**CONVEX Computer Corporation**
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

# SPP Language and Programming Overview Course Notes

Order No. TRN-XXXX

Preliminary Edition
January 1995

**CONVEX Education Center**
Richardson, Texas
United States of America

## SPP Language and Programming Overview Course Notes

Order No. TRN-XXXX



Printed in the United States of America

**Revision Information for**

**SPP Language and Programming Overview Course Notes**

| Edition | Document No. | Description |
|---|---|---|

# Contents


## Module 1 Goals

Goals of SPP Programming Model ........ 3

## Module 2 Architecture overview

Exemplar SPP1000 architecture ........ 7
SPP1000 direct mapped cache ........ 10
SPP1200 differences ........ 13
Physical memory ........ 14
Virtual memory ........ 17
Subcomplexes ........ 21
Operating system ........ 24

## Module 3 Application overview

Shared memory application ........ 29
Message passing application ........ 30
Hybrid application ........ 33

## Module 4 Overview of ALL (Assembler, Loader, Libraries)

Features ........ 37
Assembler ........ 38
Loader ........ 39
Libraries included ........ 42

## Module 5 Overview of fc compiler

Features ........ 51
Utilities ........ 54
Libraries included ........ 55

## Module 6 Overview of cc compiler

Features ........ 59
Utilities ........ 63
Libraries included ........ 64





## Module 7 C-series compatibility

SPP ALL issues ........ 67
SPP Fortran issues ........ 68
SPP C issues ........ 78
Common SPP Fortran and C issues ........ 85
Optimization directives ........ 87
C-series directives supported on SPP ........ 88
C-series directives not supported on SPP ........ 89

## Module 8 Scalar optimization

Optimization at -no ........ 93
Optimization at -O0 ........ 98
Optimization at -O0 *(contd.)* ........ 99
Optimization at -O1 ........ 100
Optimization at -O2 ........ 102
Inhibitors of localization ........ 138
Exercises ........ 146

## Module 9 Automatic parallelism

-O3 optimization ........ 149
Inhibitors of parallelization ........ 159
Data dependence in loops ........ 160
Automatically parallelized loops ........ 164
Exercises ........ 170

## Module 10 Assisted automatic parallelism

NO_LOOP_DEPENDENCE ........ 173
LOOP_PRIVATE ........ 174
SAVE_LAST ........ 176
NO_PARALLEL ........ 178
Exercises ........ 179

## Module 11 Synchronization library

Gate and Barrier data types ........ 183
Synchronization library ........ 186
Allocating a gate ........ 187
Freeing a gate ........ 188
Locking a gate ........ 189
Locking a gate conditionally ........ 190
Unlocking a gate ........ 191
Allocating a barrier ........ 192

Freeing a barrier ........ 193
Waiting on a barrier ........ 194

**Module 12 Manual parallelism**

Using manual parallelism ........ 197
PREFER_PARALLEL ........ 202
LOOP_PARALLEL ........ 204
Using *attribute-list* with parallel directives ........ 212
BEGIN_TASKS,NEXT_TASK,END_TASKS ........ 220
TASK_PRIVATE ........ 226
Using *attribute-list* with task directives ........ 229
Nested directives example ........ 244
Exercises ........ 246

**Module 13 Memory classes**

Parallel information functions ........ 249
Default memory allocation ........ 257
Using memory classes ........ 258
Summary of memory classes ........ 259
Fortran memory class directives ........ 261
C storage class extensions ........ 263
Memory class assignments ........ 265
Thread Private class ........ 266
Node Private class ........ 270
Near Shared class ........ 274
Far Shared class ........ 281
Block Shared class ........ 283





# Goals 1

### Topics:

- **Goals of SPP Programming Model**

## Goals of SPP Programming Model

Goals

- Ease of use
  - Familiar shared memory and message passing programming styles
  - Simple porting
  - Tuning easy using familiar CXdb debugger, CXpa profiler, and powerful directives
  - ConvexMLIB provides functionality of VECLIB tuned for SPP
- High performance scalability
- Implementable
  - Implicit data distribution for performance
  - Automatic parallelization techniques applied
  - Two-level nested parallelism - node and thread
  - Programmers have full control over the machine when they need it
- Same features in both Fortran and C





## Architecture overview

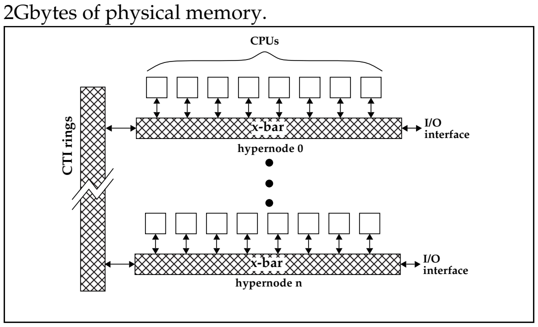
2

**Topics:**

- **Exemplar SPP1000 architecture**
- **SPP1000 direct mapped cache**
- **SPP1200 differences**
- **Physical memory**
- **Virtual memory**
- **Subcomplexes**
- **Operating system**

## Exemplar SPP1000 architecture

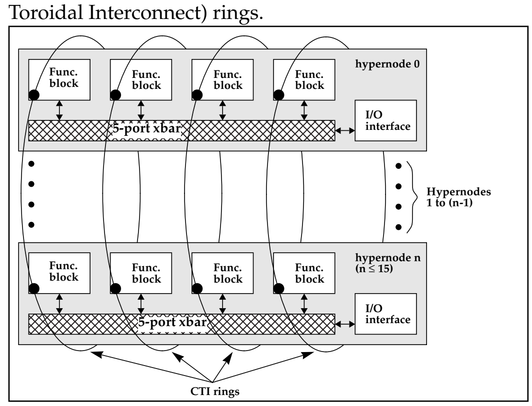
An Exemplar SPP1000 system consists of 1 to 16 hypernodes. Each hypernode contains 4 or 8 HP PA-RISC 7100 processors and 256 Mbytes, 512 Mbytes, 1 Gbyte, or 2Gbytes of physical memory.




Arch overview



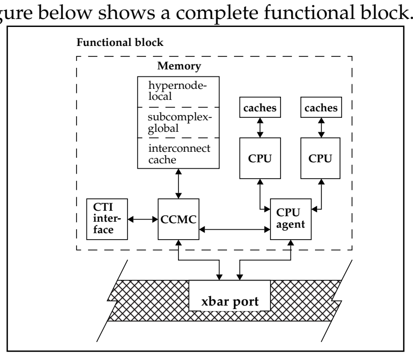
Processors are arranged in functional blocks, each containing two processors, 128 Mbytes to 512 Mbytes of memory, and some control devices. Functional blocks communicate across hypernodes via four CTI (Convex Toroidal Interconnect) rings.




At least one hypernode must have an I/O interface card. The I/O interface card is optional on all other hypernodes.



The figure below shows a complete functional block.




CPUs communicate directly with their own instruction and data caches, which are 1MB in size and are located one clock away from the CPU. The functional block's two CPUs communicate with the rest of the machine through the CPU agent. The Convex Coherent Memory Controller (CCMC) provides the interface between the functional block's two memory banks and the rest of the machine.

The 1 MB processor caches are mapped "direct-to-virtual - memory."

Arch overview



## SPP1000 direct mapped cache

When a program requests something from memory, the data cache is checked to determine whether the data is in the cache. If it is, the data can be retrieved quickly. This is referred to as a *cache hit*.

If the data is not in the cache, a cache line containing the data must be retrieved from main memory and placed into the cache, possibly overwriting what was there. This is referred to as a *cache miss*.

Exemplar uses a direct mapped cache. Under this scheme, a virtual memory location maps into a cache location via the following formula:

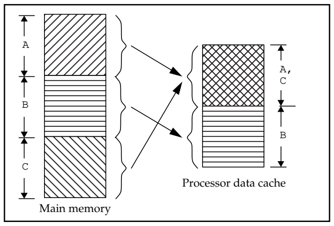
*cache location = virtual memory location MOD cache size*

Exemplar has a 1MB data cache. Using direct mapped cache, arrays whose first element addresses differ by 1MB will map to the same cache location. If these array elements are referenced in the same expression, a considerable amount of overhead will be consumed due to *cache thrashing*.

Consider the following loop:

```
real*8 a(65536), b(65536), c(65536)
common /c1/ a, b, c
do i =1, 65536
     a(i) = b(i) + c(i)
enddo
```

Each array element occupies 8 bytes, so each array occupies .5 MB; therefore arrays `A` and `C` are exactly 1 MB apart in memory, and all their elements have identical cache addresses. The layout of the arrays in virtual memory and in the data cache are shown below:




For I = 1, B(1) and C(1) will be loaded into registers. A cache miss occurs in both cases and two 32 byte cache lines will be loaded into the cache for B(1:4) and C(1:4).

The sum of B(1) and C(1) is calculated and is written to A(1), causing another 32 byte cache line load for A(1:4), overwriting C(1:4) in the cache. The storing of A overwrites C for every iteration of the loop, causing poor performance.



For better performance, change the loop as follows:

```
real*8 a(65536+8), b(65536+8), c(65536+8)
common /c1/ a, b, c
do i =1, 65536
     a(i) = b(i) + c(i)
enddo
```

By adding 64 bytes (an interconnect cache line's space) to the size of each array, corresponding elements in the arrays are shifted into different cache lines and thrashing is eliminated.

Moral:

Avoid placing arrays in memory exactly 1 MB (processor cache size) apart which will thrash the data cache.





## SPP1200 differences

- Uses HP PA-RISC 7200 processors instead of 7100
- Consists of 2 data caches instead of 1:
  - 256 kbyte off-chip direct mapped main cache
  - 2 kbyte on-chip fully associative assist cache
- Cache line prefetching (2 cache lines fetched not 1) which occurs:
  - every time data is fetched from memory
  - every time a prefetched cache line is accessed in the assist cache
- Cache thrashing occurs is a more complicated scenerio since there are 2 caches not 1

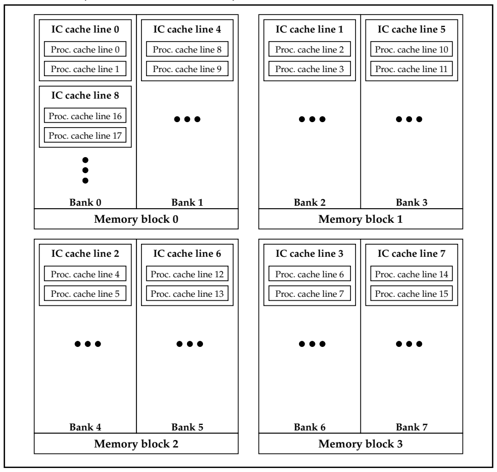


## Physical memory

As mentioned before, a hypernode can have 256 Mbytes, 512 Mbytes, 1 Gbyte, or 2Gbytes of real memory. Physical pages are either

- node local (default), accessible only by the processors on a particular hypernode
- global, accessible only by all processors within a subcomplex
- assigned to interconnect cache, which holds copies of any off-hypernode data referenced by any processor on the hypernode

Physical memory pages are interleaved across the memory banks in each functional block by 64 byte lines. Below the interleaving is shown in the interconnect cache by the 64 byte interconnect cache lines. Contiguous interconnect cache lines are assigned in round-robin fashion, first to the even banks, then to the odd, as shown below.








A processor cache line is 32 bytes wide, whereas interconnect cache lines are 64 bytes wide, containing a pair of processor cache lines. The interconnect cache is

- used to store copies of global data fetched from other hypernodes
- configurable using the Subcomplex Manager
- configurable to be 16, 32, 64, 128, 256, 512, or 1024 Mbytes in size

Compilers and loaders will assure that the following are aligned on a interconnect cache line boundary. For

- C, uninitialized externals
- Fortran, the start of COMMON blocks

For dynamically allocated data, the following is aligned on a interconnect cache line boundary:

- anything **malloc**ed within a parallel executable
- Fortran 90 ALLOCATABLE arrays



## Virtual memory

The virtual address spaces accessible from within a subcomplex are unique; this prevents processes running on one subcomplex from accessing memory on another subcomplex. Furthermore, the virtual address spaces accessible to each process running on a subcomplex are unique; each process is assigned a 4Gbyte virtual space, and processes cannot access each other’s virtual address space.

Virtual memory is divided into classes. They are

- Node local memory
    - thread private
    - hypernode private
- Global memory
    - near shared
    - far shared
    - block shared

The compilers will attempt to choose default classes for your programs that provide good performance; programmers can also manually assign data to memory classes to improve data distribution and further increase performance.





Thread private memory:

- private to each thread of a process
- Data objects have a unique virtual address accessible to a single thread of a process within a hypernode.
- Virtual addresses of thread private data objects map to unique physical addresses within hypernode local physical memory on which a thread is executing.

Node private memory:

- private to all threads on a single hypernode
- Data objects have a unique virtual address by which all threads on a hypernode access it.
- The virtual address of a node private data object maps to a single physical address per hypernode.

Near shared memory:

- Data objects have a single virtual address by which they can be accessed from any thread on any hypernode in the subcomplex.
- The virtual address of a near shared data object maps to a single physical address on a particular hypernode for all threads of a process.

Far shared memory:

- Data objects have a single virtual address by which they can be accessed from any thread on any hypernode in the subcomplex.
- Data is distributed in 4Kbyte pages round-robin to all hypernodes in the subcomplex.
- The virtual address of a far shared data object maps to a single physical address located somewhere on one of the hypernodes for all threads of a process.

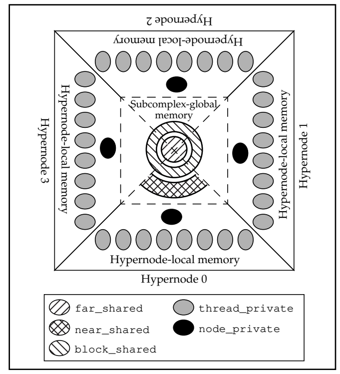
Block shared memory:

- Data objects have a single virtual address by which they can be accessed from any thread on any hypernode in the subcomplex.
- Data is distributed in contiguous blocks equally among the hypernodes, one block per hypernode.
- Must be dynamically allocated by programs at run time.

Arch overview



The figure below shows the virtual addresses associated with a data item stored in each memory class by a single process running on a conceptual 4 hypernode system, with 8 processors per hypernode. Each oval represents a unique virtual address for the same data item within a class; the memory class is indicated by the oval's fill pattern as explained in the illustration.






## Subcomplexes

On Exemplar systems, processes run on virtual machines called subcomplexes, which are arbitrary collections of processors.

Subcomplexes

- are highly configurable. Configuration is done using the CONVEX Subcomplex Manager(SCM).
- allow the system administrator to tailor processors and memory to your specific application needs, making the most efficient use of your Exemplar system resources.
- can be managed without rebooting, but need to be idle
- can consist of as few as one processor or as many as the total number installed on the machine

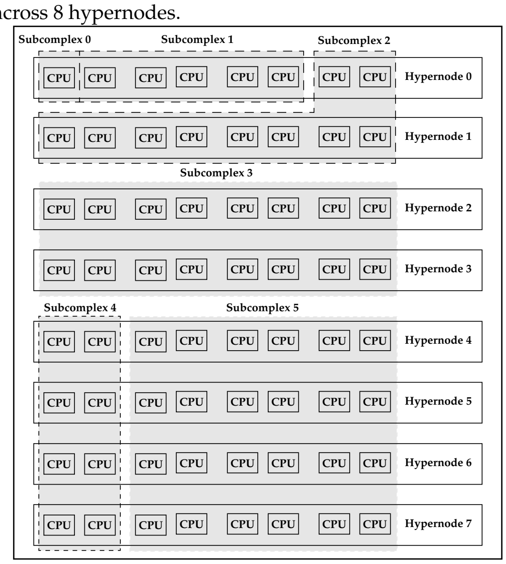
Hypernodes can be split among as many subcomplexes as there are processors in the hypernode, and subcomplexes can be subsets or supersets of hypernodes. Processors can only belong to one subcomplex at a time.

By default, the entire complex belongs to subcomplex 0, the system subcomplex. The machine boots from subcomplex 0. This is also the default subcomplex upon login.

Arch overview



This example of an SPP1000/XA shows 6 subcomplexes across 8 hypernodes.



Subcomplex memory can also be configured using the Subcomplex Manager. Subcomplex global memory can be allocated

- when the subcomplex is idle
- in 16Mbyte increments, up to the total amount available
- from any hypernode which has one or more processors in the subcomplex.

Also configurable are subcomplex permissions, which are similar to file permissions and can be set at the user, group and world level.

- Read permissions on a subcomplex allow you to read the subcomplex configuration.
- Write permissions allow you to change the subcomplex's job scheduling policies, such as process execution priority.
- Execute permissions allow you to run a process on the subcomplex.

Only root can configure or reconfigure a subcomplex.

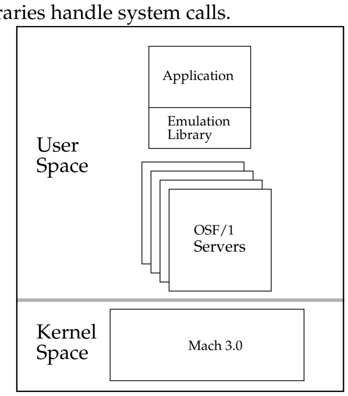
Globally Shared Memory (GSM) is global to a subcomplex, and processes run entirely within their assigned subcomplex. For an application to run on more than one subcomplex, PVM message passing via sockets (not the CTI rings) must be used to communicate between the subcomplexes.





## Operating system

SPP-UX is based on OSF/1 AD (Advanced Development). It consists of a single system image and multiple servers. There is one microkernel running on each hypernode. Emulation libraries handle system calls.



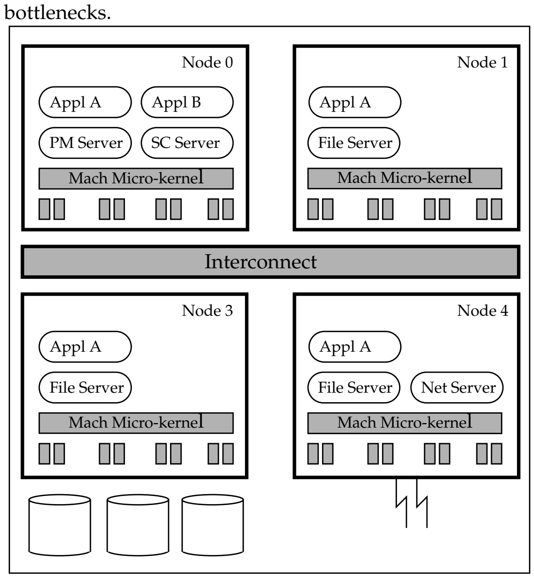

The servers include:

- process management (one per subcomplex)
- file system (as needed)
- networking (as needed)
- subcomplex (one per subcomplex)

Others may be created and used over time.



Servers are distributed across nodes to eliminate resource bottlenecks.




An HP/UX Application Binary Interface (ABI) exists for running single-threaded HP/UX binaries on top of SPP-UX. Applications will have to be recompiled using the SPP-UX compilers to take advantage of parallel processing and shared memory.

# Application overview

**3**

**Topics:**

- **Shared memory application**
- **Message passing application**
- **Hybrid application**





## Shared memory application

- Executes as a single multi-threaded process, one thread per processor within a subcomplex
- Compiler divides up the work in loops among the threads
- Adjusts itself to the number of processors available at runtime (no recompilation needed)
- Automates many of the tasks for which the message passing paradigm requires explicit coding
- Parallelization of the program is handled by the compilers
- Data distribution is taken care of by SPP-UX via the interconnect cache
- Compiler directives and pragmas available to further increase optimization opportunities

Programs that have already been restructured to perform well on conventional shared memory parallel and/or vector machines should run well on SPP.





## Message passing application

- Executes as a collection of independent processes executing in parallel
  - Within a subcomplex, one process per processor
  - One or more subcomplexes as well as external machines can be used
- Communication of data and synchronization of the processes is accomplished via ConvexPVM message passing functions
- Message passing within a hypernode in a subcomplex:
  - Direct process-to-process using shared memory
  - Message packing done in near shared memory
  - Message transferred by message queue pointer manipulation
  - Amount of memory is user configurable via the hostfile
  - Multicasts and broadcasts offer significant performance improvements
  - No PVM daemon involved

## Message passing application *- cont.*

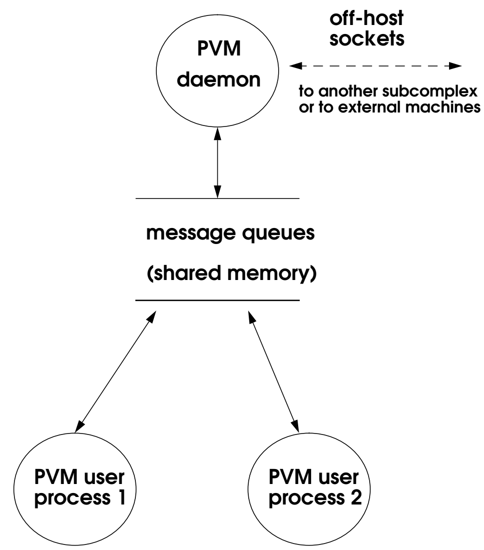
- Message passing across hypernodes in a subcomplex:
  - Direct process-to-process using CTI messaging via CTI rings
- Message passing between subcomplexes and to external machines
  1. Process-to-local PVM daemon using shared memory
  2. Local PVM daemon to remote PVM daemon via sockets
  3. Remote PVM daemon to remote process
- Each PVM process runs as a separate single-threaded process with a separate 4 GB virtual address space
- Since processes cannot access each other’s virtual memory space, cache coherency and its inherent duplication of data is not necessary, so a larger amount of physical memory can be made available to the program as a whole
- Based on public domain PVM, Oak Ridge National Laboratory (version 3.3 is current goal)

Programs that have already been written under the message passing paradigm will be easily ported to SPP.





