code above, `prefetch` is not a C statement; it indicates that a prefetch instruction for the data at the specified address is issued at that point in the program. If `b[]` is a double array, the address `b[i+16]` is one full cache line (128 bytes) ahead of the address that is being read in the current iteration. With this `prefetch` instruction inserted into the loop, each cache line is prefetched 16 iterations before it needs to be used. Now each iteration of the loop takes two cycles since it is load bound, and the prefetched instruction and the load of `b[i]` each take one cycle.

Thus, cache lines will be prefetched 32 cycles in advance of the use. A cache miss to local memory takes approximately 60 to 100 cycles, so half to a third of this latency will be overlapped with work on the previous cache line. As written, the above loop issues a `prefetch` instruction for every load. For out-of-cache data, this is not bad, because you still have to wait for half the latency of the cache miss. But if this same loop were run on in-cache data, the performance would be half of what it could be.

15.46 Prefetching (continued)

```
for (i=0; i<n; i++) {
   if ((i % 16) == 0) prefetch b[i+16];
   a += b[i];
}
```

Conditional is worse than extra prefetches

```
for (i=0; i<n; i+=4) {
   prefetch b[i+32];
   a += b[i+0];
   a += b[i+1];
   a += b[i+2];
   a += b[i+3];
}
```

Improvement:

- Prefetches two cache lines ahead
- Fewer redundant prefetches

But not ideal unless unrolled by 16

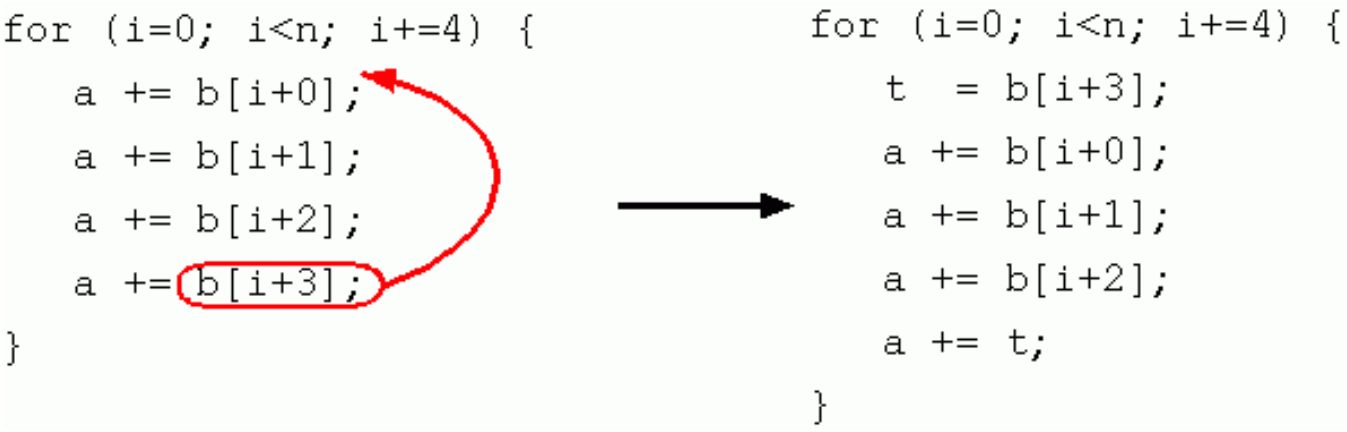
Notes:

With the unrolled loop, you reduce the number of prefetches by a factor of four. In fact, unrolling the loop by 16 eliminates all redundant prefetches. LNO automatically inserts prefetched instructions and uses unrolling to limit the number of redundant prefetch instructions it generates. Generally, LNO will not unroll the loop enough to eliminate the redundancies completely since the extra unrolling, look too much loop fusion, can have a negative affect on software pipelining. So, in prefetching, as in all its other optimizations, the LNO tries to achieve a delicate balance.

Instead of prefetching just one cache line, LNO prefetches two or more lines ahead so that most, if not all, of the memory latency is overlapped.

