Introduction OpenMP Parallelization

Christian D. Ott

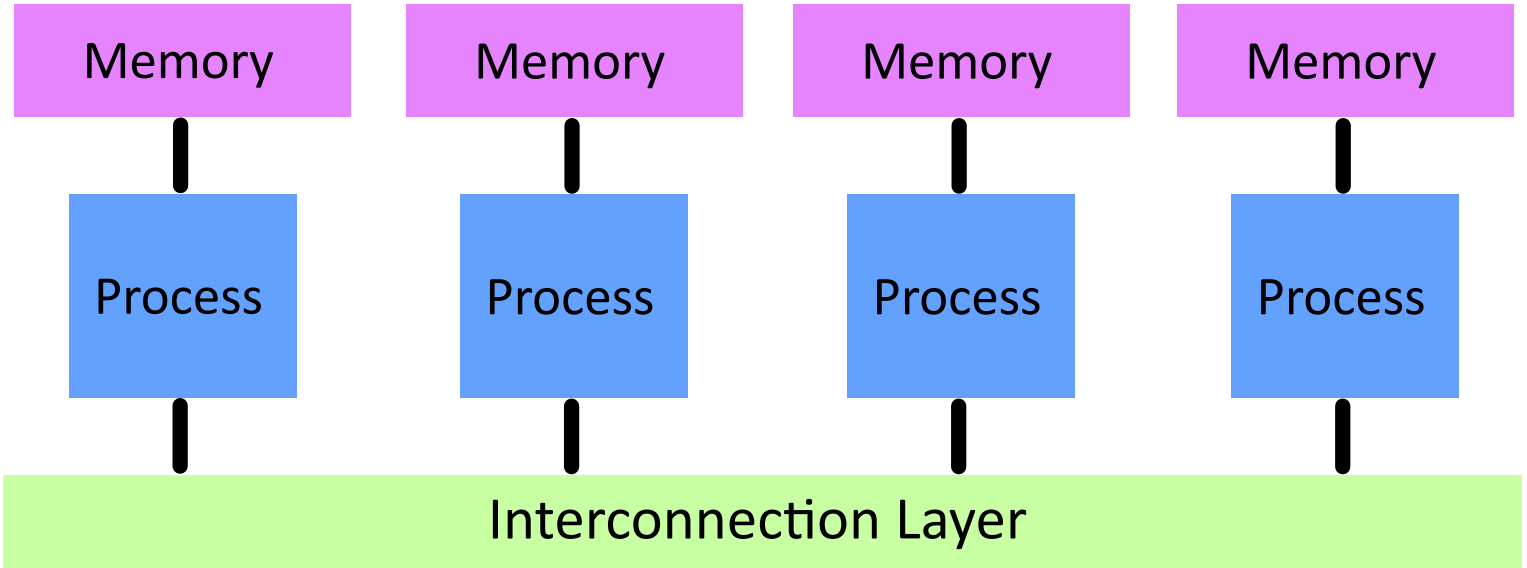
TAPIR, California Institute of Technology