## PVM within a node in a subcomplex:






## Hybrid application

A hybrid application is a collection of shared memory programs executing and using message passing to coordinate them.

This model allows the majority of the program to be written in the shared memory style while the process-private memory benefits of message passing are exploited.

# Overview of ALL (Assembler, Loader, Libraries) 4

Topics:

- Features
- Assembler
- Loader
- Libraries included





## Features

SPP ALL (Assembler, Loader, Libraries) consists of the assembler (*as*), loader (*ld*), and following runtime libraries:

- Millicode library
- C math libraries
- *libc.a*
- Compiler Parallel Support library (CPS)

New features of ALL include:

- *as* supports the SPP compilers
- *ld* supports the SPP programming model hierarchy
- New millicode routines support the SPP compilers
- *libc.a* has been modified to be thread-safe for parallel execution and also now contains SPP-UX system calls
- Compiler Parallel Support library (CPS)



## Assembler

The assembler, */usr/convex/all/as*, is a modified version of HP-UX */bin/as*. It supports the SPP compilers, and the *fc* and *cc* compiler drivers use it by default.

## Loader

The loader, */usr/convex/all/ld*, is a modified version of HP-UX */bin/ld*. It supports the SPP compilers, and the *fc* and *cc* compiler drivers use it by default.

For the SPP loader, all HP-UX */bin/ld* flags are available as well as the following new SPP flags:

- *+tmhpux* - generate an HP-UX SOM format executable. Invoking *ld* directly creates this type by default.
- *+tmspp1, +tmspp1000, +tmspp1200* - generate SPP-UX ESOM format executable. The *fc* and *cc* compiler drivers create this type by default
- *+minN* - set the minimum number of threads needed for the parallel program to *N*. If *+max* is not used, then the maximum number of cpus will also be set to *N*. The *fc* and *cc* compilers set *+min* to 1 by default
- *+maxN* - set the maximum number of threads needed for the parallel program to *N*. If *+min* is not used, then the minimum number of cpus will also be set to *N*. The *fc* and *cc* compilers set *+max* to 32 by default
- *+parallel* - set the parallel flag in the executable. This is off by default, unless you compile -O3 and the compiler creates parallel code. Don't need to specify if you have already specified *+max*, *+min* or *+over*.
- *+noparallel* - force the executable to be a serial executable.





## Loader - *cont.*

- *+over* - set the oversubscription flag on the executable. By default this flag is not set. It allows you to create more threads than you have cpus for the subcomplex the program is running in.
  - Using *+over* by itself will give you:

    (number of hypernodes * 8) threads
  - Using *+over +maxN* will give you:

    *N* threads
- *+Aalias=symbol* - alias all occurrences of *alias* to *symbol*. This logically renames *alias* to *symbol*, causing all references of the symbol *alias* to resolve to the object pointed to by *symbol*.
- *+Afile* - read an alias list from a file called *file*. The alias list is in the form *alias=symbol*, listed one to a line.
- *+spin* - causes idle threads waiting for parallel work to spin-wait, instead of being suspended after a default amount of time. Done by setting the CPS idle thread wait attribute to *CPS_SPINWAIT* vs. *CPS_SUSPEND* which is the default. See *cps_wait_attr(3)* or *cps_wait_attr(3f)* for more details.
- *+onenode* - mark the executable as a parallel executable that will only run on 1 hypernode. Sets all memory regions to use node private instead of shared.



## Loader - *cont.*

- *+stacktype* - set the memory type for the main thread (thread 0) to *type* which can be one of: far_shared or far, near_shared or near, node_private or node. Default is near shared memory for parallel executables and node private memory for serial executables.
- *+tstacktype* - set the memory type for all of the child threads (threads 1 to n-1) of the main thread (thread 0) to *type* which can be one of: far_shared or far, near_shared or near, node_private or node. Default is near shared memory for parallel executables.





## Libraries included

The millicode library is a collection of low-level routines used by the SPP compilers.

The millicode library, */usr/convex/all/milli.a*, is a modified version of HP-UX */lib/milli.a*. It supports the SPP compilers, and the *fc* and *cc* compiler drivers link it in by default.

There are two C math libraries:

- The POSIX/ANSI C math library, */usr/convex/all/spp1/libM.a*, is a thread safe version of HP-UX */lib/libM.a*. It supports the SPP compilers.
- The K & R C math library, */usr/convex/all/spp1/libm.a*, is a thread safe version of HP-UX */lib/libm.a*. It supports the SPP compilers.

The *fc* and *cc* compiler drivers link in the appropriate library by default, depending upon the mode of the compiler you are compiling in.

## Libraries included - *cont.*

The libc library, */usr/convex/all/libc.a*, is a thread safe version of HP-UX */lib/libc.a*. It also includes all of the SPP-UX system calls. The *fc* and *cc* compiler drivers link it in by default.

The following changes were made to the SPP-UX *libc.a*:

- *printf* and *scanf* family of routines have been modified to support the C language type *long long* which is only available in the Convex SPP C compiler:

  Example:

  **long long x;**

  **scanf ("%lld", &x);**

  **printf ("%lld\n", x);**

  Use the flag *ll* along with the correct %conversion character.

- Some of the */usr/include* header files have been modified to make *libc.a* thread-safe. The new header files are in */usr/convex/all/include* and are picked up first by the *cc* compiler driver before the include files in */usr/include*.

- The system call getrusage(2) has been added.





## Libraries included - *cont.*

- *malloc* has been modified to support both the HP-UX and the SPP-UX allocation algorithms. A new C function, *mallopt*, has been added to select which algorithm to use at runtime. The default for all serial (non-parallel) programs is the HP-UX algorithm. The default for all parallel programs is the SPP-UX algorithm described below:

  - SPP-UX - allocates all objects on 64-byte boundaries. Rounds up the size requested to the next power of 2 for requests < 4 Mbytes. Requests >= 4 Mbytes are rounded up to next page size.

  The *mallopt* parameters to switch between the allocation algorithms are M_USE_HPUX and M_USE_SPPUX. Once memory has been allocated from *malloc*, you can't switch allocation algorithms.



## Libraries included - *cont.*

The Compiler Parallel Support (CPS) library is a set of thread management, low-level synchronization, and high-level synchronization routines used by the compiler for parallel execution.

- */usr/convex/all/spp1/libcps.sl* - CPS shared library automatically linked in by the *fc* and *cc* compiler drivers
- */usr/convex/all/spp1/libcps.a* - archived CPS library
- */usr/convex/all/include/cps.h* - CPS C include file
- A subset of these routines are also made visible to the user for explicit parallel coding purposes:
  - Accessible from C or Fortran
  - "man 3 cps" or EPG for more information
- Thread waiting:
  - Idle threads waiting to do parallel work spin-wait for a default amount of time. If they are not needed after this time, they suspend themselves until they are needed for parallel work.
  - If the program is oversubscribed, the idle threads will suspend themselves without spin-waiting. The spin-wait time may be changed with a call to *cps_wait_attr(3)* or *cps_wait_attr(3f)*.





## Libraries included - *cont.*

- Thread stacksize:
  - The size of the main thread (thread 0) stack is controlled by SPP-UX. The tunables file, */os/tunables*, defines the stacksize parameter:
    - *maxssiz* - maximum size thread 0's stack can grow

    In C and Fortran all uninitialized local variables, unless they are static or **SAVE**d, are allocated on the stack.

    In SPP-UX versions earlier than 3.X, executables need to be run as "mpa -stack" to get more than the default stack size but up to the maximum.

## Libraries included - *cont.*

- All other thread stacks (threads 1 to n-1) are created by CPS startup code which by default creates 8 Mbyte stacks. This size can be changed by setting the environment variable CPS_STACK_SIZE:
  - If CPS_STACK_SIZE exists, CPS startup code will use it as the size to create thread stacks.
  - CPS_STACK_SIZE interpreted in kbytes. For 16 Mbytes thread stacks:

    setenv CPS_STACK_SIZE 16384
  - Following is defined on these thread stacks:

    - LOOP_PRIVATE and TASK_PRIVATE data

    - Uninitialized local variables of a routine that is run in parallel by a thread as the result of a LOOP_PARALLEL directive

    Examples found in the Manual Parallelism section.



## Libraries included - *cont.*

- Thread datasize:
  - The SPP-UX tunables file, */os/tunables*, defines 2 datasize parameters for an executable:
    - *dfldsiz* - default datasize (soft limit)
    - *maxdsiz* - maximum datasize (hard limit)
  - For all threads of an executable, data not allocated on the stack is defined in the data segment. *dfldsiz* is the default data segment size that can be used by an executable. If this is not sufficient, the user may change it by running the executable under **mpa** and specifying a datasize up to *maxdsiz*.

    Example, to run a program with 512 Mbyte data:

    **% mpa -data 512M a.out**

    **% mpa -data 524288K a.out**

    Can also specify 512m and 524288k

    NOTE : Data size is *not* pre-allocated like the stack size. It is not committed unless it is used, so *dfldsiz* can be set to a reasonably large value. It is also inherited by children of process run under mpa.
  - Any dynamically allocated data must also fit in the datasize, i.e., data allocated from the C functions *malloc*, *sbrk*, and *brk*.



# Overview of fc compiler 5

**Topics:**

- **Features**
- **Utilities**
- **Libraries included**

## Features

SPP Fortran upward compatible with C-series Fortran 8.0.

New features include:

- enhanced Fortran 90 support
- HP Fortran compatibility by default
  - variables aligned on data type boundaries
  - allocation of uninitialized local variables and arrays from stack (*maxssiz*)
- new scalar optimizations and compiler flags:
  - global register allocation
  - *-ur* and *-urn n* for unrolling; *-nur* to disable
  - *-uj* and *-ujn n* for unroll and jam; *-nuj* to disable
  - *-sr* for scalar replacement; *-nsr* to disable
- subroutines and functions re-entrant by default
- automatic parallelization of data independent loops and Fortran 90 array syntax
- new synchronization data types and built-in routines
- SPP directive support
- SPP performance options





The following compiler options have been added since version 8.0 of the CONVEX Fortran compiler:

```
-align cseries    -align spp
-ansi77           -ansi90
-blockloop n      -cache n
-cxpa             -cxpab
-cxpar            -cxpalib
-nga              -ngs
-noautopar        -noblock
-nof90            -nore
-noU77            -nsr
-nuj              -nur
-ppu              -sr
-tm spp1          -tm spp1000
-tm spp1200       -uj
-ujn n            -ur
-urn n            -W
-mrl              -nonodepar
```

Refer to the Fortran User's Guide for information on these options.



The following optimization compiler options are redefined:

**Table 1: Optimization options**

| | C-Series | Exemplar |
|---|---|---|
| **-no** | instruction level machine-dependent scalar optimization | same plus generation of multi-op instructions |
| **-O0** | basic block machine-independent scalar optimization | same |
| **-O1** | add program unit machine-independent scalar optimization | same plus global register allocation and base register optimization |
| **-O2** | add vectorization | global instruction scheduling, software pipelining, and data localization optimization |
| **-O3** | add parallelization | same |

The following compiler directives are new to CONVEX Fortran:

```
BARRIER              BLOCK_LOOP
BLOCK_SHARED         FAR_SHARED
FAR_SHARED_POINTER   GATE
LOOP_PARALLEL        LOOP_PRIVATE
NEAR_SHARED          NEAR_SHARED_POINTER
NODE_PRIVATE         NODE_PRIVATE_POINTER
NO_BLOCK_LOOP        NO_LOOP_DEPENDENCE
ORDERED_SECTION      SAVE_LAST
THREAD_PRIVATE       THREAD_PRIVATE_POINTER
CRITICAL_SECTION     SYNC_ROUTINE
NO_UNROLL_AND_JAM    UNROLL_AND_JAM
```





## Utilities

Some useful utilities to assist in porting and development of Fortran programs are:

- fct - converts older Fortran source files into a form acceptable to fc
- fcUnblock - converts a Cray format "blocked" unformatted sequential access file into a suitable fc data file
- fpr - converts Fortran carriage control conventions to UNIX line printer conventions
- fsplit - splits a file containing several Fortran program units into separate files
- fcxref - generates a detailed cross-reference to each object in a Fortran source program

There are numerous system utilities available, e.g. `getpid()`,`getuid()`,etc.; they are documented in section 3F of the Convex man pages.

## Libraries included

The Fortran runtime libraries support:

- Fortran 66, 77, and 90 compatibility
- a subset of Cray and VAX extensions
- intrinsic functions
- thread-safe I/O
- system utilities (syscall) interface
- math functions

The table below shows the name and contents of each runtime library

**Table 2: Fortran runtime libraries**

| Name | Contents |
|---|---|
| libF90.a | Fortran 90 intrinsic function library |
| libU77.a | SPP-UX interface library |
| libU77p8.a | -p8 interface library |
| libU77pd8.a | -pd8 interface library |
| libcfc.a | Cray extension library |
| libcl.a | Standard runtime library routines |

fc





# Overview of cc compiler 6

**Topics:**

- **Features**
- **Utilities**
- **Libraries included**

## Features

SPP C is upward compatible with C-Series C 5.0.

New features include:

- The default mode for the stand-alone preprocessor is extended ANSI (`-ext`).
- cc and lint recognize `TMPDIR` environment variable that specifies an alternate location for temporary files.
- `CCOPTIONS`, `CPPOPTIONS`, and `LINTOPTIONS` accept a vertical bar within the list of flags. Options before the vertical bar are processed before the command-line arguments and options specified after the vertical bar are processed after the command line arguments.
- New predefined symbols have been added that identify the mode they are compiled in (`_ _CONVEX_EXT`, ...) and identify the target structure (`_ _CONVEX_SPP`, ...)
- CONVEX C is compatible with HP-C. Some objects may contain instructions (graphics-related) that are unsupported by the Exemplar architecture.
- New predefined symbols have been added for compatibility with existing HP codes. These are `_ _ hppa`, `_ _hpux`, `_ _hp9000s800`, `_ _HPUX_SOURCE`, and `_PA_RISC1_1`.



- Because `_ _STDC_ _` is most often used to detect ANSI compatible compilers, it is now defined in the extended (`-ext`) mode.
- C++ (“`//`”) style comments are recognized.
- A new pragma, **opt_level**, has been added. It allows individual functions to be compiled at lower optimization levels.
- The limit on array size (256MB max) has been removed.
- allocation of uninitialized local variables and arrays from stack (*maxssiz*)
- new scalar optimizations and flags as in Fortran: global register allocation:
  - *-ur/-urn n/-nur*
  - *-uj/-ujn n/-nuj*
  - *-sr/-nsr*
- functions re-entrant by default
- automatic parallelization of data independent loops
- new synchronization data types and built-in routines
- SPP pragma support
- SPP performance options



The following compiler options have been added since version 5.0 of the CONVEX C compiler:

```
-align cseries    -align spp
-blockloop n      -cache n
-cxpa             -cxpab
-cxpar            -cxpalib
-nga              -ngs
-noautopar        -noblock
-nore             -nsr
-nuj              -nur
-sr               -tm spp1
-tm spp1000       -tm spp1200
-uj               -ujn n
-ur               -urn n
-W                -mrl
-nonodepar
```

Refer to the C User’s Guide for information on these options.



The following optimization compiler options are redefined:

**Table 3: Optimization options**

| | C-Series | Exemplar |
|---|---|---|
| **-no** | instruction level machine-dependent scalar optimization | same plus generation of multi-op instructions |
| **-O0** | basic block machine-independent scalar optimization | same |
| **-O1** | add program unit machine-independent scalar optimization | same plus global register allocation and base register optimization |
| **-O2** | add vectorization | global instruction scheduling, software pipelining, and data localization optimization |
| **-O3** | add parallelization | same |

The following compiler directives are new to CONVEX C:

```
BARRIER              BLOCK_LOOP
BLOCK_SHARED         FAR_SHARED
FAR_SHARED_POINTER   GATE
LOOP_PARALLEL        LOOP_PRIVATE
NEAR_SHARED          NEAR_SHARED_POINTER
NODE_PRIVATE         NODE_PRIVATE_POINTER
NO_BLOCK_LOOP        NO_LOOP_DEPENDENCE
ORDERED_SECTION      SAVE_LAST
THREAD_PRIVATE       THREAD_PRIVATE_POINTER
CRITICAL_SECTION     OPT_LEVEL
SYNC_ROUTINE         NO_UNROLL_AND_JAM
UNROLL_AND_JAM
```

## Utilities

Utilities included with this release are:

- lint - a C source file verifier.
- cpp - an ANSI C compatible macro preprocessor (use `-pcc` for non-ANSI backward compatibility).
- cref - generates a complete cross reference listing of one or more C programs.
- indent - a C program formatter.
- error - analyzes and optionally annotates your source code with comments, pointing out where the compilation error occurred.
- make - executes a pre-defined set of commands to update one or more programs.



## Libraries included

The C libraries support:

- *libbint.a* - bit intrinsics
- lint libraries





# C-series compatibility 7

**Topics:**

- **SPP ALL issues**
- **SPP Fortran issues**
- **SPP C issues**
- **Common SPP Fortran and C issues**
- **Optimization directives**
- **C-series directives supported on SPP**
- **C-series directives not supported on SPP**

## SPP ALL issues



- ALL (Assembler, Loader, and Libraries) provides *as*, *ld*, and libraries based off of HP-UX but with SPP-UX and SPP compiler support added. Additionally, the CPS compiler parallel support library is provided
- Both archived and shared libraries are provided:
  - Shared libraries by default
  - Archived libraries by specifying on the fc or cc command line:

    *fc -Wl,-aarchive foo.f*

    *cc -Wl,-aarchive foo.c*



## SPP Fortran issues

- Data alignment is different
  - All PA-RISC processors require data to be accessed from locations that are aligned on multiples of the data size. The compiler will allocate data items on these aligned boundaries. Yields substantially improved performance.
  - Holes may be left in **COMMON** blocks and can change alignment of fields within **RECORDS** in VAX **STRUCTURE**s (*-align spp*, the default*)*
  - **Bus error** generated when accessing data that has been forcibly misaligned by the user, i.e., array reshaping in **COMMON**, array reshaping of dummy arguments, **EQUIVALENCE**
  - Library routine available to handle misaligned accesses. It installs a signal handler to catch the **SIGBUS** and emulates the load or store operation. The handler completes the unaligned access and the program continues as if nothing happened. Insert the following **CALL** into the main program:

    **CALL ALLOW_UNALIGNED_DATA_ACCESS**

    and then link with *-lhppa.* Several thousand cycle penalty for each access which is misaligned



## SPP Fortran issues - *cont.*



- Tight packing of **COMMON** blocks or **STRUCTURE**s can be forced for applications that need it by compiling with *-align cseries*.
  - Will cause misalignments and bus errors if the misaligned data is accessed
  - All source code must be compiled with this same alignment



## SPP Fortran issues - *cont.*

Following are examples where unaligned data is accessed, causing a runtime Bus error (**SIGBUS**).

Example of misalignment using **COMMON**:

```
PROGRAM MAIN
INTEGER*4 I
REAL*4 X
REAL*8 Y
COMMON /BLKX/ X,Y,I(3)

CALL SUB(I(2))

END


SUBROUTINE SUB(Y)
DOUBLE PRECISION Y

Y = DBLE(1.0)

RETURN
END
```

The **COMMON** block is padded, since **Y,** an 8 byte variable, appears in the middle of 4 byte variables in the block. Subroutine **SUB** then attempts to load an 8 byte value into a 4 byte aligned variable. This causes the bus error. Can use **ALLOW_UNALIGNED_DATA_ACCESS** to fix it.

## SPP Fortran issues - *cont.*



Example of array reshaping of dummy arguments:

```
PROGRAM MAIN
INTEGER*4 I1(10000), I2(10000)

CALL VECCPY(I1(1), I2(1), 1000)
CALL VECCPY(I1(5000), I2(5000), 1000)

END


SUBROUTINE VECCPY(X, Y, N)
INTEGER*4 N
REAL*8 X(N), Y(N)

DO II=1,N
   Y(II) = X(II)
ENDDO

RETURN
END
```

The 1st call to **VECCPY** is ok, since the compiler aligned **I1(1)** and **I2(1)** each on 8 byte boundaries. The 2nd call causes the bus error in **VECCPY**, since **I1(5000)** and **I2(5000)** are each on 4 byte boundaries. Can use **ALLOW_UNALIGNED_DATA_ACCESS** to fix it.



## SPP Fortran issues - *cont.*

Example of using the **EQUIVALENCE** statement:

```
PROGRAM MAIN
INTEGER*4 J(1000)
REAL*8 A(100)
EQUIVALENCE (A(1), J(2))

A(1) = DBLE(1.0)

END
```

Here is an example of unaligned access forced by the user due to the usage of the **EQUIVALENCE** statement. The **EQUIVALENCE** puts **A(1)** on a 4 byte boundary, and then the program attempts to assign an 8 byte value to this 4 byte aligned data item. This causes the bus error. Can use **ALLOW_UNALIGNED_DATA_ACCESS** to fix it.



## SPP Fortran issues - *cont.*



Example of array reshaping in **COMMON**:

```
PROGRAM MAIN
INTEGER*4 N
REAL*4  R1, R2
COMMON /JUNQUE/ N, R1(100), R2(100)

CALL TWO

END

SUBROUTINE TWO
INTEGER*4 N
REAL*8 C
COMMON /JUNQUE/ N, C(100)

C(1) = DBLE(1.0)

RETURN
END
```

This code assumes that **C(50)** is at the same address as **R1(99)**, so the **COMMON** block needs to be tightly packed using *-align cseries*, otherwise the program would generate wrong answers. The program then attempts to load an 8 byte value into **C(1)** which is 4 byte aligned. This causes the bus error. Can use **ALLOW_UNALIGNED_DATA_ACCESS** to fix it. If **C** was a **COMPLEX*8** instead of **REAL*8**, there wouldn't be an alignment problem, since **COMPLEX*8** is implemented as 2-4 byte values.