15.47 Pseudo Prefetching

- Prefetching from secondary into primary cache



- No instruction overhead, but increased register pressure

15.48 Prefetch Example 1: CFD Solver

```
      DO 33 IP= 1, NCELL
        IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
        IS=LQ(3,IP); IT=LQ(2,IP); IB=LQ(1,IP)
        WSP= WS(IP)
        IF(IB.GT.0 .AND. IB.LT.IP) WSP=WSP+DBLE(AC(1,IP))*WS(IB)
        IF(IT.GT.0 .AND. IT.LT.IP) WSP=WSP+DBLE(AC(2,IP))*WS(IT)
        IF(IS.GT.0 .AND. IS.LT.IP) WSP=WSP+DBLE(AC(3,IP))*WS(IS)
        IF(IN.GT.0 .AND. IN.LT.IP) WSP=WSP+DBLE(AC(4,IP))*WS(IN)
        IF(IW.GT.0 .AND. IW.LT.IP) WSP=WSP+DBLE(AC(5,IP))*WS(IW)
        IF(IE.GT.0 .AND. IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
        IF(LARBE) THEN
           . . .
        ELSE    ! LARBE = .FALSE.
           . . . more if-then for WSP computation
        ENDIF
        WS2(IP)= WSP* DBLE(D(IP))
   33 CONTINUE
```

- Compiled with `-O3`, time is 9.25 seconds

15.49 Prefetch Example 1 (continued)

- Split into two loops

```
      IF(LARBE) THEN
       DO 33 IP= 1, NCELL
        IE= LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
        . . .
        IF(IE.GT.0 .AND. IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
        . . . more if-then for WSP computation
        WS2(IP)=WSP*DBLE(D(IP))
   33  CONTINUE
      ELSE ! LARBE = .FALSE.
       DO 34 IP= 1, NCELL
        IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
        . . .
        IF(IE.GT.0 .AND. IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
        WS2(IP)=WSP*DBLE(D(IP))
   34  CONTINUE
      ENDIF
```

- Compiled with `-O3`, time is 7.85 seconds

15.50 Prefetch Example 1 (continued)

- Add a `prefetch` directive

```
      DO 33 IP= 1, NCELL ! NCELL = 156000 -> 10.5 MB
cdir$ prefetch LQ, AC
      IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
      IS=LQ(3,IP); IT=LQ(2,IP); IB=LQ(1,IP)
      WSP= WS(IP)
      IF(IB.GT.0 .AND. IB.LT.IP) WSP=WSP+DBLE(AC(1,IP))*WS(IB)
      IF(IT.GT.0 .AND. IT.LT.IP) WSP=WSP+DBLE(AC(2,IP))*WS(IT)
      IF(IS.GT.0 .AND. IS.LT.IP) WSP=WSP+DBLE(AC(3,IP))*WS(IS)
      IF(IN.GT.0 .AND. IN.LT.IP) WSP=WSP+DBLE(AC(4,IP))*WS(IN)
      IF(IW.GT.0 .AND. IW.LT.IP) WSP=WSP+DBLE(AC(5,IP))*WS(IW)
      IF(IE.GT.0 .AND. IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
      WS2(IP)=WSP*DBLE(D(IP))
   33 CONTINUE
```

- Compiled with `-O3`, time is still 7.85 seconds

15.51 Prefetch Example 1 (concluded)

- Insert explicit `prefetch` calls

```
      DO 33 IP= 1, NCELL ! NCELL = 156000 -> 10.5 MB
      call lfetch_nta(LQ(1,IP+14))
      call lfetch_nta(AC(1,IP+14))
      IE=LQ(6,IP); IW=LQ(5,IP); IN=LQ(4,IP)
      IS=LQ(3,IP); IT=LQ(2,IP); IB=LQ(1,IP)
      WSP= WS(IP)
      IF(IB.GT.0 .AND. IB.LT.IP) WSP=WSP+DBLE(AC(1,IP))*WS(IB)
      IF(IT.GT.0 .AND. IT.LT.IP) WSP=WSP+DBLE(AC(2,IP))*WS(IT)
      IF(IS.GT.0 .AND. IS.LT.IP) WSP=WSP+DBLE(AC(3,IP))*WS(IS)
      IF(IN.GT.0 .AND. IN.LT.IP) WSP=WSP+DBLE(AC(4,IP))*WS(IN)
      IF(IW.GT.0 .AND. IW.LT.IP) WSP=WSP+DBLE(AC(5,IP))*WS(IW)
      IF(IE.GT.0 .AND. IE.LT.IP) WSP=WSP+DBLE(AC(6,IP))*WS(IE)
      WS2(IP)=WSP*DBLE(D(IP))
   33 CONTINUE
```