June 2, 2011

Overview

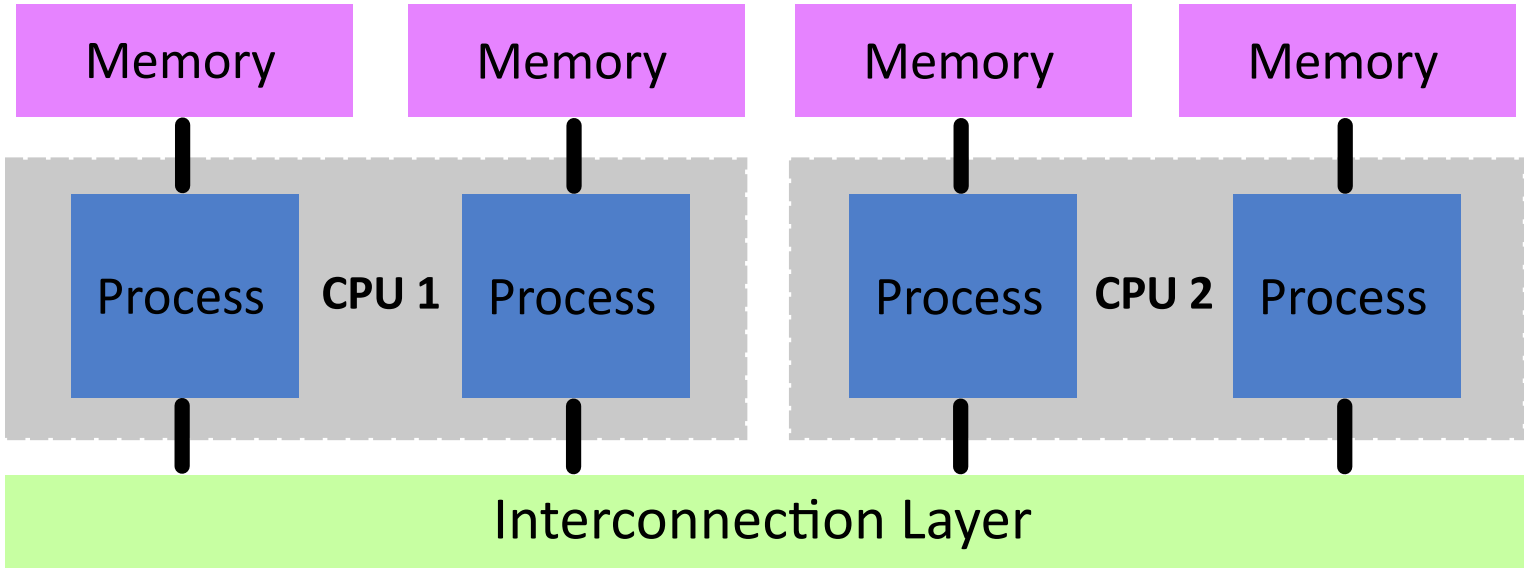
- Shared-Memory vs. Distributed-Memory Parallelism: Processes and Threads
- Approaches to Shared-Memory Parallelization
- OpenMP: Overview
- OpenMP: A Practical Introduction
- Hybrid Parallelism

Distributed-Memory Parallelism



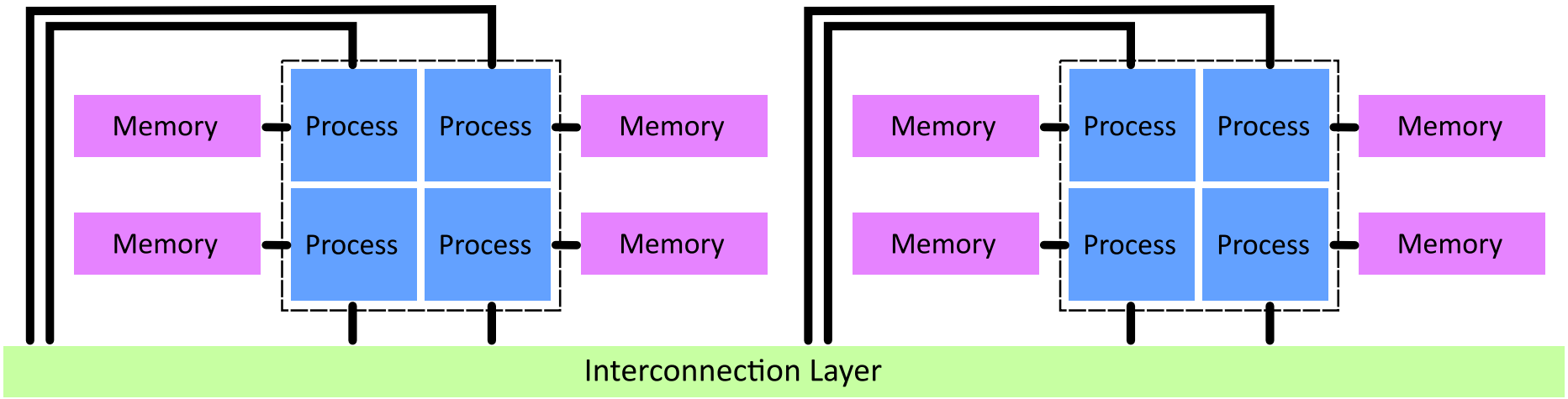
- MPI-type distributed-memory parallelism.