## SPP Fortran issues - *cont.*

- Moral of the data alignment issue:
  - **DON'T DO THAT!**
  - Do not switch back and forth between data types via any method of data "reshaping"
  - Define all **COMMON** blocks identically
  - **ALLOW_UNALIGNED_DATA_ACCESS** should be used with caution because of stiff performance penalty
  - Best solution might be to recode
- **REAL*16** supported by software as on C-series
- Re-entrant code by default
  - Re-entrant code generated for recursive invocations of subroutines and functions
  - Results in uninitialized local variables and arrays allocated off the runtime stack (unless **SAVE**d) with **unsaved** values (*maxssiz*)
  - *-nore* to disable and force static storage as on C-series (*dfldsiz/maxdsiz*)
- Data objects declared to be in **TASK COMMON** cannot be initialized with **DATA** statements
- Fortran directives for manual optimization

## SPP Fortran issues - *cont.*

C-series compat

- HP Fortran language compatibility in default mode
  - Stack based storage of uninitialized local variables and arrays (*maxssiz*)
  - Data alignment on data type boundaries
  - Use of intrinsics in constant expressions which are evaluated at compile time

```
PROGRAM MAIN
PARAMETER (MNUM = MAX(81,100))
PARAMETER (MYINT = NINT(558.7))
INTEGER*4 M, N
M = MNUM + 1
N = MYINT + M
END
```

  - External naming conventions for Fortran program blocks:

**Table 4: External Fortran naming conventions**

| Fortran program block | C Series external name | SPP and HP external name |
|---|---|---|
| Main program | _MAIN__ | main__ |
| Blank **COMMON** | ___blnk_ | _BLNK |
| Named **COMMON** | *__name_* | *name* |
| Subprogram | *_name_* | *name* |



## SPP Fortran issues - *cont.*

  - *-ppu* command line option to cause the compiler to append an underscore to external names such as subroutines, functions, and COMMON blocks
  - *-noU77* command line option to suppress trailing underscores normally added to libU77 names. Doesn't affect the way in which user-written subprograms are named
  - *-W* command line option as in C-series to pass arguments to a specified subprocess such as the preprocessor, assembler, or loader:

    Format: *-Wsubproc,arg1 [,arg2... ,argn]*

    Example: *fc -Wl,+min1,+max4,-aarchive foo.f*

    In this example, min/max options are passed to ld as well as the archived library request
  - *stdin*, *stdout*, *stderr* unit numbers 5, 6, 7 instead of 5, 6, 0 as on C-series
- *-ur* unroll optimization provided by default at -O2/-O3
  - *-nur* to disable
  - *-urn n* to provide an unroll factor
  - C-series need to explicitly specify *-ur*



## SPP Fortran issues - *cont.*

C-series compat

- Fortran 90 language features provided in default mode as on C-series
  - *-nof90* to disable
  - Array sections
  - Vector-valued subscripts
  - Array-valued expressions
  - Array constructors
  - Masked array assignments (**WHERE** statement)
  - Automatic arrays
  - Allocatable arrays
  - Certain Fortran 90 array manipulation intrinsics



## SPP C issues

- Data alignment is different
  - All PA-RISC processors require data to be accessed from locations that are aligned on multiples of the data size. The compiler will allocate data items on these aligned boundaries. Yields substantially improved performance.
  - Can change alignment of members within structures (*-align spp*, the default), since on C-series **long long** and **double** are 4 byte aligned
  - Affects the layout of binary data files if structures as a whole are written to them
  - **Bus error** generated when accessing data that has been forcibly misaligned by the user, i.e., misaligned pointers, redefined data types
  - Library routine available to handle misaligned accesses. It installs a signal handler to catch the **SIGBUS** and emulates the load or store operation. The handler completes the unaligned access and the program continues as if nothing happened. Insert the following call into the main program:

    **allow_unaligned_data_access()**

    and then link with *-lhppa.* Several thousand cycle penalty for each access which is misaligned.

## SPP C issues

C-series compat

- Tight packing of structures can be forced for applications that need it by compiling with *-align cseries*.
  - Will cause misalignments and bus errors if the misaligned data is accessed
  - All source code must be compiled with this same alignment



## SPP C issues - *cont.*

This is the C version of the Fortran example of array reshaping of dummy arguments. In C, the user misaligned pointer causes the problem:

```
main()
{
    int i1[10000], i2[10000];

    veccpy(&i1[0], &i2[0], 1000);
    veccpy(&i1[4999], &i2[4999], 1000);

}

veccpy(double *x, double *y, int n)
{
    int ii;

    for (ii=0; ii<n; ii++) {
       y[ii] = x[ii];
    }

}
```

The 1st call to **veccpy** is ok, since the compiler aligned **i1(0)** and **i2(0)** each on 8 byte boundaries. The 2nd call causes the bus error in **veccpy**, since **i1(4999)** and **i2(4999)** are each on 4 byte boundaries. Use **allow_unaligned_data_access** to fix it.



## SPP C issues - *cont.*

C-series compat

Example of a user misaligned pointer:

```
union union1{
      int z;
      double y;
};

main()
{

   struct struct1{
         int x2;
         union union1 u1;
         int i[3];
   } s1;

   sub(&(s1.i[1]));

}

sub(double *y)
{
   *y = (double)1.0;
}
```

The structure is padded, since 8 bytes must be allocated for union **u1** and it appears in the middle of 4 byte variables. **i[1]** is on a 4 byte boundary and its address is passed into function **sub**. The program then attempts to load an 8 byte value into this 4 byte aligned variable. This causes the bus error. Can use **allow_unaligned_data_access** to fix it.



## SPP C issues - *cont.*

  - Moral of the data alignment issue for C is the same as for Fortran, except in C you need to watch out for pointer and structure usage
- Re-entrant code by default
  - Re-entrant code generated for recursive invocations of subroutines and functions
  - *-nore* to disable
- Uninitialized locals always allocated on the stack as in C-series (*maxssiz*)
- **longjmp** must occur within the same thread that called **setjmp**
- Removed 256 Mbyte array size limitation as in C-series
- C++ style comments accepted as in C-series
- Compilation modes: *-ext* (the default), *-std*, *-pcc*. No *-str* mode as on C-series since there is not a library containing only the ANSI C functions like on C-series
- **__STDC__** now defined in extended (*-ext*) mode as in C-series, since most often used to detect **ANSI** compatible compilers
- C optimization pragmas that are equivalent to the supported Fortran directives

## SPP C issues - *cont.*

C-series compat

- **#pragma _cnx opt_level (** [-no | -O0 | -O1 | -O2 | -O3] **)**
  - Sets the effective optimization level to the minimum of the optimization level specified on the command line and the level specified within the pragma
  - Pragma should appear at file scope
  - Remains in effect until the next **opt_level** pragma or the end of the file
  - On C-series as well
- New predefined symbols:
  - Identifies the mode of the compiler used
  - Identifies the target machine
- New predefined symbols added for compatibility with existing HP codes



## SPP C issues - *cont.*

- HP C language compatibility in default mode
  - A “plain” **int** bit field is treated as a **signed int** bit field, whereas on the C-series it is treated as an **unsigned int** bit field.
    - Means you can only use 31 bits instead of 32
    - Declare as **unsigned int** to get 32 bits
  - A right shift is an **arithmetic** shift and a left shift is a **logical** shift.
    - On the C-series, both shifts are **logical**
    - Declare as **unsigned int** for right shift to be **logical**
  - Data alignment on data type boundaries
  - External naming conventions for C program blocks:

**Table 4: External C naming conventions**

| C program block | C Series external name | SPP and HP external name |
|---|---|---|
| Main program | _main | main |
| Subprogram | *_name* | *name* |

  - *-W*, *-ur*, *-nur*, and *-urn n* command line options as in Fortran



## Common SPP Fortran and C issues

C-series compat

- Mixed C and Fortran programs supported
- *-O2* was vectorization and is now global instruction scheduling, software pipelining, and data localization optimizations.
- Default mode of compiler was *-no* and is now *-O2*
- C-series Fortran and C flags not supported on SPP
  - *-fn*
  - *-fi*
  - *-fx*
  - *-metrics*
  - *-ds*
  - *-ep {n}*
  - *-g*
  - *-db*
  - *-except {precise | default}*
  - *-mi {n}*
  - *-pb*
  - *-p, -pg (supported on SPP for load only)*



## Common SPP Fortran and C issues - *cont.*

  - *-tl {n}*
  - *-nopm*
  - *-tm c* (no C-series architecture types supported)*
- C-series C only flags not supported on SPP
  - *-str*
  - *-compat rrf*=option
- C-series Fortran only flags not supported on SPP
  - *-sa*
  - *-xro* and *-xr1* as well as the old cross-reference generator utility *fxref*

## Optimization directives

- Default format for SPP directives is same as C-series
  - Fortran:

    **C$DIR** directive
  - C:

    **#pragma _CNX** directive
- Certain directives can apply to **both** C-series and SPP, perhaps with different parameters for each
- Optional formats of the directives which specify their target machine are also available
  - Fortran:

    **C$DIR [ SPP | CSERIES ]** directive
  - C:

    **#pragma _CNX [ SPP | CSERIES ]** directive

    If the **SPP** or **CSERIES** option is specified, the directive applies only when the target machine option is also the target machine of the compiler.

    The options may also be specified in lowercase: **spp** or **cseries** in both Fortran and C.





## C-series directives supported on SPP

The following C-series optimization directives will be supported in the shared memory programming style on SPP:

- **BEGIN_TASKS, NEXT_TASK, END_TASKS**
- **NO_PARALLEL**
- **NO_PEEL**
- **NO_PROMOTE_TEST**
- **NO_SIDE_EFFECTS**
- **PEEL**
- **PEEL_ALL**
- **PREFER_PARALLEL**
- **PROMOTE_TEST**
- **PROMOTE_TEST_ALL**
- **RETURNS_UNIQUE_POINTER**
- **ROW_WISE**
- **SCALAR**
- **TASK_PRIVATE**
- **UNROLL**



## C-series directives not supported on SPP



- **DO_PRIVATE**
- **FORCE_PARALLEL**
- **FORCE_PARALLEL_EXT**
- **FORCE_VECTOR***
- **MAX_TRIPS**
- **NO_RECURRENCE**
- **NO_VECTOR***
- **PREFER_PARALLEL_EXT**
- **PREFER_VECTOR***
- **PSTRIP**
- **SELECT***
- **SYNCH_PARALLEL**
- **VSTRIP***

Rationale:

* vector-specific directives

Semantics of **FORCE_PARALLEL** are sometimes in conflict with new directives on SPP. Loops can be forced to execute in parallel with **LOOP_PARALLEL.**



## C-series directives not supported on SPP - *cont.*

Equivalent of **DO_PRIVATE** will be **LOOP_PRIVATE**.

**LOOP_PARALLEL(ORDERED)** replaces **SYNCH_PARALLEL** but leaves synchronization to the user.

**NO_RECURRENCE** makes sense only in a vector processor, where a cycle of loop-carried and loop-independent dependences prevents vectorization. It is also too powerful, since it causes the compiler to ignore all dependences.

**NO_LOOP_DEPENDENCE (array-namelist)** deals with loop carried dependences which prevent parallelization and it allows the user to target specific potential dependencies, rather than use a shotgun approach.

# Scalar optimization

# 8

**Topics:**

- **Optimization at *-no***
- **Optimization at *-O0***
- **Optimization at *-O1***
- **Optimization at *-O2***





## Optimization at -no



Machine instruction level scalar optimization.

Compilers create object code that fully uses the scalar features of the PA-RISC architecture.

- **Instruction scheduling**.
  - Compilers work on single source statements only.
  - Rearranges machine instructions to maximize use of the functional units within a CPU. Each CPU has multiple functional units on which operations execute simultaneously.
  - Concurrent execution of machine instructions on multiple functional units within a processor *(pipelining)*.
  - **Delay slot filling**-feature of *branch instructions*: the transfer of control from a *branch instruction* occurs one instruction after the execution of the branch; as a result, the instruction following the branch, located in the **delay slot** of the *branch instruction*, is executed before control passes to the branch destination.



## Optimization at -no *(contd.)*

- **Instruction scheduling**. ***(contd.)***
  - **Example of delay slot filling:**

    **Program segment:**

```
100   STW  r3,0(r6);non-branch instr.
104   BLR  r8,r0   ;branch to loc.200
108   ADD  r7,r2,r3;instruction in
                   ;delay slot
10C   OR   r6,r5,r9;next instruction
                   ;in linear code
                   ;sequence
 ...
200   LDW   0(r3),r4;target of branch
                    ;instruction
```

**Execution sequence:**

```
100   STW  r3,0(r6) ;
104   BLR  r8,r0    ;
108   ADD  r7,r2,r3 ;delay slot
                    ;instruction is
                    ;executed before
200   LDW   0(r3),r4;execution of
                    ;target instruc-
                    ;tion
```

## Optimization at -no *(contd.)*

- **Instruction scheduling**. *(contd.)*
  - Register interlock avoidance: a register interlock occurs if an instruction attempts to use a register which is the target of a previous load instruction that has not yet completed. The delay of the load instruction may be much longer than a single execution cycle for a cache miss, a TLB miss, or a page fault. The compilers do instruction scheduling to avoid dependence on register interlocks.
- **Span-dependent instructions.**
  - The compilers automatically generate branches using the most efficient and appropriate branching techniques (it generates short branches, when possible).
- **Register allocation.**
  - Compilers use a technique for allocating registers that fully exploits the PA-RISC registers set.
  - Grouping of register loads and concurrent execution of instructions (*pipelining*), to reduce register conflicts.
- **Generation of multi-op instructions**

Scalar opt



## Optimization at -no *(contd.)*

- **Tree-height reduction (balancing).**

The compiler represents expressions internally as trees. When they involve floating point numbers, these trees are optimized by *tree-height reduction*.

**Example of expression involving REAL variables:**

```
A + B + C + D + E + F + G + H
```

**-Unbalanced tree representation:**

```
(((((((A + B) + C) + D) + E) + F) + G )+ H)
```

PA-RISC add functional unit: **2 clocks to complete an add**, since each addition depends on the result of the addition to the left.

**-Balanced tree representation:**

```
(((A + B) + (C + D)) + ((E + F) + (G + H)))
```

With full pipeline - **1 add result/clock** because none of the four additions require the result of another addition

If no evaluation order is specified using parenthesis, the compilers choose one that minimizes the depth of the expression and maximizes instruction pipelining.



## Optimization at -no *(contd.)*

- **Short-circuit evaluation of conditionals in Fortran.**

Compilers skip irrelevant tests from **IF** statements when logical operators are involved in the conditional and they are used in a logical context.

```
IF ( expr1.OR.expr2 ) then . . .
```

If ***expr1*** is true, the evaluation of ***expr2*** is skipped.

```
IF ( expr1.AND.expr2) then. . .
```

If ***expr1*** is false, the evaluation of ***expr2*** is skipped.

- C - ANSI - Standard behavior
- CONVEX FORTRAN - short-circuiting - default
  -'*-nosc*' flag - disables it

Scalar opt



## Optimization at -O0

Basic block scalar optimization.

- **Instruction scheduling**.
  - Multiple operation scheduling
    Instructions from multiple statements scheduled as a group
- **Local Optimizations.**
  - Redundant-assignment elimination
    Removal of assignment when variable is not used between two assignments
  - Assignment substitution
    Removal of redundant loads by loading variable into a register and using register in subsequent references to the variable
  - Common-subexpression elimination
    Calculation of common subexpressions once and using result when subexpression encountered again instead of recalculating
  - Redundant-use elimination
    Common subexpression elimination where the subexpression is a variable

## Optimization at -O0 *(contd.)*

- Constant propagation and folding

Replacement of variable references with constant until new value is assigned

- Algebraic and trigonometric simplification

Simplification of algebraic and trigonometric expressions

Scalar opt



## Optimization at -O1

Global optimization across a group of basic blocks within a procedure.

- **Local optimization.**

Strength reduction

Replacement of arithmetic operations with equivalent operation that executes more quickly

Constant propagation and folding

Redundant-assignment elimination

Elimination of assignments to variables that do not have subsequent references within the program unit

Dead-code elimination

Elimination of unreachable code caused by constant propagation and folding

Copy propagation

Replacement of a variable with another variable to which it has been equated

Common subexpression elimination

Code motion

Movement of invariant expressions out of loops



## Optimization at -O1 *(contd.)*

- **Global register allocation (GRA)**

Store commonly-referenced scalar variables in registers and carry across basic block boundaries

Minimize memory accesses within a procedure

Consider:

```
DO I = 1, N
  A(I) = X
  .
  .
ENDDO
```

**X** is referenced on every iteration of the loop. Eliminating loads and stores of **X** to main memory can substantially improve performance.

Using GRA, the compiler generates code equivalent to:

```
REG = X
DO I = 1, N
    A(I) = REG
    .
    .
ENDDO
X = REG
```

GRA is disabled for shared variables assigned within critical regions delimited by compiler directives.

Scalar opt



## Optimization at -O2

*Data locality* - keeps heavily used data in the processor data cache, eliminating the need for more costly interconnect cache or main memory accesses. Default mode.

Loops that manipulate arrays are eligible for the transformations that the compiler performs at -O2 to achieve localization.

```
REAL*4 A(100,100), B(100,100), C(100)
COMMON /BLK/ A,B,C
...
DO J = 1,100
  DO I = 1,100
    A(I,J) = B(I,J)*C(I)
  ENDDO
ENDDO
```

**A,B,C** - 3 **REAL*4** arrays = 78.5 kbytes. All elements of these arrays will fit easily into the processor data cache (1 Mbyte), so elements being overwritten will not be a problem.

In FORTRAN, arrays are stored in column major order. In this particular case, storing the arrays in **COMMON** block eliminates the possibility of cache thrashing because this ensures contiguous storage.

## Optimization at -O2 *(contd.)*

On the first iteration of the loop (**I=1**):

**A(1,1), B(1,1), C(1)** will be fetched from memory as part of separate 32-byte cache lines.

On the second fetch, which will not happen until any present reusable elements are used, the cache line will begin with the first element being fetched and contain the 7 following elements:

For **J=1** — **A(I,1)** will encache **A(I:I+7,1)**
**B(I,1)** will encache **B(I:I+7,1)**
**C(I)** will encache **C(I:7)**

For the next 7 iterations of the **I** loop, all data is *encached* and can be used, boosting the performance because there are no more fetches from memory.

For **J=2,100** *data reuse* gives another performance boost:

**C** already *encached* - available immediately from the processor data cache, so 13 out-of-cache memory accesses are saved.





## Optimization at -O2 *(contd.)*

- **Strip mining**

Splitting a loop into a nested loop.

The new inner loop iterates over a *strip* (section of the original loop). New outer loop runs the inner loop enough times to cover all the strips to achieve the total number of iterations.

By itself, not profitable, but **essential** for the loop **blocking** optimization.

*-strip length* - number of iterations of the inner loop

**Example:**

```
DO I = 1,10000
  A(I) = A(I)*B(I)
ENDDO
```

With strip mining:

```
DO IOUTER = 1,10000,1000
  DO ISTRIP = IOUTER, IOUTER+999
    A(ISTRIP) = A(ISTRIP)*B(ISTRIP)
  ENDDO
ENDDO
```



## Optimization at -O2 *(contd.)*

- **Loop distribution**

Compiler transforms loops to try to make all calculations in a nested loop be performed in the innermost loop.

Transforms complicated nested loops into several simple loops.

Creates more opportunities for **loop interchange**.

**Example**:

Before distribution:

```
DO I = 1,N
  B(I,1) = 0.
  DO J = 1,M
    A(I) = A(I) + B(I,J)*C(I,J)
  ENDDO
  D(I) = E(I) + A(I)
ENDDO
```





## Optimization at -O2 (contd.)

**Loop distribution** *(contd.)*

After distribution, the previous loop is transformed into three loops:

```
DO I = 1,N
  B(I,1) = 0.
ENDDO
DO I = 1,N
  DO J = 1,M
    A(I) = A(I) + B(I,J)*C(I,J)
  ENDDO
ENDDO
DO I = 1,N
  D(I) = E(I) + A(I)
ENDDO
```

All three assignments are moved to innermost loops.

## Optimization at -O2 *(contd.)*

- **Loop interchange**

  To facilitate other transformations.

  To relocate the loop that is most profitable to parallelize so that it is outermost (at optimization level -O3 only).

  To optimize inner-loop memory accesses by making them consecutive.

  **Example:**

```
DO I = 1,N
  DO J = 1, M
    A(I,J) = B(I,J) + C(I,J)
  ENDDO
ENDDO
```

  Interchanged to facilitate column access:

```
DO J = 1,M
  DO I = 1,N
    A(I,J) = B(I,J) + C(I,J)
  ENDDO
ENDDO
```

Scalar opt



## Optimization at -O2 *(contd.)*

- **Loop blocking**

  A form of strip mining for cache optimization and is driven by data reuse.

  Necessary for nested loops which manipulate arrays too large to fit into the cache, and that are accessed by nonconsecutive memory locations.

  Maximizes **data localization** through a combination of **strip mining** and **loop interchange** of newly created loops**.**

  A blocked loop accesses array elements in sections sized to optimally fit in the cache, allowing for *spatial* and *temporal* reuse of data.

  Results -> minimizes cache misses.

  **Data reuse**

  ***Spatial* reuse** - using data that was encached as a result of fetching another piece of data from memory (32 byte cache line).

  ***Temporal* reuse** - using the same data item on more than one iteration of the loop.



## Optimization at -O2 *(contd.)*

**Reuse example**:

```
REAL*4 A(100,100), B(100,100), C(100)
COMMON /BLK/ A, B, C
...
DO J = 1,100
  DO I = 1,100
    A(I,J) = B(J,I)*C(I)
  ENDDO
ENDDO
...
```

**A** - *spatial* reuse with respect to the **I** loop:

every 8th iteration of the **I** loop fetches a cache line containing 8 of its elements; the 7 iterations between main memory accesses proceed with no load delays.