- Compiled with `-O3`, time is now 4.55 seconds

15.52 Prefetch Example 2: Radix-3 FFT kernel

```
      do 101 k = 1, l1
      tr2       = cc(1,2,k) + cc(1,3,k)
      cr2       = cc(1,1,k) + taur * tr2     ! 5 loads
      ch(1,k,1) = cc(1,1,k) + tr2            ! 6 stores
      ti2       = cc(2,2,k) + cc(2,3,k)      ! 12 fp inst
      ci2       = cc(2,1,k) + taur * ti2
      ch(2,k,1) = cc(2,1,k) + ti2
      cr3       = taui * (cc(1,2,k) - cc(1,3,k))
      ci3       = taui * (cc(2,2,k) - cc(2,3,k))
      ch(1,k,2) = cr2 - ci3
      ch(1,k,3) = cr2 + ci3
      ch(2,k,2) = ci2 + cr3
      ch(2,k,3) = ci2 - cr3
  101 continue
```

- Compiled with `-O3`, it runs in 13 cycles/ iteration
- Adding `cdir$ noprefetch`, the loop runs in 10 cycles/iteration
- Compiling with `-O3 -mP3OPT_ecg_mm_fp_ld_latency=13` and no directive, the loop also runs in 10 cycles/iteration
- Compiling with both the directive and the options, the loop runs in 7 cycles/iteration

15.53 What Can You Do?

- Avoid equivalences
- Avoid `goto`’s
- Keep commons consistent
- Do not hand unroll
    - Unless you do not want the compiler to unroll
- Do not violate the Fortran standard
    - For example, don’t do out-of-bounds indexing

15.54 Intel Compiler Summary

Intel `ecc`/`efc` compilers are EPIC- and Itanium® 2 processor-aware:

- Just- in- time scheduling
- Good loop unrolling and pipelining
- Implicit and explicit prefetching
- Efficient use of predication
- Hardware instruction intrinsics
- Few tuning knobs
- SGI is working with Intel to improve quality and performance

Lab: CPU Tuning (Fortran and C)

- Use the `lipfpm`, `pfmon`, `histx` and `profile.pl` utilities (various options) to time the code: *Altix/Single_CPU_tuning/labs/[f|c]src/matmult.[fc]*
  - Record the time and the output from the verify routine.
- Use the optimization techniques from this module to improve the run time.

  This might take a little time because you will be trying a variety of techniques. Go through the notes and try different methods, some of which are suggested below:

  - Try changing the order of the loops.
  - Try unrolling the inner loop.
  - Try unrolling an outer loop.
  - Try using a library routine (`man intro_scsl`).
  - Use compiler options to automatically perform some of the above optimizations.

Module 16

SGI Altix Systems I/O Usage and Performance

16.1 Module Objectives

After completing this module, you will be able to

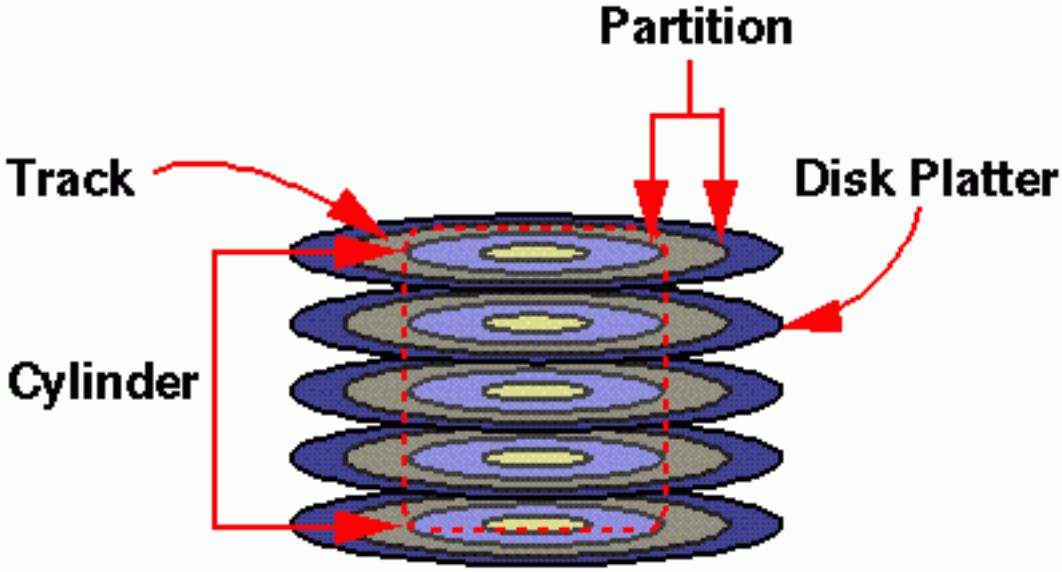
- Find I/O bottlenecks
- Tune I/O bottlenecks

16.2 I/O Terminology

- Buffer
  - Hold data temporarily to deliver at rate different from received rate
  - Managed by the application or I/O libraries
- Cache
  - Fast storage area with maximum bandwidth to the program
  - Used for reuse of data and instructions
- Types of cache on an SGI Altix system
  - Hardware
    - Primary (L1)
    - Directory memory for CC-NUMA
    - Secondary (L2)
    - Tertiary (L3)
    - Disk
  - Software
    - Kernel filesystem buffer cache
    - CacheFS

16.3 Disk Characteristics

- Disk controller interface
- Rotational speed in rpm
- Sector, track, cylinder geometry
- Disk storage capacity
- Seek time
- Latency
- Crossover
- Bandwidth



16.4 Finding I/O-Intensive Code

- `time` (shell builtin), `/usr/bin/time` show
  - *`real`* time is much greater than *`user+sys`* time
  - *`sys`* time may be fairly large
- Profiling shows
  - PC sampling and user time callstack profiling may show time spent in I/O routines
  - Warning: Much of I/O time is spent sleeping, so it will not show up in profiles
- `gtop/sar` shows
  - CPU waiting on I/O, and CPU spending time in system calls
  - Terminal I/O may be shown as time spent in graphics, if I/O is in a graphics window
  - Watch for buffer reads/writes

16.5 Special Profiling Considerations

- Most of the time spent in I/O is not CPU time
- Writes to I/O are not completely synchronous by default
- Process sleeps only while transfer is being set up, then continues
- You can see buffer activity using `sar`
- Explicit synchronous I/O (file opened with `O_SYNC`)
- Process sleeps until I/O is complete
- You can see CPU wait on I/O using `sar`
- You can see buffer activity using `sar`
- Some writes to I/O are completely asynchronous
- Process never sleeps

16.6 I/O Tuning Techniques

- Use fast I/O calls
- Do I/O in page-sized chunks
- Align data on page boundaries

16.7 Multiprocessor I/O

- One file
- All processors open, read and write
- Global file position changes, non-deterministic for sequential reads and writes
- Use Direct Access Files
- Record number based on PE number
- Record size must be multiple of physical data block

16.8 Use Fast I/O Calls

- Unformatted I/O is many times faster than formatted I/O
- System calls for I/O are more efficient than language calls for large contiguous chunks of data
- Use `read()` and `write()` rather than `fread()` and `fwrite()` for data chunks 8 kB or bigger

16.9 Page Alignment

- Filesystem is automatically page aligned
- Align array memory to pages if copying data to disk
- Use `getpagesize()` to find proper size for alignment

16.10 System Level I/O Techniques

I/O Type	Description	Purpose
UNIX `read`/`write`	Normal UNIX I/O methods	This is the standard way that processes obtain data from the filesystem
Memory mapped files	Map contents of file directly into process address space memory	Bypass `read()` operations and access disk files as though it was regular
Asynchronous (POSIX 1003.4)	Allow process to continue without waiting for operations to complete	Keep I/O operations from delaying process execution
Direct	Copy data directly from process space to disk	Avoid delay of copying data to kernel and having the buffer flushed to disk at a later time

Lab: I/O Tuning

Objective

Compare different methods of I/O transfer

Module 17

Thinking Parallel

17.1 Module Objectives

By the end of this module, you should be able to

- Understand the effects of parallelization on functionality and performance of applications
- Choose suitable parallelization approaches to a problem

17.2 Thinking Parallel

- Placement of data
- Choice of algorithm
- Number of processors
- Size of data
- Coordination of processors

17.3 Time to Solution

$$T_{\text{SOLUTION}} = T_{\text{COMPUTE}} + T_{\text{COMMUNICATION}}$$

$$T_{\text{COMPUTE}} = T_{\text{OPS}} + T_{\text{DATA ACCESS}}$$

$$T_{\text{COMMUNICATION}} = T_{\text{WORK DISTRIBUTION}} + T_{\text{DATA DISTRIBUTION}} + T_{\text{SYNC}}$$

17.4 Compute Time

$$T_{\text{COMPUTE}} = T_{\text{OPS}} + T_{\text{DATA ACCESS}}$$

$$T_{\text{OPS}} = T_{\text{SERIAL}} + T_{\text{PARALLEL}}/P$$

T_{OPS} decreases as the number of processors P increases (with limit T_{SERIAL})
$T_{\text{DATA ACCESS}}$ can vary greatly depending on placement of data
On the SGI Altix systems data may reside in