Scalar opt



## Optimization at -O2 *(contd.)*

**Reuse example** *(contd.)*:

**B** - *spatial* reuse:

during the first iteration of **J**, every referenced element of **B**, along with its containing cache line, must be fetched from memory, so some reuse may be possible. On subsequent iterations of **J**, whenever a cache line is fetched from memory, all the elements it contains will be usable. Fetches are function of **I** and may occur for different **J** values. Since **B**'s row index is **J**, any unused encached elements are used on the subsequent iterations of **J** for a given value of **I**. Though the fetches do not synch up as nicely as they do for **A**, *spatial reuse* is still fully exploited.

**C** - *spatial* reuse with respect to the **I** loop, for **J = 1** only.

- *temporal* reuse with respect to the **J** loop (for every subsequent iteration of **J**; after first iteration of **J** finishes, **C** is completely contained in the processor's data cache and will remain there for the duration of **J**).

## Optimization at -O2 *(contd.)*

**Blocking example:**

```
REAL*8 A(1000,1000),B(1000,1000),C(1000)
COMMON /BLK/ A,B,C
...
DO J = 1,1000
  DO I = 1,1000
    A(I,J) = B(J,I)+C(I)
  ENDDO
ENDDO
```

Array elements occupy nearly 16 Mbytes of memory, and blocking becomes profitable.

First, the compiler strip mines the inner loop: the iteration space is selected such that all array elements it references will fit into the cache without overwriting themselves.

```
DO J = 1,1000
  DO IOUT = 1,1000, IBLOCK
    DO I = IOUT, IOUT+IBLOCK-1
      A(I,J) = B(J,I)+C(I)
    ENDDO
  ENDDO
ENDDO
```





## Optimization at -O2 *(contd.)*

**Blocking example** *(contd.)*:

Loop interchange of the new loop nest, such that the innermost loop is 'blocked' from overwriting its strips.

```
DO IOUT = 1,1000, IBLOCK
  DO J = 1,1000
    DO I = IOUT, IOUT+IBLOCK-1
      A(I,J) = B(J,I)+C(I)
    ENDDO
  ENDDO
ENDDO
```

**I** loop is strip mined.

**IBLOCK** - *block factor* (also the *strip mine length*) chosen based on the size of the arrays and the size of the cache.

**J** and **IOUT** loops interchanged to block the **I** loop.

New nest accesses **IBLOCK** rows of **A** and **IBLOCK** columns of **B** for every iteration of **J**.

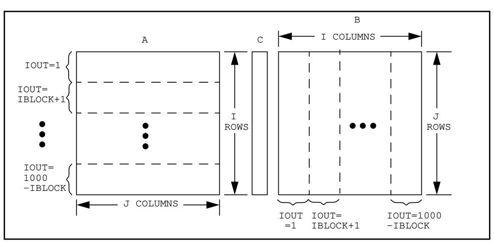
At every iteration of **IOUT**, the nest accesses **1000 IBLOCK** -length columns of A (or an **IBLOCK*1000** chunk of **A**), and **1000 IBLOCK**-width rows of **B**.

This is illustrated in the next picture:



## Optimization at -O2 *(contd.)*

**Blocking example (contd.):**




*Spatial reuse* on **A**: fetches of **A** occurring after the first fetch encaches the needed element and the next 3 elements that are used in the 3 subsequent operations.

*Spatial reuse* on **B**: **I** loop traverses columns of **B,** so fetches of **B** encache extra elements that will not be spacially reused until **J** increments.

**IBLOCK** is chosen to maximize *spatial reuse* of **B**.



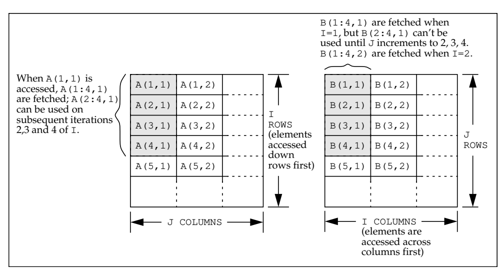


## Optimization at *-O2* *(contd.)*

**Blocking example (contd.):**

The next picture illustrates how cache lines of each array are fetched, assuming that **A** and **B** both start on cache line boundaries.

The shaded area represents the initial cache line fetched.




*Temporal reuse* of **C** - **IBLOCK** elements of **C** will remain in the cache for several iterations of **J** before being overwritten.

By the time any of the arrays are overwritten, all *spatial reuse* has been exhausted.

## Optimization at -O2 *(contd.)*

- **Blocking directives/pragmas**

  **BLOCK_LOOP, NO_BLOCK_LOOP**

  **Fortran**

```
C$DIR BLOCK_LOOP(BLOCK_FACTOR = n)
C$DIR NO_BLOCK_LOOP
```

  **C**

```
#pragma _CNX block_loop
                 (block_factor = n)
#pragma _CNX no_block_loop
```

Use the **BLOCK_LOOP** directive when the compiler can't tell the size of the arrays in the loop, and therefore has to rely on heuristics to guess. An example would be using adjustable or assumed-size arrays in the loop.





## Optimization at -O2 *(contd.)*

**BLOCK_LOOP** tells the compiler to use block factor ***n*** to strip mine the immediately following loop.

- ***n*** must be an integer constant or integer constant expression

- For best performance choose ***n*** such that:

  (***n*** * data type size) = multiple of the cache line size

  AND

  The cache will be full of elements of the array that is accessed nonconsecutively. If more than 1 array is accessed nonconsecutively, calculate ***n*** to fill the cache with the array with the largest leading dimension for Fortran or second dimension for C.

- Example on following page

**NO_BLOCK_LOOP** tells the compiler to not stripmine the loop

- **Blocking compiler options**

  *-noblock* - disables blocking

  *-blockloop n* - specifies blocking factor ***n*** for all loops



## Optimization at -O2 *(contd.)*

**BLOCK_LOOP, NO_BLOCK_LOOP** *(contd.)*

- **Example:**

```
SUBROUTINE MYSUB (A, B, N)
REAL*8 A(N,N), B(N,N)

DO J = 1,N
  DO I = 1,N
    A(I,J) = B(J,I)
  ENDDO
ENDDO

RETURN
END
```

If the compiler were to interchange the **I** and **J** loops, consecutive memory accesses would be lost on either the loads or stores. To compromise and minimize cache misses, the compiler blocks the loop, but must guess on the blocking factor, since it doesn't know the size of the arrays.

If the user knows the size of the arrays, the blocking factor can be calculated and used in the **BLOCK_LOOP** directive.





## Optimization at -O2 *(contd.)*

Suppose the user knows **N** = 1024, and the total size of the arrays is greater than the 1 Mbyte cache.

In Fortran arrays are stored in column-major order, so array **B** is being accessed non-consecutively and array **A** consecutively. When a cache line is fetched for **A**, the elements are used immediately in the next iterations of the loop. However, when a cache line is fetched for **B**, only 1 element is used immediately. It is desirable to leave those elements of **B** in the cache to be used later.

The question is, at what point does **B** fill the 1 Mbyte cache? This determines our blocking factor:

1048576 bytes cache / (1024 leading dimension of **B** * 8 bytes data type size of **B**) = 128 blocking factor

Further, 128 is a good choice because:

128 * 8 bytes = 1024 bytes = 32*32 bytes (cache line size)

As a result,.......................................................................

## Optimization at -O2 *(contd.)*

The code with the directive is:

```
        SUBROUTINE MYSUB (A, B, N)
        REAL*8 A(N,N), B(N,N)

        DO J = 1,N
C$DIR     BLOCK_LOOP(BLOCK_FACTOR=128)
          DO I = 1,N
            A(I,J) = B(J,I)
          ENDDO
        ENDDO

        RETURN
        END
```

which the compiler uses to transform the loop into:

```
        DO IOUT = 1,N,128
          DO J = 1,N
            DO I = IOUT,IOUT+127
              A(I,J) = B(J,I)
            ENDDO
          ENDDO
        ENDDO
```

Summary: Loops to be blocked by hand, must be analyzed to determine the optimal blocking factor.





## Optimization at -O2 *(contd.)*

- **Loop unrolling**

  Involves replicating the body of a loop and increasing its loop step value. Sometimes it eliminates the loop structure completely.

  Allows for less loop overhead, and opportunities for pipelined execution of operations, parallel/multiple execution of operations such as FMPYADD, improved register use, and more efficient scheduling.

  Unrolling can be total or partial. Total unrolling involves eliminating the loop structure completely. This only makes sense for loops with small trip counts.

  **Example:**

```
DO I = 1,4
  A(I) = A(I) + B(I) * C
ENDDO
```

  This loop is completely unrolled as shown:

```
A(1) = A(1) + B(1) * C
A(2) = A(2) + B(2) * C
A(3) = A(3) + B(3) * C
A(4) = A(4) + B(4) * C
```

  Here, loop overhead is completely eliminated.



## Optimization at -O2 *(contd.)*

- **Loop unrolling** *(contd.)*

  Partial unrolling is performed on loops with larger or unknown trip counts. The loop structure is retained, but the body is replicated a number of times equal to the *unrolling depth* (also known as the *unroll factor*). References to the iteration variable are adjusted accordingly.

  **Example:**

```
DO I = 1,100
  A(I) = A(I) + B(I) * C
ENDDO
```

  This loop is can be unrolled to a depth of 4 as shown:

```
DO I = 1,100,4
  A(I) = A(I) + B(I) * C
  A(I+1) = A(I+1) + B(I+1) * C
  A(I+2) = A(I+2) + B(I+2) * C
  A(I+3) = A(I+3) + B(I+3) * C
ENDDO
```

  Each iteration of the loop now computes 4 values of A instead of 1 value.

  The compiler completely unrolls loops with iteration counts determined at compile time to be < 5. Loops with iteration counts >= 5 or undeterminable are partially unrolled.





## Optimization at -O2 *(contd.)*

- **Loop unrolling** *(contd.)*

  **Loop unrolling directives/pragmas**

  **UNROLL**

  **Fortran**

```
C$DIR UNROLL[(UNROLL_FACTOR=n)]
```

  **C**

```
#pragma _CNX unroll[(unroll_factor=n)]
```

  Unrolling is performed on innermost loops. If you use the directive or pragma on a loop nest, you must specify it on the loop which ends up, after any compiler transformations, to be innermost.

  **Loop unrolling compiler options**

  -*ur* - enables loop unrolling (compiler default)

  -*nur* - disables loop unrolling

  -*urn n* - specify the unroll factor *n* for all unrolled loops

## Optimization at -O2 *(contd.)*

- **Loop unroll and jam**

  Performed on a nested loop, it involves partially unrolling one or more loops higher in the nest than the innermost loop and fusing or "jamming" the resulting loops back together.

  Allows for increased register exploitation and a reduction of memory loads and stores per iteration of the innermost loop in the nest.

  To be effective, a loop must be nested and must contain data references that can be temporally reused with respect to some loop other than the innermost.

  Blocked loops are prime candidates.

  **Example:** matrix multiply loop (1000 x 1000)

```
DO I = 1, 1000
  DO J = 1, 1000
    DO K = 1, 1000
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

  Temporal reuse is achieved on A(I,K) references with respect to the J loop, and on B(K,J) references with respect to the I loop. Therefore, both the I and J loops are good candidates for unroll and jam.

Scalar opt



## Optimization at -O2 *(contd.)*

- **Loop unroll and jam** *(contd.)*

  The loop is blocked as shown below:

```
DO IOUT = 1, 1000, IBLOCK
  DO KOUT = 1, 1000, KBLOCK
    DO J = 1, 1000
      DO I = IOUT, IOUT+IBLOCK-1
        DO K = KOUT, KOUT+KBLOCK-1
          C(I,J) = C(I,J) + A(I,K)*B(K,J)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

  As written, the compiler can use 3 registers, one for each C(I,J), A(I,K) and B(K,J). Unroll and jam can increase this. First the compiler unrolls the I and J loops.



## Optimization at -O2 *(contd.)*

- **Loop unroll and jam** *(contd.)*

  The I and J loops are unrolled as shown below:

```
DO IOUT = 1, 1000, IBLOCK
 DO KOUT = 1, 1000, KBLOCK
  DO J = 1, 1000, 2
   DO I = IOUT, IOUT+IBLOCK-1, 2
    DO K = KOUT, KOUT+KBLOCK-1
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    ENDDO
    DO K = KOUT, KOUT+KBLOCK-1
      C(I+1,J)=C(I+1,J)+A(I+1,K)*B(K,J)
    ENDDO
   ENDDO
   DO I = IOUT, IOUT+IBLOCK-1, 2
    DO K = KOUT, KOUT+KBLOCK-1
      C(I,J+1)=C(I,J+1)+A(I,K)*B(K,J+1)
    ENDDO
    DO K = KOUT, KOUT+KBLOCK-1
      C(I+1,J+1)=C(I+1,J+1)+A(I+1,K)
                               *B(K,J+1)
    ENDDO
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

Scalar opt



## Optimization at -O2 *(contd.)*

- **Loop unroll and jam** *(contd.)*

  Next, the loops are "jammed" or fused back together:

```
DO IOUT = 1, 1000, IBLOCK
 DO KOUT = 1, 1000, KBLOCK
  DO J = 1, 1000, 2
   DO I = IOUT, IOUT+IBLOCK-1, 2
    DO K = KOUT, KOUT+KBLOCK-1
     C(I,J)=C(I,J)+A(I,K)*B(K,J)
     C(I+1,J)=C(I+1,J)+A(I+1,K)*B(K,J)
     C(I,J+1)=C(I,J+1)+A(I,K)*B(K,J+1)
     C(I+1,J+1)=C(I+1,J+1)+A(I+1,K)
                                *B(K,J+1)
    ENDDO
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

  As a result, the new loop exploits more registers and requires fewer loads and stores than the original. Now, the compiler can use 8 registers, one for each C(I,J), A(I,K), B(K,J), C(I+1,J), A(I+1,K), C(I,J+1), B(K,J+1) and C(I+1,J+1). Fewer loads and stores per iteration of the K loop are required because all of the registers containing these elements are referenced twice.

## Optimization at -O2 *(contd.)*

- **Loop unroll and jam** *(contd.)*

**Loop unroll and jam directives/pragmas for individual loop nests**

**UNROLL_AND_JAM**

**Fortran**

```
C$DIR UNROLL_AND_JAM[(UNROLL_FACTOR=n)]
```

**C**

```
#pragma _CNX
      unroll_and_jam[(unroll_factor=n)]
```

**NO_UNROLL_AND_JAM**

**Fortran**

```
C$DIR NO_UNROLL_AND_JAM
```

**C**

```
#pragma _CNX no_unroll_and_jam
```

Unroll and jam is only performed on nested loops. If you use the directive or pragma on a loop nest, you must specify it on the loop which ends up, after any compiler transformations, to be non-innermost.

Scalar opt



## Optimization at -O2 *(contd.)*

- **Loop unroll and jam** *(contd.)*

**Loop unroll and jam compiler options**

*-uj* - enables loop unroll and jam (compiler default)

*-nuj* - disables loop unroll and jam

*-ujn n* - specify the unroll factor *n* for all loops the compiler unrolls and jams



## Optimization at -O2 *(contd.)*

- **IF - DO and IF-FOR optimizations**

Minimizing the number of tests within a loop by removing or promoting them out of the loop.

Performance improvement is obtained by reducing the number of branches.

**Redundant-test elimination**

```
DO I = 1,N
  IF (I.GT.0) THEN
    DO J = 1,I
      A(I,J) = 0.
    ENDDO
  ENDIF
ENDDO
```

The explicit test is redundant, since the test is implicit in the DO loop, so it is removed:

```
DO I = 1,N
    DO J = 1,I
      A(I,J) = 0.
    ENDDO
ENDDO
```

Scalar opt



## Optimization at -O2 *(contd.)*

- **IF - DO and IF-FOR optimizations** *(contd.)*

**Loop boundary-value peeling**

Removing first, last, or first and last iterations of a loop to remove conditional tests from the loop.

Done when the loop containing a test involving an explicit reference to the loop index variable always evaluates identically for the first, last or first and last iterations.

```
DO I = 1,100
  IF (I .EQ. 1) THEN
    A(I) = B(I)
  ELSE IF (I .EQ. 100) THEN
    A(I) = C(I)
  ELSE
    A(I) = -A(I)
  ENDIF
ENDDO
```

After peeling:

```
A(1) = B(1)
 DO I = 2,99
    A(I) = -A(I)
 ENDDO
A(100) = C(100)
```

First and last tests are peeled off.

## Optimization at -O2 *(contd.)*

- **IF - DO and IF-FOR optimizations** *(contd.)*

  **Loop boundary-value peeling directives/pragmas**

  **PEEL, PEEL_ALL, NO_PEEL**

  **Fortran**

```
C$DIR PEEL
C$DIR PEEL_ALL
C$DIR NO_PEEL
```

  **C**

```
#pragma _CNX peel
#pragma _CNX peel_all
#pragma _CNX no_peel
```

  **Loop boundary-value peeling compiler options**

  *-peel* - increases the limit of code expansion (up to a conservative limit - compiler default)

  *-peelall* - code expansion without bound

  *-nopeel* - disables boundary-value peeling

Scalar opt



## Optimization at -O2 *(contd.)*

- **IF - DO and IF-FOR optimizations** *(contd.)*

  **Test promotion**

  Promoting a test out of a loop by replicating the containing loop(s) for each branch of the test. Replicated loops contain fewer tests than the original loops or no tests at all, so loops execute much faster.

  Can greatly increase the size of the code.

  Compiler will do test promotion up to a conservative limit. This can be controlled via directives or compiler options.

  **Example:**

```
DO I = 1,N
  IF (NA .EQ. NB) THEN
    A(I) = B(I)
  ELSE
    A(I) = 0.
  ENDIF
ENDDO
```

  Test promotion produces the following code:



## Optimization at -O2 *(contd.)*

- **IF - DO and IF-FOR optimizations** *(contd.)*

  **Test promotion** *(contd.)*

```
IF (NA .EQ. NB) THEN
  DO I = 1,N
    A(I) = B(I)
  ENDDO
ELSE
  DO I = 1,N
    A(I) = 0.
  ENDDO
ENDIF
```

Scalar opt



## Optimization at -O2 *(contd.)*

- **IF - DO and IF-FOR optimizations** *(contd.)*

  **Test promotion directives/pragmas**

  **PROMOTE _TEST, PROMOTE_TEST_ALL,**

  **NO_PROMOTE_TEST**

  **Fortran**

```
C$DIR PROMOTE_TEST
C$DIR PROMOTE_TEST_ALL
C$DIR NO_PROMOTE_TEST
```

  **C**

```
#pragma _CNX promote_test
#pragma _CNX promote_test_all
#pragma _CNX no_promote_test
```

  **Test promotion compiler options**

  *-ptst* - increases the limit of code replication (up to a conservative limit - compiler default)

  *-ptstall* - all tests promoted regardless of code replication

  *-noptst* - disables test promotion

## Optimization at -O2 *(contd.)*

- **Scalar replacement**

  Stores **loop-invariant** array elements in scalars, which can then be stored in registers by global register allocation (GRA).

  Substantially increases the performance of the loop by eliminating the necessity of accessing the element in main memory or the cache.

  **Example:**

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

  Here, **A(I)** is invariant with respect to the **J** loop. The compiler can therefore store **A(I)** in a scalar before entering the **J** loop and store the scalar back to **A(I)** between iterations of **I**. GRA can then assign this scalar to a register for the duration of the iteration, further improving efficiency.

Scalar opt



## Optimization at -O2 *(contd.)*

- **Scalar replacement** *(contd.)*

  Employing scalar replacement with GRA, the compiler generates code equivalent to:

```
DO I = 1, N
  REG = A(I)    ! put A(I) in register
  DO J = 1, M
    REG = REG + B(J)
  ENDDO
  A(I) = REG   !store back to A(I)
ENDDO
```

  Scalar replacement increases performance for this kind of loop by about 50%.

  The compiler will not attempt scalar replacement if the loop-invariant array element is aliased or if the array element is contained in conditional code.

  **Scalar replacement compiler options**

  *-sr* - enables scalar replacement (compiler default)

  *-nsr* - disables scalar replacement



## Optimization at -O2 *(contd.)*

- **Unlimiting loop optimizations**

  In subroutines and functions with lots of loops or IF-tests, some -O2 optimizations may be limited to control compile times.

  *-mrl (*More Replication of Loops*)* compiler flag:

  - prevents the compiler from limiting -O2 optimizations
  - much better execution times may be observed
  - for large codes, significantly longer compile times can be expected, and the compiler may actually run out of memory exceeding the space for its internal tables with the error:

    *Error: out of memory; cannot continue*

    To workaround this, invoke the compiler under mpa giving it more data space for its tables:

    **mpa -DATA /usr/convex/bin/fc foo.f**

Scalar opt



## Inhibitors of localization

- **Loop-carried dependencies**
- **Aliased scalar or array variables**
- **Computed or assigned GO TO statements in Fortran**
- **Multiple loop entries or exits**
- **RETURN or STOP statements in Fortran**
- **Procedure calls**
- **I/O statements**

## Inhibitors of localization *(contd.)*

- **Loop-carried dependencies (LCD)**

  When one iteration of a loop assigns a value to an address referenced or assigned on another iteration.

  In some cases an LCD can inhibit interchange, thereby inhibiting localization.

  **Example** of interchange-inhibiting LCD:

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

  This loop computes the elements of the current row of **A** using the elements of the previous row of **A**.

  As written, this loop uses **A(I-1,J+1)** to compute **A(I,J)**.