- Registers
- Primary D-cache
- Secondary cache
- Tertiary cache
- Local DRAM
- Remote DRAM

$$T_{\text{DATA ACCESS}} = T_{\text{LOCAL}}/P + T_{\text{REMOTE}}$$

If data is local, data access time decreases as P increases.
If data is remote, access adds a new term to T_{COMPUTE}

17.5 Parallel Overhead

$$T_{\text{COMMUNICATION}} = T_{\text{WORK DISTRIBUTION}} + T_{\text{DATA DISTRIBUTION}} + T_{\text{SYNC}}$$

- Depends on speed of supporting hardware
- May be explicit or implicit depending on programming model
- May increase with the number of processors, P
- $T_{\text{WORK DISTRIBUTION}}$
  - Coarse distribution
    - Less overhead
    - Potentially less parallel benefit
  - Fine-grain distribution
    - Higher overhead
    - Higher parallel potential
- $T_{\text{DATA DISTRIBUTION}}$
  - May overlap T_{COMPUTE}
  - Cost of redistribution could be offset by a reduced $T_{\text{DATA ACCESS}}$
- T_{SYNC}
  - Processors go idle
  - Required for correct answers

17.6 Limits on Parallel Processing Performance

- Summarized in Amdahl's Law:

$$S_P = \frac{1}{(1 - f_p) + \frac{f_p}{P}}$$

- S_P: Maximum, theoretical, ideal wall-clock speedup on P processors
- f_p: fraction of the single-processor running time that is parallelizable
- $(1 - f_p)$: fraction of the single-processor running time that is serial
- P: Number of processors

17.7 Limits on Parallel Processing Performance (continued)

f_p \ P	2	4	8	32	64	256	512	1024
0.700	1.54	2.11	2.58	3.17	3.22	3.30	3.32	3.36
0.800	1.67	2.50	3.33	4.44	4.71	4.92	4.96	4.99
0.850	1.74	2.76	3.90	5.66	6.12	6.52	6.59	6.63
0.900	1.82	3.08	4.71	7.81	8.77	9.66	9.83	9.91
0.950	1.91	3.48	5.93	12.55	15.42	18.62	19.28	19.63
0.970	1.94	3.67	6.61	16.58	22.15	29.60	31.35	32.31
0.990	1.98	3.88	7.48	24.43	39.26	72.11	83.80	91.18
0.999	2.0	3.99	7.94	31.04	60.21	203.98	338.85	506.18

17.8 Effective Parallel Programs

$$lim_{m \to \infty}(1 - f_p) = 0$$

$$lim_{m \to \infty} S_P = P$$

- m: Problem size
- $(1 - p)$: Serial fraction in a code
- P: Number of processors
- This means you “reduce” the serial fraction as you increase the problem size (for example, more grid points)
- Observation: This seems to be valid for a large number of important problems

17.9 Parallel Algorithms

You may be able to get an increase in the parallel fraction of the code by choosing a parallel, or a more highly scalable parallel algorithm.

- Algorithm selection criteria:
  - Number of calculations required
  - Memory required
  - Ease of understanding/coding/maintaining
- Additional criteria:
  - Percentage of parallel calculations
  - Communications required
  - Scalability Granularity

17.10 Choosing an Algorithm

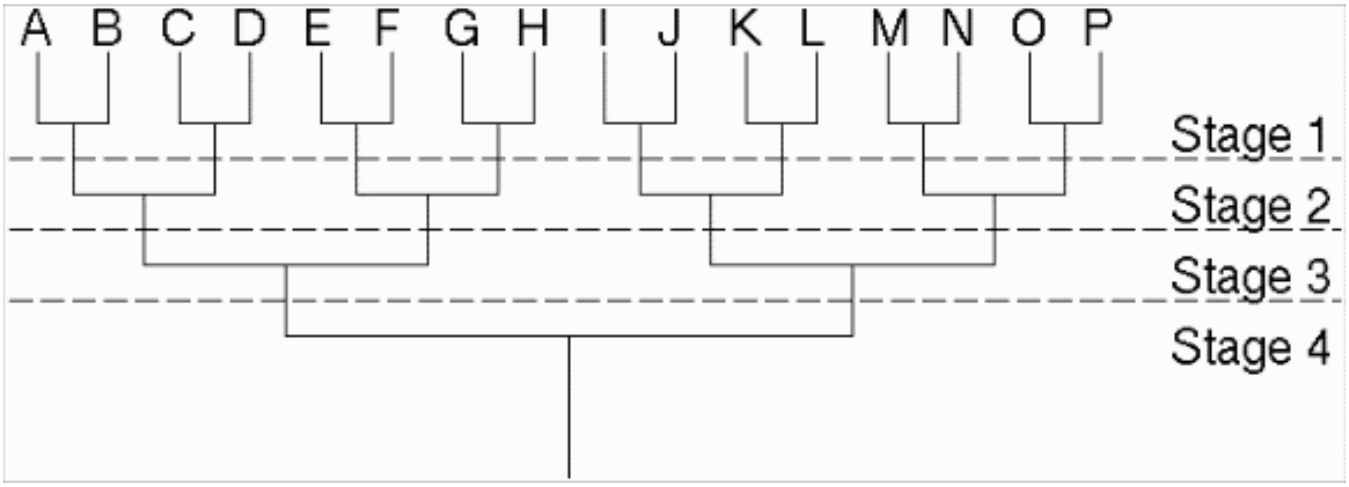
Consider this straightforward method of finding the maximum value in an array of numbers:

```
IMAX = 1
MAX = A(1)
DO I = 2, N
  IF (A(I).GT. MAX) THEN
     MAX = A(I)
     IMAX = I
  ENDIF
ENDDO
```

- The number of comparisons grows with N ($O(N)$)
- Memory required grows with N ($O(N)$)
- Each iteration of the loop requires the results of the previous pass

17.11 Choosing an Algorithm (continued)

Consider a slightly more complex algorithm:



- Number of calculations is $P\ O(N/P)$
- Size of memory is $O(N)$

At any stage of this algorithm, blocks of N/P calculations are independent and can be done in parallel

17.12 Choosing an Algorithm (continued)

Because of the nature of this problem, you could run the first (simpler) algorithm in each processor on a portion of the data set, creating P (number of processors) potential maximums.

- Using the first algorithm to then find the true maximum across all processors would require:
  - $O(N/P) + O(P)$ calculations in $O(N/P) + O(P)$ time
  - $O(P)$ messages in $O(P)$ time
- Using the second algorithm would require:
  - $O(N/P) + O(P)$ calculations in $O(N/P) + O(\log_2 P)$ time
  - $O(P)$ messages in $O(\log_2 P)$ time

17.13 Data Scoping

To run the `i` loop in parallel, which arrays should be scoped as shared, which ones should be private?

```
do i = 1 , n
   do j = 1 , m
      X (j) = A (j,i) * B (j,i+1)
   enddo
   do j = 1 , mm
      C (j,i) = X (j) + C (j,i) + ...
   enddo
enddo
```

17.14 Calculations Versus Communications

In the code segment below, should you declare X, Y, and Z as private or shared?

```
X = sin (alpha)** 2
Y = cos (alpha)** 2
Z = tan (alpha)** 2
do i = 1 , n
   A (i) = B (i) * X ...
   C (i) = D (i) * Y ...
   E (i) = F (i) * Z ...
enddo
```

17.15 Dependency Analysis

To run the k loop in parallel, which arrays should be shared?

```
do k = 1 , kmax
   do j = 1 , jmax
      X (j,k) = X (j,k) + H (j)
   enddo
   do j = 1 , jmax
      H (j) = H (j) + 1.0
   enddo
enddo
```

17.16 Global Updates

What is the best way to obtain the global sum?