  Interchanging the **I** and **J** loops yields the next code:

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

  **Tables 1** and **2** show the sequence in which values of **A** are computed for the loops in each of these examples.

Scalar opt



## Inhibitors of localization *(contd.)*

- **Loop-carried dependencies (LCD)** *(contd.)*

  **Table1.** Computation sequence of **A(I,J)** - original loop

| I | J | A(I,J) | A(I-1,J+1) |
|---|---|---|---|
| 2 | 1 | A(2,1) | A(1,2) |
| 2 | 2 | A(2,2) | A(1,3) |
| 2 | 3 | A(2,3) | A(1,4) |
| ... | ... | ... | ... |
| 3 | 1 | A(3,1) | A(2,2) |
| 3 | 2 | A(3,2) | A(2,3) |
| 3 | 3 | A(3,3) | A(2,4) |
| ... | ... | ... | ... |

  **Table2.** Computation sequence of **A(I,J)** - interchanged loop

| I | J | A(I,J) | A(I-1,J+1) |
|---|---|---|---|
| 2 | 1 | A(2,1) | A(1,2) |
| 3 | 1 | A(3,1) | A(2,2) |
| 4 | 1 | A(4,1) | A(3,2) |
| ... | ... | ... | ... |
| 2 | 2 | A(2,2) | A(1,3) |
| 3 | 2 | A(3,2) | A(2,3) |
| 4 | 2 | A(4,2) | A(3,3) |
| ... | ... | ... | ... |

  In this case, interchanging the loops would yield wrong answers.



## Inhibitors of localization *(contd.)*

- **Loop-carried dependencies (LCD)** *(contd.)*

  **Directives and pragmas:**

  Used to indicate to the compiler that the loop is free of dependencies, so that it will interchange the loop.

  **NO_LOOP_DEPENDENCE**

  **Fortran**

  **C$DIR NO_LOOP_DEPENDENCE*(array-namelist)***

  **C**

  **#pragma _CNX no_loop_dependence*(array-namelist)***

Scalar opt



## Inhibitors of localization *(contd.)*

- **Aliasing**

  When two or more names are attached to the same memory location.

  Typically caused by EQUIVALENCE in Fortran and use of pointers in C.

  **Fortran EQUIVALENCE example:**

```
INTEGER A(100,100),B(100,100),C(100,100)
EQUIVALENCE (A,B)
DO I = 1,N
  DO J = 2,M
    A(I,J) = B(I-1,J+1) + C(I,J)
  ENDDO
ENDDO
```

  **A** and **B** refer to the same array - LCD on **A.**

  This prevents interchange, thus interfering with localization.

## Inhibitors of localization *(contd.)*

- **Aliasing** *(contd.)*

  **C Example:**

  (The C equivalent of the Fortran EQUIVALENCE example)

```
int a[100][100], c[100,100], i, j;
int (*b)[100];
b=a;
.
.
.
for(j=0;j<n;j++){
   for(i=1;i<m;i++){
     a[j][i] = b[j+1][i-1] + c[j][i];
   }
}
```





## Inhibitors of localization *(contd.)*

- **Multiple loop entries or exits**

  Inhibit data localization because they cannot be safely interchanged (the order of computation changes if loops are interchanged).

  -Extra entries usually created when a loop contains a branch destination.

  -Extra exits usually created in **Fortran** using the **GO TO** statement and in **C** using the **break** statement.

  **Fortran example:**

```
      DO I= 1,M
        DO J= 1,N
          A(I,J) = B(I,J) + C(I,J)
          IF(A(I,J).EQ.0) GO TO 50
      ...
        ENDDO
      ENDDO
      ...
  50  CONTINUE
```

  In this example, the order of computation changes if the loops are interchanged.



## Inhibitors of localization *(contd.)*

- **I/O statements**

  The order in which values are read into or written from a loop may change if the loop is interchanged.

  Interchange is inhibited, therefore data localization is inhibited.

- **RETURN or STOP statements in FORTRAN**

  Inhibit loop interchange, thus inhibiting localization (the order of computation is changed if loops are interchanged).

- **Procedure calls**

  Compilers are unaware of the side effects of most procedures, so can’t determine if they might interfere with loop interchange.





## Exercises

**[1]** Can the compiler do data localization in this example? If not, why not? If yes, should this be allowed?

```
...
CALL ALI(A,A,C)
...
SUBROUTINE ALI(A,B,C)
INTEGER A(100,100),B(100,100),C(100,100)
DO I = 2,M
   DO J = 1,N
     A(I,J) = B(I-1,J+1) + C(I,J)
   ENDDO
ENDDO
...
```

**[2]** Same question: C equivalent of the previous Fortran exercise.

```
...
ali(&a,&a,&c)
...
void ali(a,b,c)
int a[100][100],b[100][100],c[100][100];
{
  int i,j;

  for(i=1;i<m;i++){
    for(j=0;j<n;j++){
     a[j][i] = b[j+1][i-1] + c[j][i];
    }
  }
}
```

# Automatic parallelism

# 9

**Topics:**

- **-O3 optimization**
- **Inhibitors of parallelization**
- **Data dependence in loops**
- **Automatically parallelized loops**
- **Exercises**





## -O3 optimization

- Primary goal of -O3 optimizations is *parallelization*
  - Divides a program into threads
  - A *thread* is a sequence of instructions that execute on a single CPU
- Parallelism can exist at both the **loop** and **task** level
  - Compilers automatically exploit *loop* level in which all loops are examined:
    - Explicitly coded loops
    - Fortran 90 array expressions
  - Task level parallelism can be created by the user using the **BEGIN_TASK**, **NEXT_TASK** and **END_TASKS** directives discussed later





## -O3 optimization - *cont.*

- Requirements for **automatic** loop parallelization:
  - Loop must **not** contain any **data dependencies**
  - Loop must have a known iteration count at runtime
  - Loop nest has sufficient parallel work to be done
  - Loop must not contain any I/O statements
  - Loop must not contain multiple entries or exits (includes STOP and RETURN statements)
  - Loop must not contain any procedure calls other than intrinsic functions
  - Loop must not contain potentially aliased scalar or array variables
- When a loop is automatically parallelized:
  - The loop is divided up into several smaller iteration spaces parceled out to be run simultaneously on all available processors
  - Compiler will try to parallelize the outermost loop
  - Compiler takes care of privatization of loop variables as needed, for example, the loop index is always privatized

## -O3 optimization- *cont.*

- Compiler creates following types of parallelism:
  - thread-way (one-dimensional parallelism)
  - node-way (two-dimensional parallelism)
  - *-or all* compiler flag shows the complete optimization report including the variables privatized
- Executable code generated will automatically run on as many processors as are available at runtime without recompilation
  - Normally all processors of the subcomplex
  - Smaller number of processors specified via:
    - **mpa(1)** utility
    - **LOOP_PARALLEL(max_threads=m)** and **PREFER_PARALLEL(max_threads=m)** compiler directives and pragmas which limit threads
    - **+min**/**+max** loader options





## -O3 optimization- *cont.*

- Thread activity:
  - Shared memory program runs as a collection of threads on multiple processors
  - At program initiation, a separate thread of execution is started on each of the processors in the subcomplex
  - All threads spin for a default amount of time and then go idle except for thread 0 which runs all of the serial code of the program
  - When thread 0 encounters a parallel loop or task, it wakes-up or "spawns" the other threads signalling them to begin execution of the parallel code
  - The spawned threads then become active acquiring spawned thread IDs (1 to numprocs-1), run until their portion of the parallel code is finished or until they get time-sliced out, and then spin and go idle once again. Thread 0 also participates running its portion of the parallel code.
  - All spawned threads execute to completion of their spawned context before thread 0 continues



## -O3 optimization - *cont.*

- **Thread-way parallelism** - Encountered by thread 0, it causes all threads available to the application to participate. Each of these threads is assigned a portion of the loop iteration space to execute.

  Consider the following loop:

```
DO I = 1, 1024
   A(I) = B(I) + C(I)
ENDDO
```

  - If run on 8 processors, each thread will execute:

    **`1024/8 = 128`** iterations
  - If run on 128 processors, each thread will execute:

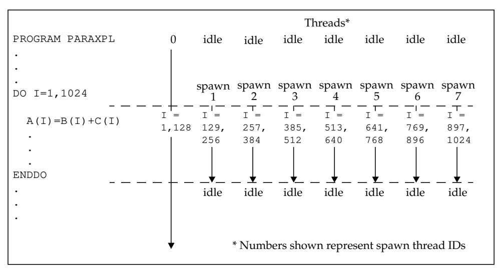
    **`1024/128 = 8`** iterations
  - Compiler transforms the loop such that the starting and stopping iteration values for each thread are determined at runtime based on the number of available processors
  - If the iteration count is not evenly divisible by the number of threads, some threads perform fewer iterations than others

Auto parallel



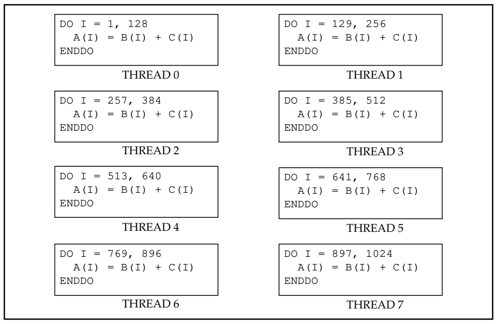
## -O3 optimization - *cont.*

This figure shows **thread activity** and the **parceling of loop iterations** for the previous loop running on 8 processors.



## -O3 optimization - *cont.*

This figure shows the loop parallelized running on 8 processors.




Auto parallel



## -O3 optimization - *cont.*

- **Node-way parallelism** - Encountered by thread 0, it causes 1 thread per hypernode available to the application to participate. Each of these threads is assigned a portion of the node-way loop iteration space to execute. However, node-way automatic parallelism will never be created, unless the compiler finds a thread-way parallel inner loop or an opportunity for thread-way parallelism via a function call.

  Thread-way loop parallelism encountered within a node-way parallel construct, causes each thread within the hypernode to be assigned a portion of the thread-way loop iteration space to execute.

  Consider the following loop:

```
DO J = 1, 1024
  DO I = 1, 1024
    A(I,J) = B(I,J) + C(I,J)
    .
    .
  ENDDO
ENDDO
```

  - Assuming no inhibitors and there is enough work, the compiler automatically parallelizes the J loop across hypernodes and the I loop across threads within those hypernodes.



## -O3 optimization - *cont.*

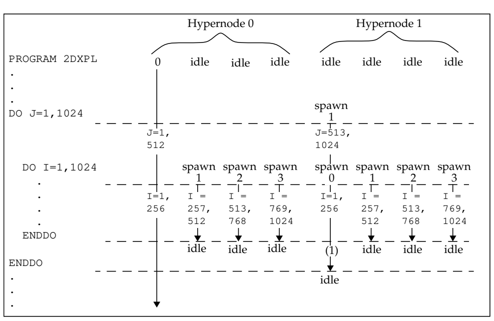
- If run on a 2 4-processor hypernode subcomplex:

  The node-way J loop spawns 2 threads, one on each hypernode. Each of these threads will execute:

  **`1024/2 = 512`** iterations of the J loop

  The I loop then spawns thread-way parallelism within each hypernode. Each of these threads will execute:

  **`1024/4 = 256`** iterations of the I loop




Auto parallel



## -O3 optimization - *cont.*

- Node-way parallelism can be disabled by specifying the *-nonodepar* compiler flag
  - Disables automatic and directive-specified node-way parallelism
  - Automatic and directive-specified thread-way parallelism still enabled
  - Eliminates node-way parallel overhead on a single node subcomplex
- Node-way and thread-way automatic parallelization can be disabled by specifying the *-noautopar* compiler flag:
  - Directive-specified node-way and thread-way parallelism still enabled through the usage of PREFER_PARALLEL, LOOP_PARALLEL, and BEGIN_TASKS compiler directives.
  - All other loops treated as if the NO_PARALLEL directive was specified for them

## Inhibitors of parallelization

Most constructs that inhibit data localization also inhibit parallelization for the same reason. Specifically:

- Loop carried dependencies (LCDs). More categories of LCDs can inhibit parallelization than data localization:
  - Backward LCDs (B-LCD)
  - Forward LCDs (F-LCD)
  - Output LCDs (O-LCD)
  - Apparent LCDs (A-LCD)

  Examples of each will be given in the following **Data dependence in loops** section
- Potential for aliased scalar or array variables
- Multiple loop entries or exits (includes STOP and RETURN statements)
- Procedure calls other than intrinsic functions
- I/O statements

More inhibitors:

- Insufficient amount of parallel work to be done in loop
- Loop iteration count unknown at runtime





## Data dependence in loops

- A loop-carried-dependence (LCD) results from an address being assigned a value in one loop iteration and the same address being assigned or referenced in another iteration.

  Example:

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

  An example of a **Backward LCD (B-LCD)**:

  - The **A(I-1)** reference on iterations 3 through **N** was assigned on the previous iteration as **A(I)**
  - Each iteration must execute to completion before the next can begin. Therefore it is fruitless to assign parallel threads of execution to compute different iterations
  - B-LCDs cannot be automatically parallelized



## Data dependence in loops - *cont.*

Example:

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

An example of a **Forward LCD (F-LCD)**:

- The **A(I+1)** referenced on iterations 2 through **N-1** is assigned by the following iteration
- If parallel threads of execution attempt to execute different iterations of the loop, it is quite possible, for example, that **A(3)** might be assigned in iteration 3 before **A(3)** is referenced by iteration 2
- F-LCDs cannot be automatically parallelized
- The example can be automatically parallelized by making an extra copy of array **A**:

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

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

  Creating the 2nd loop clearly adds overhead. Therefore, it must be used with care.





## Data dependence in loops - *cont.*

Example:

```
DO I = 2,N
   A(J(I)) = B(I)
ENDDO
```

An example of a *potential* **Output LCD (O-LCD)**:

- If **J(I)** contains repeated values (i.e., **J(3)** = **J(7)** = 4), then 2 different iterations are attempting to assign a value to the same address
- If parallel threads are executing the iterations, then the values *output* by the loop into array **A** depend on the order in which the iterations are executed
- The compiler will not automatically parallelize such loops

## Data dependence in loops - *cont.*

Example:

```
DO I = 1,N
   A(I) = A(J(I)) + 1.0
ENDDO
```

This is an example of an **Apparent LCD (A-LCD)**:

- Since the value assigned to **A(I)** in one iteration *might* be used in a later iteration, the compiler lacks sufficient information to determine whether an LCD exists or not
- Rather than risk wrong answers, the compiler will not automatically parallelize such loops

Summary: the compiler does not automatically parallelize loops containing actual or apparent array based LCDs.

Auto parallel



## Automatically parallelized loops

Following are examples of loops that are automatically parallelized by the compiler at -O3. All examples assume that there is no aliasing among the arrays.

```
SUBROUTINE MYSUB(A,B,C,N)

REAL*4 A(N,N), B(N,N), C(N,N)

DO J = 1, N
   DO I = 1, N
      A(I,J) = B(I,J) + C(J,I)
   ENDDO
ENDDO

RETURN

END
```

The above nested loop will automatically parallelize, since there are no LCDs.



## Automatically parallelized loops - *cont.*

The compiler can handle some *scalar* LCDs.

```
SUBROUTINE MYSUB(A,B,Y,N)

REAL*4 A(N), B(N), Y(N), X(N), S

J = 5
DO I = 1, N
   S = A(I) * B(I)
   J = J + 1
   X(I) = S * Y(J)
ENDDO

RETURN

END
```

- If parallel threads execute different iterations, they can overwrite each other's values of **S** (O-LCD)
- The compiler avoids this problem by providing a private version of **S** for each thread
- If the value of **S** is needed after the loop, the processor executing the **N**th iteration stores its private value in **S**
- **J**, a loop induction variable like **I**, has the same problem as **S** (O-LCD) plus is a B-LCD. A private version of **J** is provided to each thread and its value is determined as a function of **I**:
(**J** = **Jinit** + **I**, where **Jinit** = **J** at loop invocation)

Auto parallel



## Automatically parallelized loops - *cont.*

The compiler can handle some *scalar* LCDs known as *reductions*. Generally, a reduction has the form:

**X** = **X** *operator* **Y**

where:

**X** - variable not assigned or used elsewhere in the loop
**Y** - a loop constant expression not involving **X**
*operator* is +, -, *, .AND., .OR., .EQV. or .NEQV.

```
SUBROUTINE MYSUB(A,B,C,D,E,N,SUM)

REAL*8 A(N),B(N),C(N),D(N),E(N),SUM
INTEGER*4 N

DO I = 1, N
   A(I) = E(I) / C(I)
   SUM = SUM + A(I) * B(I)   !reduction
   D(I) = A(I) / B(I)
ENDDO

RETURN
END
```

- If parallel threads execute different iterations, they can overwrite each other's values of **SUM** (O-LCD)
- The compiler avoids this problem by providing a private version of **SUM** for each thread in which each thread's partial sum is computed. The private **SUM**s are added to compute final **SUM** by the compiler.

## Automatically parallelized loops - *cont.*

The compiler also recognizes scalar reductions of the form:

**X =** *function* **(X, Y)**

where:

**X** - variable not assigned or used elsewhere in the loop
**Y** - a loop constant expression not involving **X**
*function* is the intrinsic MAX or MIN function

```
SUBROUTINE MYSUB(A,B,C,D,E,N)

REAL*8 A(N),B(N),C(N),D(N),E(N)
REAL*8 MAX, X
INTEGER*4 N

X = 0.0

DO I = 1, N
   A(I) = E(I) / C(I)
   X = MAX(X,A(I))        !reduction
   D(I) = A(I) / B(I)
ENDDO
RETURN
END
```

- If parallel threads execute different iterations, they can overwrite each other's values of **X** (O-LCD)
- The compiler provides a private version of **X** for each thread. Thread 0 uses private **X**s to set original **X**.





## Automatically parallelized loops - *cont.*

Even entire arrays can be privatized as needed for the array **Q** in order to parallelize the **J** loop in the example below:

```
SUBROUTINE MYSUB(U,M)

PARAMETER (N = 99)

REAL*4 P(N), Q(N), R(N), U(M), T(N,M)

DO J = 1, M
   DO I = 1, N
      Q(I) = P(I) * R(I)
      T(I,J) = Q(I) * U(J)
   ENDDO
ENDDO

RETURN

END
```

- If parallel threads execute different iterations of **J**, they can overwrite each other's values of **Q(I)** (O-LCD)
- The compiler avoids this problem by providing a private version of the **Q** array for each thread, since it knows **Q**'s size at compile time
- Parallelized, each thread sets a unique subarray of **T**
- Compiler also provides a private version of **I** for each thread



## Automatically parallelized loops - *cont.*

```
SUBROUTINE MYSUB

PARAMETER (M = 50, N = 10)

REAL*4 X(100,100), Y(100,100)

Y(2:M+1:2, 2:N+1) = 1.0
X(1:M:2, 1:N) = Y(2:M+1:2, 2:N+1)

RETURN

END
```

The above Fortran 90 array assignments will automatically parallelize, creating 2 parallel loops, since by definition there are no dependencies.





## Exercises

**[1]** Can the compiler automatically parallelize any of these loops?

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


(b) DO I = 1, N
      IF (A(I) .LT. 0.0) THEN
        J = J + 1
        A(I) = B(J)
      ENDIF
    ENDDO


(c) DO WHILE (A(I) .LT. Z)
      I = I + 1
      A(I) = B(I)
    ENDDO


(d) DO WHILE (I .LT. N)
      I = I + 1
      A(I) = B(I)
    ENDDO


(e) DO I = 1, N
      S = C(I) * B(I)
      IF (S .GT. 0.D0) THEN
        T = S + 5.D0
      ELSE
        T = S + 4.D0
      ENDIF
      A(I) = A(I) + T
    ENDDO


(f) DO I = 1, N
      A(J(I)) = A(K(I)) + 1
    ENDDO
```

# Assisted automatic parallelism

# 10

**Topics:**

- No_loop_dependence
- Loop_private
- Save_last
- No_parallel
- Exercises





## NO_LOOP_DEPENDENCE

- Fortran

```
C$DIR NO_LOOP_DEPENDENCE(array-
                           namelist)
```

- C

```
#pragma _CNX
  no_loop_dependence(array-namelist)
```

- Resolves dependencies on subscripted array
- Tells compiler that in immediately following loop, it can safely ignore any potential loop-carried dependencies on array(s) specified in *array-namelist*
- Arrays appearing in the list should have at least one assignment in the loop that has a subscript expression depending directly or indirectly on the loop index of the loop that goes parallel
- As safeguard, compiler will emit warnings about each dependence removed as result of directive
- *array-namelist* required
- Example - correct usage

```
C$DIR NO_LOOP_DEPENDENCE(A)

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

Asst auto parallel



## LOOP_PRIVATE

- Fortran

```
C$DIR LOOP_PRIVATE(namelist)
```

- C

```
#pragma _CNX loop_private(namelist)
```

- Declares variables and arrays in *namelist* private to immediately following loop
- Each thread of execution is provided with its own private copy of these variables for the duration of the loop
- No starting values can be assumed. The variables must be assigned in the loop by the thread before they are used in the loop
- Final values are not available at loop termination unless SAVE_LAST directive is used
- Compiler will assume that these variables have no loop carried dependencies. If they do, wrong answers could result

## LOOP_PRIVATE - *cont*

- Example

```
C$DIR LOOP_PRIVATE(S)
      DO I = 1,N
        IF (A(I) .GT. 0.) S = A(I)
        IF (Q(I) .LT. P(I)) S = P(I)
        IF (Z(I) .LE. Y(I)) S = Z(I)
        B(I) = S * C(I) + D(I)
      ENDDO
```

This example has a potential recurrence unless at least one of the IF tests is true on each iteration. That is, the value of S can wrap around (backward LCD) from one iteration to the next. LOOP_PRIVATE implies there is no wrap around. If a backward LCD does exist, wrong answers could occur.