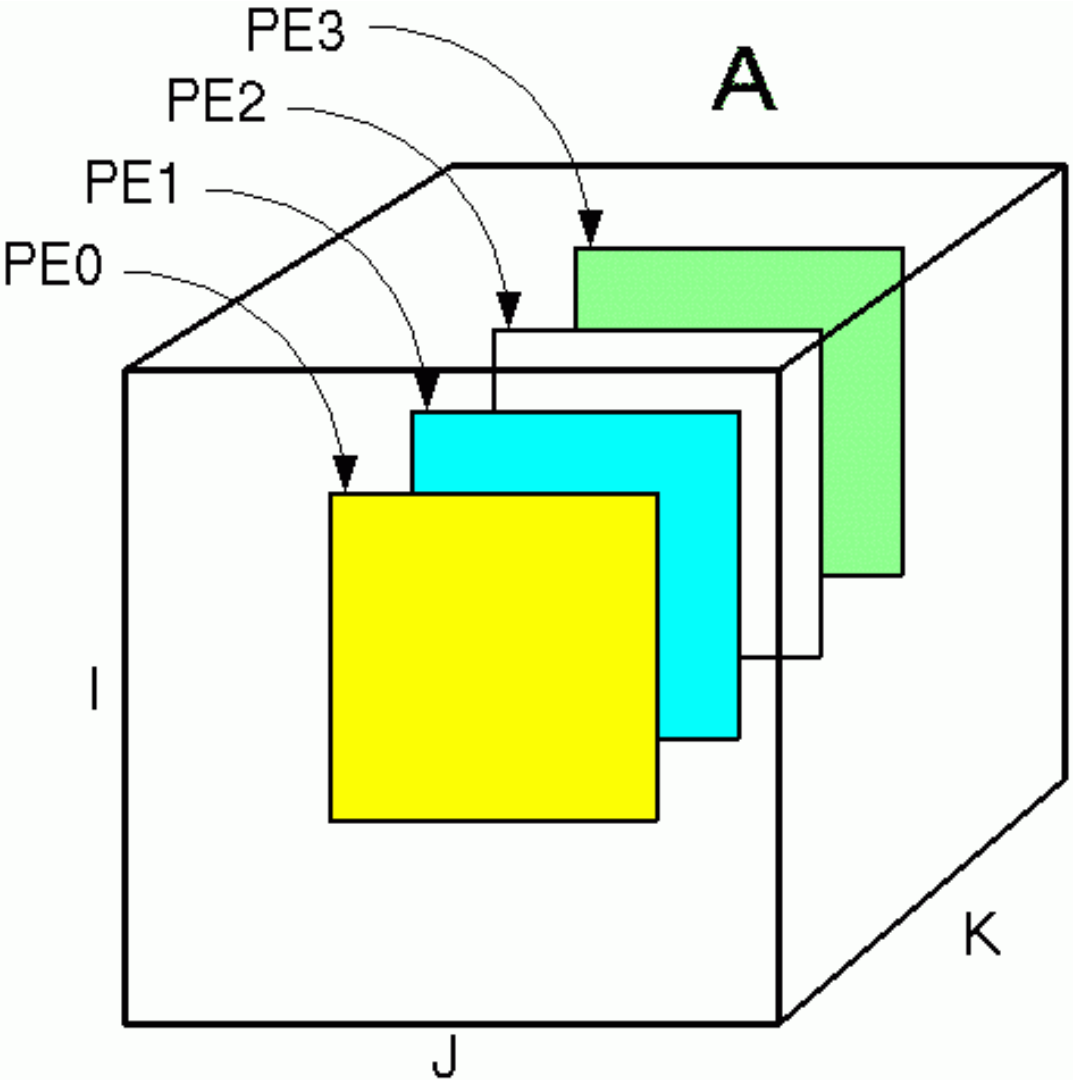
```
S = 0.0
do i = 1 , n
   S = S + A (i) * B (i)
enddo
```

17.17 Finding Parallelism

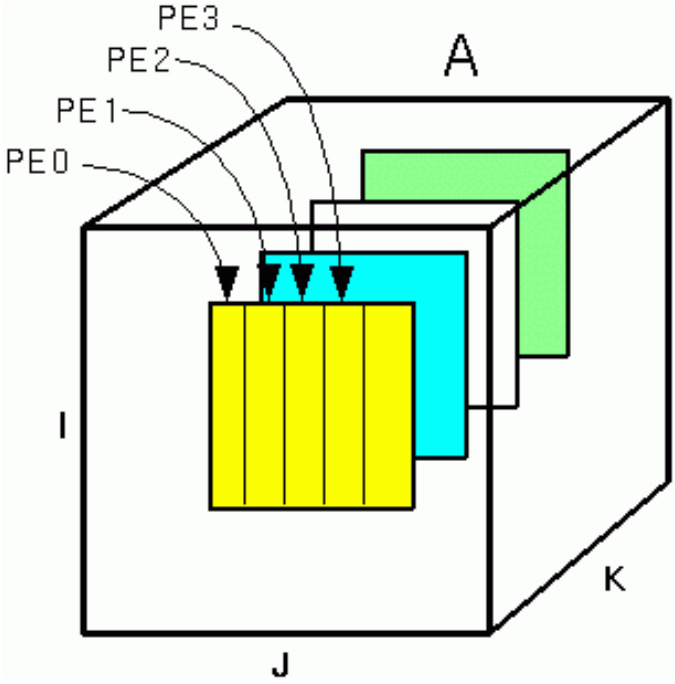
Which loop(s) should be run in parallel?