Asst auto parallel



## SAVE_LAST

- Fortran

```
C$DIR SAVE_LAST
```

- C

```
#pragma _CNX save_last
```

- Tells compiler that in immediately following loop, all arrays and variables declared to be LOOP_PRIVATE will have their values from final iteration saved for use after loop termination
- Can only be reliable for variables that are assigned on final iteration of loop
  - Variables that are conditionally assigned but not assigned on the last iteration of the loop should use ORDERED_SECTION directives to save their values



## SAVE_LAST - *cont*

- Example

```
C$DIR LOOP_PRIVATE(ATMP), SAVE_LAST
      DO I = 1,N
        IF (J .EQ. I) ATMP = A(I)
        IF (J .NE. I) ATMP = C(I)
        S = B(I) + C(I)
        A(I) = B(I) + C(I)
        B(I) = ATMP * S
      ENDDO
      IF (ATMP .GT. MAX) THEN
        CALL DOIT(ATMP)
      ENDIF
      J = S
```

The SAVE_LAST directive is used because the value of ATMP is needed later for comparison. Note that ATMP is assigned on the last iteration of the loop.

Now, the compiler parallelizes the loop. Note, it automatically privatizes S as it would have if it had automatically parallelized the loop.

Asst auto parallel



## NO_PARALLEL

- Fortran

```
C$DIR NO_PARALLEL
```

- C

```
#pragma _CNX no_parallel
```

- Suppresses parallelization of the immediately following loop
- Other loop optimizations, like loop interchange, loop distribution, etc., are unaffected
- Useful when automatically parallelized loop appears to be running slower than it would in serial mode
- Example:

```
C$DIR NO_PARALLEL
      DO I = 1,N
        A(I) = B(I) * C(I)
      ENDDO
```

Supposing N = 3 in this example, it might be useful to use the NO_PARALLEL to stop the loop from parallelizing.

## Exercises

**[1]** What is stopping automatic parallelization of the following loops? How could they be restructured to cause them to parallelize?

```
(a) DO I = 1,N
      IF (A(I) .GT. 0.0) S2 = A(I)
      IF (B(I) .GT. 0.0) S2 = B(I)
      A(I) = S2 * B(I)
    ENDDO
```

```
(b) DO I = 1,N
      A(J(I)) = B(J(I)) + C(J(I))
    ENDDO
```

Asst auto parallel



**[2]** The following loops parallelize but run slowly. Why? How could they be fixed?

```
(a) REAL*8 A,B,C
    COMMON /ABC/ A(256,256), B(256,256), C(256,256)
    DO J = 1,N
      DO I = 1,N
        A(I,J) = B(I,J) + C(I,J)
      ENDDO
    ENDDO
```

```
(b) REAL *8 X,Y,Z
    COMMON /ABC/ X(1024),Y(512),Z(1024,512)
    DO J = 1,N
      DO I = 1,M
        X(J) = X(J) + Y(I) * Z(J,I)
      ENDDO
    ENDDO
```



# Synchronization library 11

**Topics:**

- Gate and Barrier data types
- Synchronization library
- Allocating a gate
- Freeing a gate
- Locking a gate
- Locking a gate conditionally
- Unlocking a gate
- Allocating a barrier
- Freeing a barrier
- Waiting on a barrier

## Gate and Barrier data types

- Used for manual synchronization
- Gates provide mechanism for restricting execution of code block to single thread at a time
- Gates are usually used around code blocks that update shared variable(s) within parallel constructs
- Barriers block further execution until all executing threads reach the barrier
- Variable declarations are actually pointers to synchronization structures created with intrinsic routines ALLOC_GATE and ALLOC_BARRIER
- Fortran specifics
    - May only appear
        - In COMMON statements
        - In DIMENSION statements, each on a separate statement
        - In type statements lexically preceding directive declarations
        - Only where Fortran “type” declarations are legal
        - As dummy and actual arguments to synchronization routines

Synch library



## Gate and Barrier data types - *cont*

- Of type INTEGER*4
- May only be initialized by passing them as arguments to synchronization routines
- Directives override any preceding type declaration and once declared as a gate or barrier, a variable cannot be redeclared as another type or be made global via COMMON
- Error if the variables appear on other type statements following directives
- May not be initialized in DATA statements or appear in EQUIVALENCE statements
- GATE variables that appear in COMMON can only appear with other variables of type GATE and BARRIER variables can only appear in COMMON with other variables of type BARRIER
- Fortran Directives
    - **C$DIR GATE** (*namelist*)
    - **C$DIR BARRIER** (*namelist*)
    - namelist is a comma-delimited list of one or more gate or barrier names



## Gate and Barrier data types - *cont*

- C specifics
    - gate_t and barrier_t variables may only appear
        - In definition and declaration statements
        - As formal and actual arguments
    - Of type int (4 bytes)
    - May only be initialized and referenced by passing them as arguments to the synchronization intrinsic routines
    - C Data types
        - gate_t *namelist*
        - gate8_t *namelist*
        - barrier_t *namelist*
        - barrier8_t *namelist*
        - *namelist* is a comma-delimited list of one or more gate or barrier names
        - Must include spp_prog_model.h

Synch library



## Synchronization library

- Used to coordinate actions among threads
- All functions return zero (0) to indicate successful completion and minus one (-1) to indicate failure
- Synchronization functions
    - Have gate functions that
        - Allocate
        - Free
        - Lock
        - Conditionally lock
        - Unlock
    - Have barrier functions that
        - Allocate
        - Free
        - Wait on barrier
- Have 4 and 8 byte version of routines
    - 8-byte version used with compiler options that change the default data size to 8 bytes (e.g., -cfc,-p8)
    - Must be allocated and deallocated in same way

## Allocating a gate

- Fortran

```
INTEGER FUNCTION ALLOC_GATE(gate)
INTEGER*8 FUNCTION ALLOC_GATE(gate)
```

- C

```
int alloc_gate(gate_t* gate_p);
long long alloc_gate_8(gate8_t*
                       gate_p);
```

- Allocates gate and assigns its address to *gate*
- Gate is left in unlocked state
- Should be called prior to any other use of a gate
- Error to initialize a gate more than once without intervening call to FREE_GATE





## Freeing a gate

- Fortran

```
INTEGER FUNCTION FREE_GATE(gate)
INTEGER*8 FUNCTION FREE_GATE_8(gate)
```

- C

```
int free_gate(gate_t* gate_p);
long long free_gate_8(gate8_t*
                      gate_p);
```

- Deallocates gate structure and sets pointer *gate* to zero



## Locking a gate

- Fortran

```
INTEGER FUNCTION LOCK_GATE(gate)
INTEGER*8 FUNCTION LOCK_GATE_8(gate)
```

- C

```
int lock_gate(gate_t* gate_p);
long long lock_gate_8(gate8_t*
                      gate_p);
```

- Acquires a gate for exclusive access.
- If gate cannot be immediately acquired, calling thread waits, and may or may not relinquish processor while waiting





## Locking a gate conditionally

- Fortran

```
INTEGER FUNCTION COND_LOCK_GATE(gate)
INTEGER*8 FUNCTION
           COND_LOCK_GATE_8(gate)
```

- C

```
int cond_lock_gate(gate_t* gate_p);
long long cond_lock_gate_8
               (gate8_t* gate_p);
```

- Acquires a gate for exclusive access IF it can do so without waiting
- Returns 0 if the gate was acquired, -1 if not

## Unlocking a gate

- Fortran

```
INTEGER FUNCTION UNLOCK_GATE(gate)

INTEGER*8 FUNCTION
                UNLOCK_GATE_8(gate)
```

- C

```
int unlock_gate(gate_t* gate_p);

long long unlock_gate_8(gate8_t*
                          gate_p);
```

- Releases gate from exclusive access
- Normally done by thread that acquired the gate
    - Sometimes thread 0 may acquire a gate in a serial code region that is subsequently released by a different thread in a parallel region





## Allocating a barrier

- Fortran

```
INTEGER FUNCTION
             ALLOC_BARRIER(barrier)

INTEGER*8 FUNCTION
            ALLOC_BARRIER_8(barrier)
```

- C

```
int alloc_barrier(barrier_t*
                         barrier_p);

long long alloc_barrier_8(barrier_t*
                          barrier_p);
```

- Allocates a barrier structure and assigns its address to *barrier*
- Should be called prior to any other use of a barrier
- Error to initialize a barrier more than once without intervening call to FREE_BARRIER



## Freeing a barrier

- Fortran

```
INTEGER FUNCTION
             FREE_BARRIER(barrier)

INTEGER*8 FUNCTION
            FREE_BARRIER_8(barrier)
```

- C

```
int free_barrier(barrier_t*
                      barrier_p);

long long free_barrier_8(barrier_t*
                        barrier_p);
```

- Deallocates a barrier structure and sets the pointer *barrier* to 0





## Waiting on a barrier

- Fortran

```
INTEGER FUNCTION
      WAIT_BARRIER(barrier,nthreads)

INTEGER*8 FUNCTION
   WAIT_BARRIER_8(barrier,nthreads8)
```

- C

```
int wait_barrier(barrier_t*
      barrier_p,const int* nthreads);

long long wait_barrier_8(barrier_t*
         barrier_p,const long long*
         nthreads8);
```

- Each of the *nthreads* waits in the barrier until the final thread makes this reference, then all are released simultaneously
- Each of *nthreads* must reference this routine
- Barrier variables may be reused in other WAIT_BARRIER calls as long as programmer makes sure that two such barriers cannot be active simultaneously

# Manual parallelism

# 12

**Topics:**

- **Using manual parallelism**
- **Prefer_parallel**
- **Loop_parallel**
- **Using attribute list with parallel directives**
- **Begin_tasks,next_task,end_tasks**
- **Task_private**
- **Using attribute list with task directives**
- **Synchronization directives**
- **Nested directives example**
- **Exercises**







## Using manual parallelism

- When using all the directives below with the **exception** of PREFER_PARALLEL:
  - Compiler automatic parallelization is disabled
  - User must manage use of private memory variables and synchronize access to certain shared memory variables
  - User must hand code nested parallelism (node and thread)
- Directives to be discussed:
  - Fortran:
    - C$DIR LOOP_PARALLEL [(*attribute-list*)]
    - C$DIR PREFER_PARALLEL [(*attribute-list*)]
    - C$DIR BEGIN_TASKS [(*attribute-list*)]
    - C$DIR NEXT_TASK
    - C$DIR END_TASKS
    - C$DIR CRITICAL_SECTION [(*gate*)]
    - C$DIR END_CRITICAL_SECTION
    - C$DIR ORDERED_SECTION(*gate*)
    - C$DIR END_ORDERED_SECTION



## Using manual parallelism - *cont*

- C:
  - #pragma _CNX loop_parallel[(*attribute-list*)]
  - #pragma _CNX prefer_parallel[(*attribute-list*)]
  - #pragma _CNX begin_tasks [(*attribute-list*)]
  - #pragma _CNX next_task
  - #pragma _CNX end_tasks
  - #pragma _CNX critical_section [(*gate*)]
  - #pragma _CNX end_critical_section
  - #pragma _CNX ordered_section(*gate*)
  - #pragma _CNX end_ordered_section

## Using manual parallelism - *cont*

- optional *attribute-list* for PREFER_PARALLEL, LOOP_PARALLEL,BEGIN/NEXT/END_TASKS directives can contain one of the following:
  - THREADS - causes thread-way parallelism across the subcomplex
  - NODES - causes node-way parallelism. One single thread which executes a portion of the loop is started on each hypernode
  - MAX_THREADS = *m* - allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer which has a value at compile time.
  - ORDERED - causes ordered invocation of each loop iteration; provides no automatic synchronization
  - IVAR = *induction variable* - identifies the induction variable whose iteration space is partitioned among the threads. Optional for Fortran DO loops, required for loops in C.
  - ORDERED, NODES - cause ordered invocation of each iteration across hypernodes
  - ORDERED, THREADS - causes ordered invocation of each iteration across threads



## Using manual parallelism - *cont*

- optional *attribute-list* (*contd.*):
  - ORDERED, MAX_THREADS = *m* - ordered parallelism of no more than *m* threads
  - NODES, MAX_THREADS = *m* - node-way parallelism on no more than *m* hypernodes each running one thread
  - THREADS, MAX_THREADS = *m* - thread-way parallelism on no more than *m* threads
  - ORDERED, NODES, MAX_THREADS = *m* - ordered node-way parallelism on no more than *m* hypernodes
  - ORDERED,THREADS,MAX_THREADS=*m* - ordered thread-way parallelism on no more than *m* threads
- optional *attribute-list* for PREFER_PARALLEL, LOOP_PARALLEL directives only can also contain one of the following:
  - CHUNK_SIZE = *n* - divides loop into chunks of *n* iterations and distributes chunks round-robin to the processors
  - NODES, CHUNK_SIZE = *n* - node-way parallelism by chunks of size *n*





## Using manual parallelism - *cont*

- optional *attribute-list* (*contd.*):
  - THREADS, CHUNK_SIZE = *n* - thread- way parallelism in chunks of size *n*
  - CHUNK_SIZE=*n*, MAX_THREADS = *m* - chunks of size *n* are distributed round-robin to no more than *m* threads
  - NODES,CHUNK_SIZE=*n*, MAX_THREADS=*m* - node-way parallelism by chunks of size *n* on no more than *m* hypernodes
  - THREADS,CHUNK_SIZE=*n*, MAX_THREADS=*m* - thread-way parallelism by chunks of size *n* on no more than *m* threads
- In above combinations, the attributes can be listed in any order
- IVAR = *induction variable* can be used with any of the above combinations
- Other than adding IVAR attribute to the above list, there are no other valid combinations
  - Other combinations will be flagged as a fatal compiler error



## PREFER_PARALLEL

- Fortran syntax
  - C$DIR PREFER_PARALLEL [(*attribute-list*)]
- C syntax
  - #pragma _CNX prefer_parallel[(*attribute-list*)]
- **Advises** the compiler that it *should* automatically parallelize the immediately following loop *if* it can:
  - No LCDs
  - No procedure calls
- Requires less manual intervention and is less forceful than the LOOP_PARALLEL directive
- Retains all of the automatic creation of private variables
- Default behavior without an [(*attribute-list)*] - allows the compiler to pick either node-way (NODES) or thread-way (THREADS) parallelism, whichever is more beneficial.

## PREFER_PARALLEL - *cont*

Manual parallel

- Can be used to indicate the preferred loop in a nest to parallelize when the compiler is parallelizing a different loop than the programmer prefers
  - Example:

```
      DO J = 1,100
C$DIR PREFER_PARALLEL
        DO I = 1,100
          . . .
        ENDDO
      ENDDO
```

  Here, PREFER_PARALLEL causes the compiler to choose the innermost loop for parallelization, provided it is free of dependencies and procedure calls.

- ORDERED attribute provided to be consistent with LOOP_PARALLEL but of limited usefulness
  - ORDERED is only useful if the loop contains dependencies needing to be synchronized and PREFER_PARALLEL will not parallelize a loop with dependencies
  - ORDERED more useful with LOOP_PARALLEL



## LOOP_PARALLEL

- Fortran syntax
  - C$DIR LOOP_PARALLEL [(*attribute-list*)]
- C syntax
  - #pragma _CNX loop_parallel[(*attribute-list*)]
- **Forces** parallelization of the immediately following loop
- Compiler does not check for data dependencies or perform variable privatization; **users** must synchronize data dependencies and privatize data as needed
- Default behavior without an [(*attribute-list)*] - compiler performs thread-way (THREADS) parallelism
- Non-parallel optimizations (such as loop interchange) will be applied to the loop if applicable
- Does not apply to implicit loops of Fortran 90 array section syntax
  - Fortran 90 array assignments are data independent by nature and candidates for automatic parallelization



## LOOP_PARALLEL - *cont*

Manual parallel

- Can be useful for manually parallelizing loops containing procedure calls when the procedure called has no side effects:
  - Does not modify its scalar arguments
  - Does not modify any COMMON variables
  - Does not perform any I/O
  - Does not call any other procedures that have side effects
  - Example:

```
C$DIR LOOP_PARALLEL(IVAR = I)
      DO I = 1,N
        X(I) = FUNC(I)
      ENDDO
```

  Normally the call to FUNC in this example would prevent the loop from parallelizing. If the user is sure that FUNC has no side effects, it can be safely parallelized as shown. If FUNC does have side effects or this routine somehow gets compiled as non-reentrant (-nore), it could yield wrong answers.

  IVAR = I is optional in the example above.



## LOOP PARALLEL - *cont*

- Example:

```
      PROGRAM MAIN
      REAL SUM(8)
C$DIR LOOP_PARALLEL
      DO I = 1, 8
        CALL SUMIT(I,SUM(I))
      ENDDO
      END

      SUBROUTINE SUMIT(ITASK,MYSUM)
      REAL A(10*1024*1024)
      REAL B(10*1024*1024)
      REAL C(10*1024*1024),MYSUM
      INTEGER ITASK,J

      DO J = 1, 10*1024*1024
        A(J) = ITASK
        B(J) = ITASK
        C(J) = ITASK
      ENDDO

      MYSUM = 0.0
      DO J= 1, 10*1024*1024
        MYSUM = MYSUM + A(J)
      ENDDO

      RETURN
      END
```

## LOOP_PARALLEL - *cont*

Manual parallel

In this example, 8 copies of SUMIT run in parallel. Each computes a separate sum, SUM(I). Since A, B and C are local arrays which are allocated on the stack, each invocation of SUMIT has its own copy of these allocated on its own thread private stack. Thread 0's stack is allocated by SPP-UX (dflssiz/maxssiz). Thread 1-7's stack is allocated by CPS startup code which by default creates 8 Mbyte stacks, unless CPS_STACK_SIZE environment variable is set. Due to the large arrays, this will not be enough. Each thread will need 120 Mbytes for its thread private stack, so CPS_STACK_SIZE must be set by the user.



## LOOP_PARALLEL - *cont*

- Example:

```
C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(I,J,TEMPX)
      DO K=1,NC
        DO I=1,NS1
          TEMPX = STEP5(I,K)
          DO J=1,N
            TEMPX = TEMPX + A(K,I,J)
          ENDDO
          TEMP(I,K) = TEMPX
        ENDDO
        CALL DO_WORK(K)
      ENDDO
```

LOOP_PARALLEL is used on the K loop. Since K's iteration space is divided up amongst the threads, induction variables I and J of the inner loops need to be made private, so each thread has its own copy of I and J which does not overwrite the values used by other threads for their I and J. Also, since each thread computes and uses its own TEMPX, it is made private. Last, each thread sets a unique location in array TEMP.



## LOOP_PARALLEL - *cont*

Manual parallel

- Example:

```
INTEGER*4 UPD_FREQ(10), FREQ(10)
C$DIR LOOP_PARALLEL
C$DIR LOOP_PRIVATE(J,UPD_FREQ)
      DO I=1,N
        DO J=1,10
          UPD_FREQ(J)=FREQ(J)+I
        ENDDO
        CALL DO_WORK(I,UPD_FREQ)
      ENDDO
```

LOOP_PARALLEL is used on the I loop to run subroutine DO_WORK in parallel. Since I's iteration space is divided up amongst the threads, induction variable J of the inner loop needs to be made private, so each thread has its own copy of J which does not overwrite the values used by other threads for their J. On each iteration of I, UPD_FREQ is set and its value is not dependent upon any other iteration of I. Since this is true and its size is known, it can be made private, so that threads are working on their own copy of UPD_FREQ which does not overwrite the values used by the other threads for their UPD_FREQ.



## LOOP PARALLEL - *cont*

- Compiler will not honor a LOOP_PARALLEL directive on a loop with "extended range"
  - Example:

```
      DO 10 I = 1,N
        IF (A(I) .GT. 0.) GO TO 30
 20     CONTINUE
          . . .
 10   CONTINUE
      . . .
 30   CONTINUE
        A(I) = B(I) * C(I)
      GO TO 20
```

## LOOP PARALLEL - *cont*

- Will not parallelize any loop that does not have an iteration count that is determinable prior to loop invocation at runtime
  - Example:

```
C$DIR LOOP_PARALLEL
      DO WHILE (A(I) .GT. 0)
        . . . . .
        A(I) = . . .
      ENDDO
```

  Since A(I) is assigned within the loop, the compiler cannot determine the loop's iteration count prior to loop invocation and cannot parallelize it.
- Will not parallelize any loops that have exits other than normal termination





## Using *attribute-list* with parallel directives

- THREADS
  - Default behavior
  - Called "thread-way" parallelism
  - If encountered by thread 0 alone, it causes all threads available to the application to participate
  - If encountered within thread-way parallel construct it is ignored
  - If encountered within node-way parallel construct it causes the threads within this hypernode to split up the iteration space of the loop
- NODES
  - Called "node-way" parallelism
  - One thread per hypernode participates in loop
  - Only effective if encountered by thread 0 executing alone



## Using *attribute-list* with parallel directives - *cont*

- Example:

```
      REAL A(128,128),B(128,128)
      REAL C(128,128)
C$DIR LOOP_PARALLEL(NODES)
C$DIR LOOP_PRIVATE(I)
      DO J = 1,128
C$DIR LOOP_PARALLEL(THREADS)
C$DIR LOOP_PRIVATE(K)
        DO I = 1,128
          C(I,J) = 0.
          DO K = 1,128
            C(I,J)=C(I,J)+A(I,K)*B(K,J)
          ENDDO
        ENDDO
      ENDDO
```





## Using *attribute-list* with parallel directives - *cont*

Assume that at execution time this process runs on a subcomplex of 4 hypernodes each with 8 processors. The J loop begins executing on one thread on each of the 4 hypernodes. Each of these 4 threads encounters the I loop, causing 8 threads on each hypernode to inherit their nodes partitioning of the J loop, and then to partition the iterations of the I loop among themselves. This results in a 32-way partitioning of the address space such that each thread executes 32 iterations of the J loop and 16 iterations of the I loop.

K needs to be made private to the I loop, so that the threads in the inner loop have their own copy of K and are not overwriting the value in use by other threads.

I needs to be made private to the J loop, since there needs to be a copy of I for each of the J hypernodes. Otherwise, the I variable would be shared by the outer loop hypernode threads who would overwrite each other's values for I.

## Using *attribute-list* with parallel directives
- *cont*

Manual parallel

- CHUNK_SIZE = *n*
  - *n* must be an integer constant or constant expression evaluated at compile time
  - Chunks of n (or fewer) iterations are executed cyclically by the threads until all loop iterations are executed
  - No restriction on order in which iterations start or finish
  - Primarily for load balancing
  - Example:

```
C$DIR PREFER_PARALLEL(CHUNK_SIZE = 4)
      DO I = 1,100
        A(I) = B(I) + C(I)
      ENDDO
```

In this example, the loop is parallelized by parcelling out chunks of 4 iterations to each available thread. See Figure 10-1 for a pictorial description using Fortran 90 array syntax to illustrate the iterations performed by each thread, assuming 8 threads.



Figure 11-1: Loop iterations for CHUNK_SIZE example

| | |
|---|---|
| A(1:4)=B(1:4)+C(1:4)<br>...<br>A(97:100)=B(97:100)+C(97:100) | A(5:8)=B(5:8)+C(5:8)<br>...<br>A(69:72)=B(69:72)+C(69:72) |
| THREAD 0 | THREAD 1 |
| A(9:12)=B(9:12)+C(9:12)<br>...<br>A(73:76)=B(73:76)+C(73:76) | A(13:16)=B(13:16)+C(13:16)<br>...<br>A(77:80)=B(77:80)+C(77:80) |
| THREAD 2 | THREAD 3 |
| A(17:20)=B(17:20)+C(17:20)<br>...<br>A(81:84)=B(81:84)+C(81:84) | A(21:24)=B(21:24)+C(21:24)<br>...<br>A(85:88)=B(85:88)+C(85:88) |
| THREAD 4 | THREAD 5 |
| A(25:28)=B(25:28)+C(25:28)<br>...<br>A(89:92)=B(89:92)+C(89:92) | A(29:32)=B(29:32)+C(29:32)<br>...<br>A(93:96)=B(93:96)+C(93:96) |
| THREAD 6 | THREAD 7 |

This figure show 100 iterations parcelled out in chunks of 4 iterations to 8 threads. There is one chunk left over after the even distribution, iterations 97-100, which is executed by thread 0.



## Using *attribute-list* with parallel directives
- *cont*

Manual parallel

- CHUNK_SIZE = *n* *(contd.)*
  - Most useful on loops in which the amount of work increases or decreases as a function of the iteration count
    - Example:

```
C$DIR PREFER_PARALLEL
     *             (CHUNK_SIZE=4)
      DO J = 1,N
        DO I = J,N
          A(I,J) = . . .
        ENDDO
      ENDDO
```

In this example, the work of the I loop decreases as J increases. By specifying a CHUNK_SIZE for J, the load across the threads executing the loop is more evenly balanced.



## Using *attribute-list* with parallel directives
- *cont*

- MAX_THREADS = *m*
  - Useful when the maximum number of threads the loop will run on efficiently is known
  - Primarily for efficiency
- ORDERED
  - Causes the iterations of the loop to be *initiated* in loop order across the processors
  - Compiler does not generate any synchronization code so programmer must do this explicitly
  - No control over the order in which iterations complete unless imposed by the programmer through synchronizations
  - Most useful when the loop contains dependencies needing to be synchronized
  - Could also be used to balance work among threads when the workload varies widely among the iterations
  - Only useful with LOOP_PARALLEL not with PREFER_PARALLEL

## Using *attribute-list* with parallel directives - *cont*

Manual parallel

- Example:

```
C$DIR LOOP_PARALLEL(ORDERED)
      DO I = 1,N
        . . .
      ENDDO
```

Given Nthreads with thread ids 0, 1, . . . , Nthreads-1, then each thread would execute as follows:

```
      DO I=MY_THREAD_ID+1,N,nthreads
        . . .
      ENDDO
```

This example presumes there is some ordered critical section within the loop which the programmer synchronizes explicitly.



## BEGIN_TASKS,NEXT_TASK,END_TASKS

- Fortran syntax
  - C$DIR BEGIN_TASKS [(*attribute-list*)]
  - C$DIR NEXT_TASK
  - C$DIR END_TASKS
- C syntax
  - #pragma _CNX begin_tasks [(*attribute-list*)]
  - #pragma _CNX next_task
  - #pragma _CNX end_tasks
- BEGIN_TASKS tells compiler to start parallelizing a series of tasks; NEXT_TASK marks end of one task and start of another; END_TASKS marks end of series of tasks to be parallelized
- Any number of NEXT_TASKS can be specified
- Default behavior without an [(*attribute-list)*] - compiler performs thread-way (THREADS) parallelism, i.e., tasks will run thread-parallel, i.e., each task is assigned a separate thread until all tasks are complete
- Starting order of the tasks is indeterminate
- Useful for parallelizing code outside of a loop



## BEGIN_TASKS,NEXT_TASK,END_TASKS - *cont*

Manual parallel

- Compiler will not do any dependency checking or synchronization. Programmer must verify:
  - Two tasks do not assign values to the same data item (without synchronization)
  - Any data item assigned by one task must not be referenced by another task (unless synchronization directives used - see later example)
- If number of tasks > number of available threads (or hypernodes), each thread (or hypernode) will execute multiple tasks
- If number of threads (or hypernodes) > number of tasks, some threads (or hypernodes) will not execute any tasks



## BEGIN_TASKS,NEXT_TASK,END_TASKS - *cont*

- Simple example:

```
C$DIR BEGIN_TASKS
      DO I = 1,N-1
        A(I) = A(I+1) + B(I)
      ENDDO
C$DIR NEXT_TASK
      CALL TSUB(X,Y)
C$DIR NEXT_TASK
      C(1:1000:2) = D(1:500)
C$DIR END_TASKS
```

In this example, one thread executes the DO I loop, another threads executes the CALL TSUB(X,Y) and a third assigns the elements of the array D to every other element of C. These threads execute in parallel but their starting and ending orders are indeterminate.

## BEGIN_TASKS,NEXT_TASK,END_TASKS - *cont*

Manual parallel

Since the NODES attribute is not used in this example, the tasks are thread-way parallelized. This means there is no room for nested parallelism within the individual parallel tasks. So the forward LCD on the DO I loop is inconsequential because the loop will only run scalar. The Fortran 90 array assignment in the last task will also not parallelize even though it's technically parallelizable.

- Nested parallelism is possible
  - Thread parallelism must be nested within hypernode parallelism:
    - BEGIN_TASKS(NODES) must enclose BEGIN_TASKS(THREADS)
    - BEGIN_TASKS(NODES) must enclose LOOP_PARALLEL(THREADS) , if a node-parallel task is to contain a parallel loop, since automatic parallelization is disabled
    - Thread-way parallelism nested within node-way parallelism can only run on the threads of the hypernode where it is contained



## BEGIN_TASKS,NEXT_TASK,END_TASKS - *cont*

- Example:

```
C$DIR BEGIN_TASKS(NODES)
C$DIR     LOOP_PARALLEL(THREADS)
          DO I = 1,NBIG
            A(I) = B(I) * C(I)
            D(I) = E(I) * F(I)
            .........
          ENDDO
C$DIR NEXT_TASK
C$DIR     BEGIN_TASKS(THREADS)
          CALL T1SUB()
C$DIR     NEXT_TASK
          CALL T2SUB()
C$DIR     END_TASKS       !(THREADS)
C$DIR END_TASKS           !(NODES)
```



## BEGIN_TASKS,NEXT_TASK,END_TASKS - *cont*

Manual parallel

In this example, the first node-parallel task contains a thread-way parallel loop that goes parallel on the threads of the hypernode on which the task is running. The second node-parallel task contains subroutine calls, each of which runs thread-parallel within the hypernode (each task executes on one processor).



## TASK_PRIVATE

- Fortran

  **C$DIR TASK_PRIVATE(***namelist***)**

- C

  **#pragma _CNX task_private(***namelist***)**

- Declares variables and arrays in *namelist* private to immediately following tasks
- Similar to LOOP_PRIVATE except used for tasks
- Must immediately precede or appear on the same line as its corresponding BEGIN_TASKS directive
- Compiler assumes that TASK_PRIVATE data objects have no dependencies in the tasks where they are used
  - Can use synchronization directives and techniques to handle dependencies
- Cannot assume any starting or ending values for the object
- If a TASK_PRIVATE data object is referenced within a task, it must have been assigned previously in that task

## TASK_PRIVATE - *cont*

- Example:

```
        REAL*8 A(100),B(100),WRK(100)
  C$DIR TASK_PRIVATE(WRK)
  C$DIR BEGIN_TASKS
        DO I = 1,N
          WRK(I) = A(I)
        ENDDO
        DO I = 1,N
          A(I) = WRK(N-I)
        ENDDO
  C$DIR NEXT_TASK
        DO J = 1,M
          WRK(J) = B(J)
        ENDDO
        DO J = 1,M
          B(J) = WRK(M-J)
        ENDDO
  C$DIR END_TASKS
```





## TASK_PRIVATE - *cont*

In the first task in this example, WRK is used to hold array A so that its order can be reversed. Notice that it is assigned before it is used in the second loop. The second task uses it in a similar way. The two tasks each have their own private copy of WRK and they do not have access to interfere with the WRK array of the other task.



## Using *attribute-list* with task directives

- THREADS
  - Default behavior
  - All threads within this process are activated to partition the execution of the tasks
  - If nested within BEGIN_TASKS (NODES), then the process threads within the hypernode partition the execution of the tasks
- NODES
  - Tasks execute on individual hypernodes, one thread per hypernode
- MAX_THREADS=*m*
  - No more than *m* threads may be used to execute the tasks
  - If specified along with NODES attribute then it restricts execution to no more than *m* hypernodes
- ORDERED
  - Causes tasks to be *initiated* in their lexical order
  - Does not provide any synchronization or guarantee any ending order





## Using *attribute-list* with task directives - *cont*

- Can impose partial ordering (including ending order) among tasks by synchronizations
  - Example:

```
        COMMON /FO/ A(999),B(999),
       *  C(999),D(999),P(999),
       *  Q(999),R(999),X(999),
       *  Y(999),Z(999)
  C$DIR GATE(EVENT1,EVENT2)
        . . .
        LOK1 = ALLOC_GATE(EVENT1)
        LOK2 = ALLOC_GATE(EVENT2)
        . . .
        LOK1 = LOCK_GATE(EVENT1)
        LOK2 = LOCK_GATE(EVENT2)
  C$DIR BEGIN_TASKS(ORDERED)
        DO I = 1,N
         A(I) = 2.0 * A(I) + C(I)
        ENDDO
        LOK1 = UNLOCK_GATE(EVENT1)
  C$DIR NEXT_TASK
        DO J = 1,N
         B(J) = C(J) * SIN(X(J))
        ENDDO
        LOK2 = UNLOCK_GATE(EVENT2)
```

## Using *attribute-list* with task directives - *cont*

```
C$DIR NEXT_TASK
      DO K = 1,N
       P(K) = EXP(3.0*SQRT(Q(K)))
     *    /ATAN(R(K))
      ENDDO
      LOK1 = LOCK_GATE(EVENT1)
      LOK1 = UNLOCK_GATE(EVENT1)
      LOK2 = LOCK_GATE(EVENT2)
      LOK2 = UNLOCK_GATE(EVENT2)
      DO K = 1,N
       D(K) = P(K) * B(K) / A(K)
      ENDDO
C$DIR NEXT_TASK
      DO L = 1,N
       Y(L) = X(L) * Y(L) / Z(L)
      ENDDO
C$DIR END_TASKS
```



## Using *attribute-list* with task directives - *cont*

In this example, the second loop in the third task relies on data calculated in the other previous tasks. So there are locks set at the beginning representing the first and second tasks. These tasks unlock these gates when they are through calculating the needed arrays for the third task. The third task tries to lock the gates and will wait on the lock until the other tasks have unlocked the gate. It then unlocks the gates because the only real reason for locking them was to wait for the other tasks to complete. In this way the third task will not start its loop that depends on other data until that data has been calculated.





## Synchronization Directives:

Aid in synchronization among threads executing:

- manually parallelized loop, or

- parallel tasks created with the BEGIN_TASKS / NEXT_TASK / END TASKS directives.

Both synchronization directives discussed below should only be used if the amount of parallel work is significantly larger than the amount of work that needs to be synchronized. They add significant synchronization overhead to a program.

- **Critical Section Directives**

  This directive pair is used to create a block of code that can be executed by only one thread at a time (in any order):

  **Fortran:**

  ```
  C$DIR CRITICAL_SECTION [(gate)]
  C$DIR END_CRITICAL_SECTION
  ```

  **C:**

  ```
  #pragma _CNX critical_section [(gate)]
  #pragma _CNX end_critical_section
  ```



## Synchronization Directives - *cont*:

- **Critical Section Directives** ***(contd.)***
  - Must appear within a single procedure.
  - The directives do not have to be in the same procedure as the parallel construct in which they are used: they can be in a procedure that is called from within a LOOP_PARALLEL loop or other parallel construct.
  - Act as a structured LOCK_GATE / UNLOCK_GATE pair.
  - **'gate' parameter:**

    - Optional parameter used to specify a gate variable to be used for access to the critical section.

    - If *no* gate is specified, the compiler creates a *unique* internal gate variable for the critical section.

    - Required when synchronizing access to a shared variable from multiple parallel tasks.

    - Must be declared, allocated, and then initialized to UNLOCKED prior to use via the ALLOC_GATE function. Should be deallocated when finished using the FREE_GATE function.

    - Initialization must be performed outside of any parallel construct in which gates are used.

## Synchronization Directives - *cont*:

- **Critical Section Directives** *(contd.)*

**Example 1.**

**(using critical section directives):**

```
C$DIR LOOP_PARALLEL
        DO I = 1,N
          ...
C$DIR CRITICAL_SECTION
          SUM = SUM + X(I)*Y(I)
C$DIR END_CRITICAL_SECTION
           ...
        ENDDO
```

Each thread will wait for exclusive access at the **CRITICAL_SECTION** directive and relinquish exclusive access at the **END_CRITICAL_SECTION** directive.

The results in **SUM** will be updated only by one thread at a time.



## Synchronization Directives - *cont*:

- **Critical Section Directives** *(contd.)*

**Example 2.**

**This is another way of coding the Example 1, using manual synchronization (synchronization functions) rather than critical section directives:**

```
C$DIR GATE(CRITSEC)
...
        ALLOC_GATE(CRITSEC)
C$DIR LOOP_PARALLEL
        DO I = 1,N
          ...
        LOCK = LOCK_GATE(CRITSEC)
          SUM = SUM + X(I)*Y(I)
        LOCK = UNLOCK_GATE(CRITSEC)
           ...
        ENDDO
        FREE_GATE(CRITSEC)
```



## Synchronization Directives - *cont*:

- **Critical Section Directives** *(contd.)*

**Example 3.**

```
C$DIR GATE(SUM_GATE)
       ...
      LOK1 = ALLOC_GATE (SUM_GATE)

C$DIR BEGIN_TASKS
      SUM1 = 0.0
      DO J =1,M
        SUM1 = SUM1 + FUNC1(J)
      ENDDO

C$DIR CRITICAL_SECTION (SUM_GATE)
      GLOBAL_SUM =GLOBAL_SUM + SUM1
C$DIR END_CRITICAL_SECTION (SUM_GATE)

C$DIR NEXT_TASK

      SUM2 = 0.0
      DO I = 1,N
        SUM2 = SUM2 + FUNC2(I)
      ENDDO

C$DIR CRITICAL_SECTION (SUM_GATE)
      GLOBAL_SUM =GLOBAL_SUM + SUM2
C$DIR END_CRITICAL_SECTION (SUM_GATE)

C$DIR END_TASK
```



## Synchronization Directives - *cont*:

- **Critical Section Directives** *(contd.)*

**Example 3.** *(contd.)*

This example illustrates two different tasks, each computing a contribution to a global sum.

The gate variable **SUM_GATE** must be present on both **CRITICAL SECTION** directives, in order to assure that only one thread at a time updates the variable **GLOBAL_SUM**.

When one task reaches the critical section, the **CRITICAL_SECTION** directive automatically locks **SUM_GATE**. The **END_CRITICAL_SECTION** directive unlocks **SUM_GATE** on exit from the section. Since access to both critical sections is controlled by a single gate, the sections will execute one thread at a time.

## Synchronization Directives - *cont*:



- **Ordered Critical Section Directives**

  Necessary when a "critical section" must be executed in the order of the iterations of the parallel loop.

  Usually because of an embedded recurrence in otherwise parallelizable code.

  **Fortran**:

```
C$DIR ORDERED_SECTION (gate)
C$DIR END_ORDERED_SECTION
```

  **C**

```
#pragma _CNX ordered_section (gate)
#pragma _CNX end_ordered_section
```

  Directives must appear within the same flow of control

  Directives need not be in the same program unit as the parallel construct that contains them.

  Directives may only be used within a **LOOP_PARALLEL (ORDERED)** loop.



## Synchronization Directives - *cont*:

- **Ordered Critical Section Directives** *(contd.)*

  **Requirements for the code contained within an ordered section:**

  Need to be executed one thread at a time.

  Threads are granted access to the section, in the iteration order specified by the containing **LOOP_PARALLEL (ORDERED)** loop.

  Must not jump out of the ordered section - exit only through **END_ORDERED_SECTION**.

  **gate** variable *required*.

  **gate** variable must be "unlocked" prior to invocation of the **LOOP_PARALLEL (ORDERED**) loop.



## Synchronization Directives - *cont*:



- **Ordered Critical Section Directives** *(contd.)*

  Example:

```
DO I = IS, IE, INC
  ... (parallelizable code)
  A(I) = A(I-INC) + X(I)*Y(I)
  ... (parallelizable code)
ENDDO
```

  If executed in parallel as written, there would be no guarantee that the recurrence in **A** would be properly computed. Manual synchronization within the "ordered section" is necessary to insure that **A(I-INC)** is computed before **A(I)** uses it:



## Synchronization Directives - *cont*:

- **Ordered Critical Section Directives** *(contd.)*

```
C$DIR GATE (SEQ)
      LOK1 = ALLOC_GATE (SEQ)
        ...
C$DIR LOOP_PARALLEL (ORDERED)
        DO I = 1,N
         ...
C$DIR ORDERED_SECTION (SEQ)
          A(I) = A(I-1) + X(I)*Y(I)
C$DIR END_ORDERED_SECTION
        ...
        ENDDO
      LOK1 = FREE_GATE (SEQ)
```

  The ORDERED_SECTION enforces the restriction that the code contained between it and the END_ORDERED_SECTION be executed in iteration order.

  It is a requirement that the ordered section directives be executed exactly once on *every* iteration *or* not executed on *any* iteration.

  If the recurrence is executed conditionally, this can be handled in the following manner:

## Synchronization Directives - *cont*:

- **Ordered Critical Section Directives** ***(contd.)***

  Example:

```
C$DIR GATE(SEQ)
      LOK1 = ALLOC_GATE(SEQ)
        ...
C$DIR LOOP_PARALLEL (ORDERED)
        DO I = 1,N
         ...
         IF (Z(I).GT.0.) THEN
C$DIR ORDERED_SECTION (SEQ)
           A(I) = A(I-1) + X(I)*Y(I)
C$DIR END_ORDERED_SECTION
         ELSE
C$DIR ORDERED_SECTION (SEQ)
C$DIR END_ORDERED_SECTION
         ENDIF
         ...
        ENDDO
```

The iterations that do not compute the recurrence still execute the empty ordered section in the ELSE block.





## Nested directives example

- Example using TASK, LOOP_PARALLEL and CRITICAL_SECTION directives

```
C$DIR GATE(LOCK1,LOCK2)
      LOK1 = ALLOC_GATE(LOCK1)
      LOK2 = ALLOC_GATE(LOCK2)
C$DIR BEGIN_TASKS(NODES)
C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1,N
       CALL SUM_SUB(A,M1,SUM1,LOCK1)
      ENDDO
C$DIR NEXT_TASK
C$DIR LOOP_PARALLEL(THREADS)
      DO J = 1,L
       CALL SUM_SUB(B,M2,SUM2,LOCK2)
      ENDDO
C$DIR END_TASKS
      LOK1 = FREE_GATE(LOCK1)
```



## Nested directives example - *cont*

```
      LOK2 = FREE_GATE(LOCK2)
       . . .
      SUBROUTINE SUM_SUB(X,N,SUM,LOK)
      REAL*8 X(N),SUM
C$DIR GATE(LOK)
       . . .
C$DIR CRITICAL_SECTION(LOK)
      SUM = SUM + . . .
C$DIR END_CRITICAL_SECTION
```

In this example, two different sums are being computed by different sets of threads. The LOK argument provides separate protection for each of the sums even if SUM_SUB is invoked simultaneously from each of the task blocks.