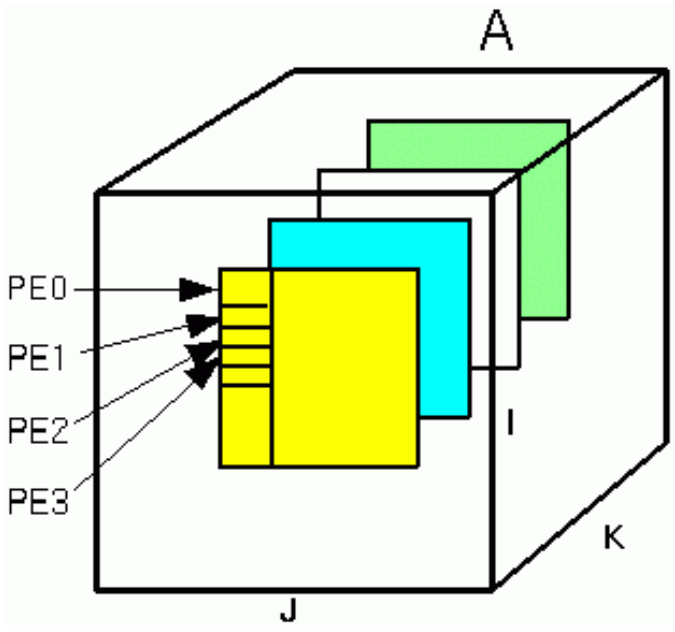
```
do k = ks , ke
   do j = js , je
      do i = is , ie
         A (i,j,k) = A(i,j+jd,k+kd)+ ...
      enddo
   enddo
enddo
```

17.18 Selecting a Loop

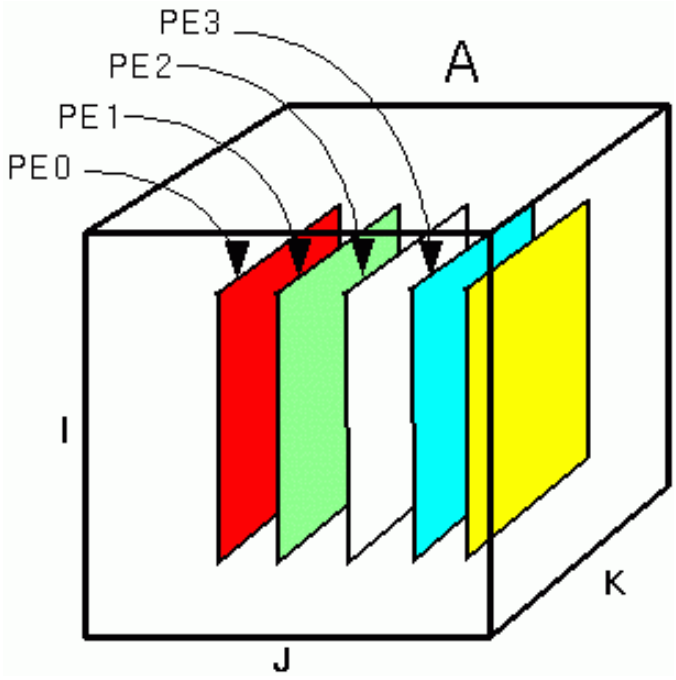


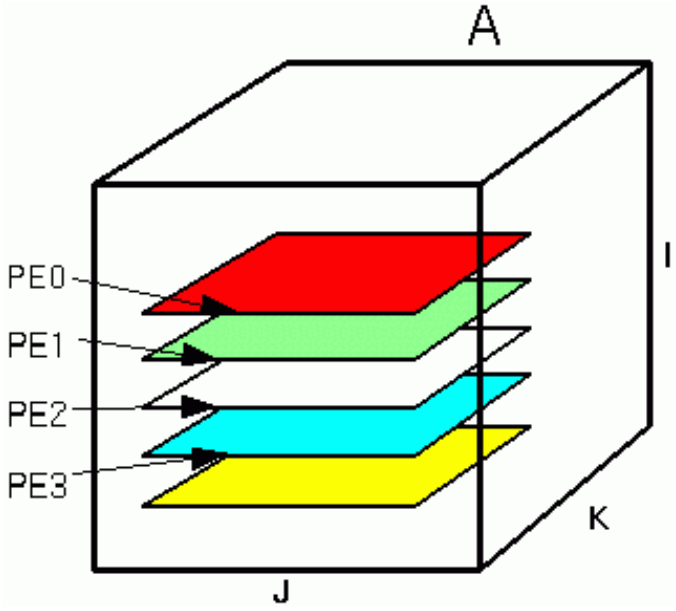
17.19 Choosing a Different Loop





17.20 Reordering Loops





17.21 Parallelization Approaches

- Define your performance goals
- Identify parallel regions
- Scope data
- Distribute data effectively
- Balance the load
- Get the right answers
- Synchronize processes
- Control critical updates
- Minimize overhead
- Hot spots
- Communications
- ReTHINK the problem
- Think BIG
- Think