## Exercises

**[1]** What is stopping automatic parallelization of the following loop? How could it be restructured to cause it to parallelize?

```
(a) DO J = 1,N
      DO I = J,N
        A(I,J) = B(I,J) + C(I,J)
        CALL SUBVERT(N,I,J,A,B)
      ENDDO
    ENNDO
 ...
    SUBROUTINE SUBVERT(N,I,J,A,B)
    REAL*8 A(N,*),B(N,*)
    IF (A(I,J) .GT. 1.0) A(I,J) = B(I,J)
 ...
    END
```

# Memory classes

# 13

Topics:

- Parallel information functions
- Default memory allocation
- Using memory classes
- Summary of memory class
- Fortran memory class directives
- C storage class extensions
- Memory class assignments
- Thread Private class
- Node Private class
- Near Shared class
- Far Shared class
- Block Shared class





## Parallel information functions

- Provide information regarding the parallelism or potential parallelism of your program
  - information returned is from the application view of the subcomplex on which your program is executing, not a system configuration view
  - dependent upon whether your program is marked parallel or not
- Integer functions available in 4 and 8 byte versions
- Can appear in executable statements anywhere an integer expression is legal
- Available in Fortran and C
- C programs must include *spp_prog_model.h* for prototypes
- Useful when using memory classes





## Parallel information functions - *cont*

## Number of threads

- Fortran

```
INTEGER FUNCTION NUM_THREADS()
INTEGER*8 FUNCTION NUM_THREADS_8()
```

- C

```
int num_threads(void);
long long num_threads_8(void);
```

- Returns the total number of threads the process creates at initiation, regardless of how many are idle or active
- Typically used to manually define thread-parallel loops which may span hypernodes

## Parallel information functions - *cont*

### Number of processors

- Fortran

```
INTEGER FUNCTION NUM_PROCS()
INTEGER*8 FUNCTION NUM_PROCS_8()
```

- C

```
int num_procs(void);
long long num_procs_8(void);
```

- Returns the total number of processors the process is using at runtime
- Can be used to dimension Fortran automatic and adjustable arrays
- Can be used in C or Fortran to dynamically specify array dimensions and to allocate storage
- Only different than num_threads() when **+over** is used





## Parallel information functions - *cont*

### Number of hypernodes

- Fortran

```
INTEGER FUNCTION NUM_NODES()
INTEGER*8 FUNCTION NUM_NODES_8()
```

- C

```
int num_nodes(void);
long long num_nodes_8(void);
```

- Returns the number of hypernodes on which the process is running
- Can be used to dimension Fortran automatic and adjustable arrays
- Can be used in C or Fortran to dynamically specify array dimensions and to allocate storage



## Parallel information functions - *cont*

### Number of threads on current hypernode

- Fortran

```
INTEGER FUNCTION NUM_NODE_THREADS()
INTEGER*8 FUNCTION NUM_NODE_THREADS_8()
```

- C

```
int num_node_threads(void);
long long num_node_threads_8(void);
```

- Returns the number of threads running on the hypernode from which the function is called





## Parallel information functions - *cont*

### Thread id

- Fortran

```
INTEGER FUNCTION MY_THREAD()
INTEGER*8 FUNCTION MY_THREAD_8()
```

- C

```
int my_thread(void);
long long my_thread_8(void);
```

- Returns the spawned thread id of the calling thread, in the range of 0...*nst*-1 where *nst* is the number of threads in the current spawn context.
- Used to direct specific tasks to specific threads inside parallel constructs
- Only valid to use in a parallel program

## Parallel information functions - *cont*

## Hypernode id



- Fortran

```
INTEGER FUNCTION MY_NODE()
INTEGER*8 FUNCTION MY_NODE_8()
```

- C

```
int my_node(void);
long long my_node_8(void);
```

- Returns the logical hypernode id of the hypernode on which the calling thread is running, in the range of 0...*num_nodes()*-1 where hypernode 0 is where thread 0 is running
- Use to direct specific tasks to specific hypernodes inside parallel constructs



## Parallel information functions - *cont*

## Level of parallelism

- Fortran

```
INTEGER FUNCTION LEVEL_OF_PARALLELISM()
INTEGER*8 FUNCTION
                LEVEL_OF_PARALLELISM_8()
```

- C

```
int level_of_parallelism(void);
long long level_of_parallelism_8(void);
```

- Returns a value representing the level of parallelism of the calling process
- Returns one or a sum of the following values:

| Function return value | C symbolic constant name | Meaning |
|---|---|---|
| 0 | CPS_PL_NONE | Not parallel |
| 1 | CPS_PL_PARALLEL | Asymmetric thread active |
| 2 | CPS_PL_NODE | Node-parallelism |
| 4 | CPS_PL_NTHREAD | Thread-parallelism within a hypernode |
| 8 | CPS_PL_THREAD | Single-dimensional thread-parallelism |
| 16 | CPS_PL_ASYMMETRIC | Current thread is asymmetric or child of asymmetric thread |

For C, values defined in *spp_prog_model.h*



## Default memory allocation



Before beginning an explanation of memory classes and what happens when variables are assigned alternate memory classes, let's discuss the default behavior.

- All local and global data objects default to shared memory
- Global data objects
  - Allocated in static storage
  - Have save semantics
  - Are zero initialized
- Local variables that have initial values specified or appear on SAVE statements
  - Allocated in static storage
  - Have save semantics
  - Are zero initialized when on SAVE statement and no initialization by the user
- All other local variables
  - Allocated off the stack
  - Do not have save semantics



## Using memory classes

- Helpful when you want to enhance data locality in memory, or access more than 4 GBytes of address space.
- Local variables defined in memory classes are:
  - Allocated in static storage
  - Have save semantics
  - Are zero initialized
- Dynamic variables allocated to memory classes are:
  - Allocated from the heap
  - Do not have save semantics
- Declarations must appear with other specification statements and not within executable statements
- In some cases, loops which manipulate variables which have been put into specific memory classes by the methods described in this chapter may have to be manually parallelized.

## Summary of memory classes

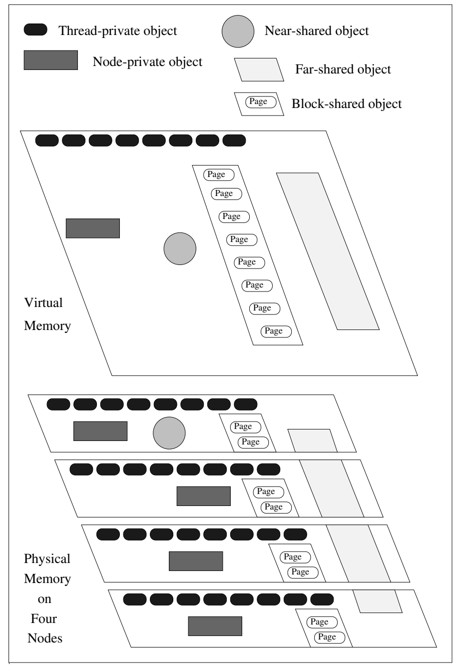


- Thread-private
  - Accessible only to a single thread
- Node-private
  - Accessible only to threads residing on the same hypernode
- Near-shared
  - Accessible to all threads but “closer” to threads on the hypernode on which they are allocated
- Far-shared
  - Accessible to all threads and relatively equidistant from all threads
- Block-shared
  - Accessible to all threads but uniformly distributed in equal-sized contiguous chunks onto each hypernode of executing process

See the following diagram for memory class layouts.



Memory class layout:






## Fortran memory class directives



Memory classes are assigned to data items using the following compiler directives which appear with other specification statements:

- **C$DIR THREAD_PRIVATE** (*namelist*)
- **C$DIR NODE_PRIVATE** (*namelist*)
- **C$DIR NEAR_SHARED**(*namelist*)
- **C$DIR FAR_SHARED**(*namelist*)
- **C$DIR BLOCK_SHARED** (*allocatable-array-list*)

where *namelist* is a comma-delimited list of COMMON block names, array names and scalar variables names

NOTE: COMMON block names must be enclosed in slashes. Individual arrays and variables in *namelist* must not also appear in COMMON block declarations or be equivalenced to any objects in a COMMON block. That is, either all the COMMON block must be declared or none of it.

- Can dynamically allocate or statically define any of the above classes, except for BLOCK_SHARED which can only be dynamically allocated
- Dynamic memory class assignments made using Fortran 90 ALLOCATABLE arrays



## Fortran memory class directives - *cont*

- Example - static

```
      REAL*8 FIRST_CLASS(16)
      REAL*8 PROPERTY(80),X,Y,Z
      INTEGER EXPERIENCES(256)
      COMMON /ANCESTORS/ X,Y,Z
C$DIR THREAD_PRIVATE(FIRST_CLASS)
C$DIR NODE_PRIVATE(PROPERTY)
C$DIR NEAR_SHARED(EXPERIENCES)
C$DIR FAR_SHARED(/ANCESTORS/)
```

- Example - dynamic

```
      REAL*8 BOXES
      ALLOCATABLE BOXES(:,:)
      ALLOCATABLE A(:,:)
C$DIR BLOCK_SHARED(BOXES)
C$DIR NEAR_SHARED(A)
  ...
      ALLOCATE(A(N,M))
      ALLOCATE(BOXES(I,J))
```

## C storage class extensions

- Memory classes are assigned in variable declarations using the following type-qualifier extensions:
  - thread_private
  - node_private
  - near_shared
  - far_shared
- Must include */usr/convex/all/include/spp_prog_model.h*
- No block_shared type since it can only be dynamically allocated
- Can dynamically allocate or statically define any of the above classes
- Data assigned a memory class within a function must have the storage class **static** or **extern** because local variables are allocated on the stack and cannot be assigned a memory class. This is not true for globals.
- Examples - static

```
#include <spp_prog_model.h>

/* allocate thread private object */
static thread_private int num1;

/* allocate far shared object */
static far_shared float a[100][100];
```





## C storage class extensions - *cont*

- Dynamic memory class assignments made using a special form of malloc:

  **#include <spp_prog_model.h>**

  ***mptr* =**
  **memory_class_malloc(*Nbytes,class_name*)**

  where:

  *mptr* is a previously declared pointer to a variable of the desired memory class

  *Nbytes* is the requested number of bytes

  *class_name* is one of the predefined constants defined in *spp_prog_model.h*:

  ```
  THREAD_PRIVATE_MEM
  NODE_PRIVATE_MEM
  NEAR_SHARED_MEM
  FAR_SHARED_MEM
  BLOCK_SHARED_MEM
  ```

- Allocates a single object of *Nbytes* in the requested memory class and returns a pointer to it.



## Memory class assignments

- Static assignments make sense for:
  - thread_private
  - node_private
  - far_shared
- Dynamic assignments make sense for:
  - near_shared
  - block_shared
- Once a data item has been assigned a memory class, the class cannot be changed





## Thread Private class

- Accessible to single thread only
- Implemented by the compiler
- Synchronization must be done through message passing or synchronized copying to shared variables
- Each thread has a copy at a unique virtual address
- Unique virtual address replicated on each hypernode in multi-hypernode subcomplex
- Virtual addresses map to unique physical addresses on each hypernode
  - NOTE: Each virtual replication of thread-private object subtracts from virtual address space available to any other thread. Physical address replication can cause a single data item to occupy a large amount of physical memory
- Static declarations make the most sense since variables are replicated for each thread on each hypernode
- TASK COMMON statement will result in thread private COMMON blocks
- Cannot be initialized in Fortran DATA statements

## Thread Private class - *cont*

- Example 1
  - Fortran

```
      REAL*8 TP(1000)
C$DIR THREAD_PRIVATE(TP)
```

  - C

```
static thread_private double tp[1000]
```

When loaded onto subcomplex with 4 hypernodes of 8 processors each, each of 32 threads will have a unique physical replication of TP. TP will be replicated 8 times in virtual memory occupying 8*8000 = 64000 bytes of virtual space. When loaded onto 4 hypernodes, each of the 8 replications maps to unique physical addresses on each hypernode occupying 4*64000 = 256000 bytes of physical space.

Memory classes



## Thread Private class - *cont*

- Example2 - Fortran thread_private COMMON blocks

```
      PROGRAM SIMPLE
      INTEGER I

C$DIR LOOP_PARALLEL(THREADS)
      DO I = 1, 8
        CALL INIT()
      ENDDO

      END

      SUBROUTINE INIT()
      INTEGER MY_THREAD, IAM
      COMMON /PROC/ IAM
C$DIR THREAD_PRIVATE (/PROC/)

      IAM = MY_THREAD()
      CALL SUB()

      RETURN
      END

      SUBROUTINE SUB()
      INTEGER IAM
      COMMON /PROC/ IAM
C$DIR THREAD_PRIVATE (/PROC/)

      PRINT *, `I am `, IAM
      RETURN
      END
```



## Thread Private class - *cont*

In this example, 8 copies of subroutine INIT run in parallel. Each copy of INIT has its own COMMON block called PROC containing the variable IAM. This is because PROC is defined to be THREAD_PRIVATE. INIT in turn calls subroutine SUB which also has PROC defined as THREAD_PRIVATE, so that it can reference the appropriate IAM for the copy of INIT in which it is called.

Output of the program on a given run:

I am 5
I am 7
I am 6
I am 0
I am 3
I am 4
I am 1
I am 2

Memory classes



## Node Private class

- Accessible only to threads on the same hypernode as where they are defined/allocated
- Same virtual address in all threads of process at runtime
- Maps to different physical addresses on each hypernode
- Requires synchronization of access to variables shared within threads on a hypernode
- Sharing of values with other hypernodes must be done through message passing or synchronized copying to shared variables
- **Only** class that effectively expands the physical address space by replicating virtual addresses
- Static declarations make the most sense since variables are replicated on each hypernode
- Provide guaranteed local memory access for any thread
- Can be initialized in Fortran DATA statements

## Node Private class - *cont*

- Examples
  - Fortran

```
      REAL*8 NP(1000)
C$DIR NODE_PRIVATE(NP)
```

  - C

```
static node_private double np[1000]
```

Assuming allocation on a subcomplex with 4 hypernodes, this declaration allocates 8000 bytes on each hypernode at the same virtual address = 8000 bytes virtual memory. Since the virtual address maps to a local physical address on each hypernode, the physical memory allocated is 4*8000 = 32000 bytes physical memory.





## Node Private class - *cont*

- Examples
  - Fortran

```
      REAL*8 NODE_V(10000)
C$DIR NODE_PRIVATE(NODE_V)
      . . .
C$DIR LOOP_PARALLEL(NODES)
C$DIR LOOP_PRIVATE(J)
      DO I = 1, NUM_NODES()
C$DIR LOOP_PARALLEL(THREADS)
        DO J = 1, 10000
          NODE_V(J)=PI*(RAD(J)**2)
          P(I,J)=T(I,J)/NODE_V(J)
        ENDDO
C$DIR LOOP_PARALLEL(THREADS)
        DO J = 2, 10000
          IF(comparison on NODE_V(J))
            LMAX(I,J) = NODE_V(J)
        ENDDO
        . . .
      ENDDO
```

Each hypernode has its own array NODE_V of 10,000 elements to use privately in the computation of its portion of shared array P, and to find the values of its portion of shared array LMAX.



## Node Private class - *cont*

  - C

```
static node_private double node_v[10000];
.
.
#pragma _CNX loop_parallel(nodes,ivar=i)
#pragma _CNX loop_private(j)
for (i = 0; i < num_nodes(); i++) {

#pragma _CNX loop_parallel(threads,ivar=j)
   for (j = 0; j < 10000; j++) {
      node_v[j] = pi*rad[j]*rad[j];
      p[i,j] = t[i,j] / node_v[j];
   }

#pragma _CNX loop_parallel(threads,ivar=j)
   for (j = 1; j < 10000; j++) {
      if (comparison on node_v[j])
        lmax[i,j] = node_v[j];
   }

}
```





## Near Shared class

- Accessible to all threads but “closer” to threads on hypernode where allocated
- Typically accessed heavily by threads of process executing on the hypernode containing the data and occasionally by threads of the same process on other hypernodes
- Has single virtual and physical address
- Can be initialized in Fortran DATA statements
- Static near shared objects
  - Maps to same physical address on logical hypernode 0 for all threads
  - Not replicated on other hypernodes
  - Has same virtual address in all threads of process at runtime
  - Limited usefulness

## Near Shared class - *cont*

- Examples
  - Fortran

```
      REAL*8 NS(1000)
C$DIR NEAR_SHARED(NS)
      DO I = 1, N
     ...NS(I)...
      ENDDO
```

  - C

```
static near_shared double ns[1000]
```

Array NS will occupy 8000 bytes of virtual memory and 8000 bytes of physical memory.

Since the array is shared, its address must be the same for all threads and it must exist in a fixed location in the program image. Since it is near-shared, it will be allocated completely on one hypernode which will always be hypernode 0. The example only makes sense if the loop will be run by threads on hypernode 0 or never run on multiple hypernodes.

Memory classes



## Near Shared class - *cont*

- Dynamic near shared objects
  - Maps onto the hypernode where requesting thread executes
  - Reasoning - near-shared data object is to be heavily accessed by creating thread with occasional access by other threads
  - Normally allocated from and most effective when used within explicitly defined hypernode parallel regions, i.e., LOOP_PARALLEL(NODES)



## Near Shared class - *cont*

- Examples
  - Fortran

```
      ALLOCATABLE A(:,:)
C$DIR NEAR_SHARED(A)
      ...
      ALLOCATE  (A(N,M))
```

  - C

```
static near_shared double *a;

a =(double*)memory_class_malloc
     (sizeof(double)*N,
                NEAR_SHARED_MEM);
```

Array A will occupy the same amount of both virtual and physical memory, amount to be determined by allocation. Virtual memory will be decreased by this amount when the variable is allocated but will not map to physical memory until the allocated objects are referenced.

Memory classes



## Near Shared class - *cont*

- Examples
  - Fortran

```
      REAL*8 XNS(:), YNS(:)
      ALLOCATABLE(XNS,YNS)
C$DIR NEAR_SHARED(XNS,YNS)
      INTEGER NODE_ID
      .
      .
C  code runs node parallel on
C  a 2 node subcomplex
      .
C$DIR LOOP_PARALLEL(NODES)
C$DIR LOOP_PRIVATE(NODE_ID)
      DO I = 1, NUM_NODES()
        NODE_ID = MY_NODE()
        IF (NODE_ID .EQ. 0) THEN
          ALLOCATE(XNS(1000))
        ELSE
          ALLOCATE(YNS(1000))
        ENDIF
        .
        .
        .
      ENDDO
```

## Near Shared class - *cont*

- C

```
static near_shared double *xns, *yns;
int node_id;
.
.
/* code runs node parallel on a 2 node */
/* subcomplex                          */
.
#pragma _CNX loop_parallel(nodes,ivar=i)
#pragma _CNX loop_private(node_id)

for (i = 0; i < num_nodes(); i++) {

node_id = my_node();
if (node_id == 0)
   xns =(double *)memory_class_malloc
     (sizeof(double)*1000,NEAR_SHARED_MEM);
else
   yns =(double *)memory_class_malloc
     (sizeof(double)*1000,NEAR_SHARED_MEM);
.
.
}
```

Memory classes



## Near Shared class - *cont*

This example assumes the program runs on a 2 hypernode subcomplex:

By a single thread on each hypernode, a near_shared array XNS of 1000 elements is allocated on hypernode 0, and near_shared array YNS of 1000 elements is allocated on hypernode 1.

XNS and YNS are both stored in hypernode-global physical memory on their respective hypernodes. So, although each is accessible from either hypernode, XNS is accessed from hypernode 0 with least latency and YNS from hypernode 1 with least latency.



## Far Shared class

- Accessible to all threads and relatively equidistant from all threads
- Typically used for data which is accessed equally by all the hypernodes in a subcomplex, therefore static declarations make the most sense
- Default class for all variables
- Provides highest overall bandwidth
- Provides best average access time when used equally by threads running on all hypernodes
- Same virtual and physical address in all threads of process at runtime
- Virtual pages are mapped by pages in round-robin manner to physical pages on all the hypernodes (page = 4K)
- Relatively uniform access to all threads when array elements spread across subcomplex
- Can be initialized in Fortran DATA statements

Memory classes



## Far Shared class - *cont*

- Examples
  - Fortran

```
      REAL*8 FS(1024)
C$DIR FAR_SHARED(FS)
```

  - C

```
static far_shared double fs[1024];
```

    Array FS occupies 8192 bytes virtual memory and 8192 bytes of physical memory.

## Block Shared class

- Provides for block distributed arrays
- Accessible to all threads, the data is uniformly distributed in contiguous equal-sized chunks across the hypernodes on which a process is executing(One chuck per hypernode starting on logical hypernode 0)
- Typically accessed by all threads but with most accesses by a particular thread being to data residing on its hypernode with occasional access to data on other hypernodes
- Ideal use is for array-manipulating loops which parallelize across hypernodes with each hypernode computing chunks of contiguous elements of the arrays
  - Chunks of the array are distributed evenly across hypernodes
  - Work can be distributed such that each hypernode accesses the elements that reside on it most frequently
  - If necessary, hypernodes can still directly access each other's block_shared data
- Can only be allocated dynamically and should take place outside of any parallel region
- Has unique virtual and physical address





## Block Shared class - *cont*

- Amount allocated must be integral number of page size (4096 bytes) and number of hypernodes (NUM_NODES()). If not, it is rounded up.
- In the following example, an easy way is shown to access hypernode-local elements of a block_shared array



## Block Shared class - *cont*

- Examples
  - Fortran

```
      REAL*8 A
      ALLOCATABLE A(:,:)
C$DIR BLOCK_SHARED(A)
   ...
      ALLOCATE(A(N,NUM_NODES())

C$DIR LOOP_PARALLEL(NODES)
C$DIR LOOP_PRIVATE(J)
      DO I = 1, NUM_NODES()
C$DIR LOOP_PARALLEL(THREADS)
        DO J = 1, N
           A(J,I) = . . . . .
           .
        ENDDO
      ENDDO
```

N must be an integral multiple of 4096/8 = 512.

Assume N = 1024 and **`NUM_NODES()`** returns 4, the requested size is 8*1024*4 = 32768 bytes. Each hypernode has mapped 8*1024/4096 = 2 contiguous pages.

A(1:1024,1) resides on hypernode 0, A(1:1024,2) on hypernode 1, A(1:1024,3) on hypernode 2 and A(1:1024,4) on hypernode 3.





## Block Shared class - *cont*

- C

```
static far_shared double *a;

a =(double*)memory_class_malloc
      (sizeof(double)*N,BLOCK_SHARED_MEM);

#pragma _CNX loop_parallel(nodes,ivar=i)
#pragma _CNX loop_private(j)

for (i = 0; i < num_nodes(); i++) {
#pragma _CNX loop_parallel(threads,ivar=j)
   for (j = 0; i < n; j++) {
      a[j,i] = . . . .
      .
      .
   }
}
```

Same restrictions on **`sizeof(double)*N`**

Figure 9-2: Far-Shared vs. Block-Shared Memory Layout

```
      ALLOCATABLE FS(:,:), BS(:,:)
      REAL*8 FS, BS
C$DIR FAR_SHARED (FS)
C$DIR BLOCK_SHARED (BS)
      ALLOCATE (FS(1024,4), BS(1024,4))
```