Distributed-Memory Parallelism



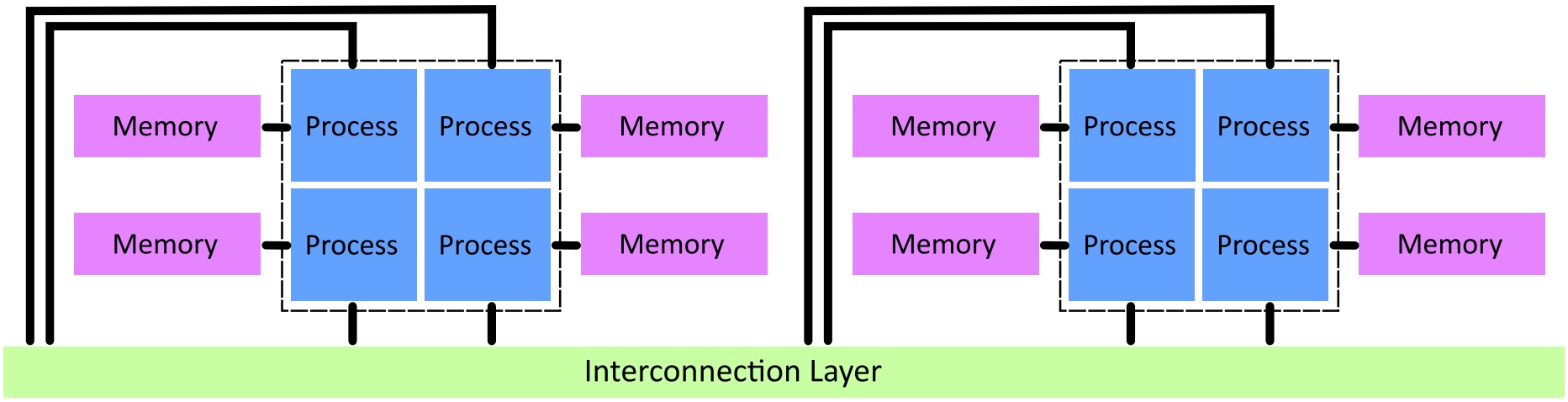
- MPI-type distributed-memory parallelism
- Multicore CPU $\rightarrow$ multiple MPI processes per CPU. Interconnection via networking software stack.
- Multiple CPUs per node $\rightarrow$ Interconnection via networking software stack.
- Between nodes: Interconnect via networking hardware.

Distributed-Memory Parallelism



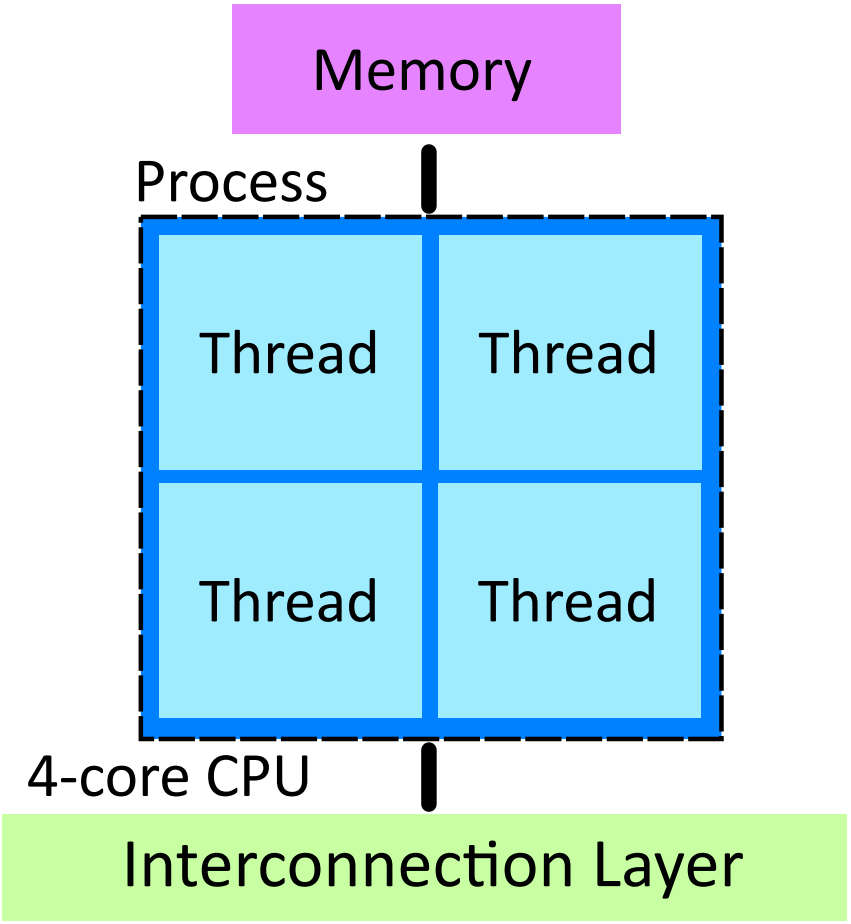
- Iowa's Helium cluster has 2 quad-core CPUs per node.
- MPI: each core runs one process with its own memory. Communication via network stack within node and with other nodes.

Distributed-Memory Parallelism



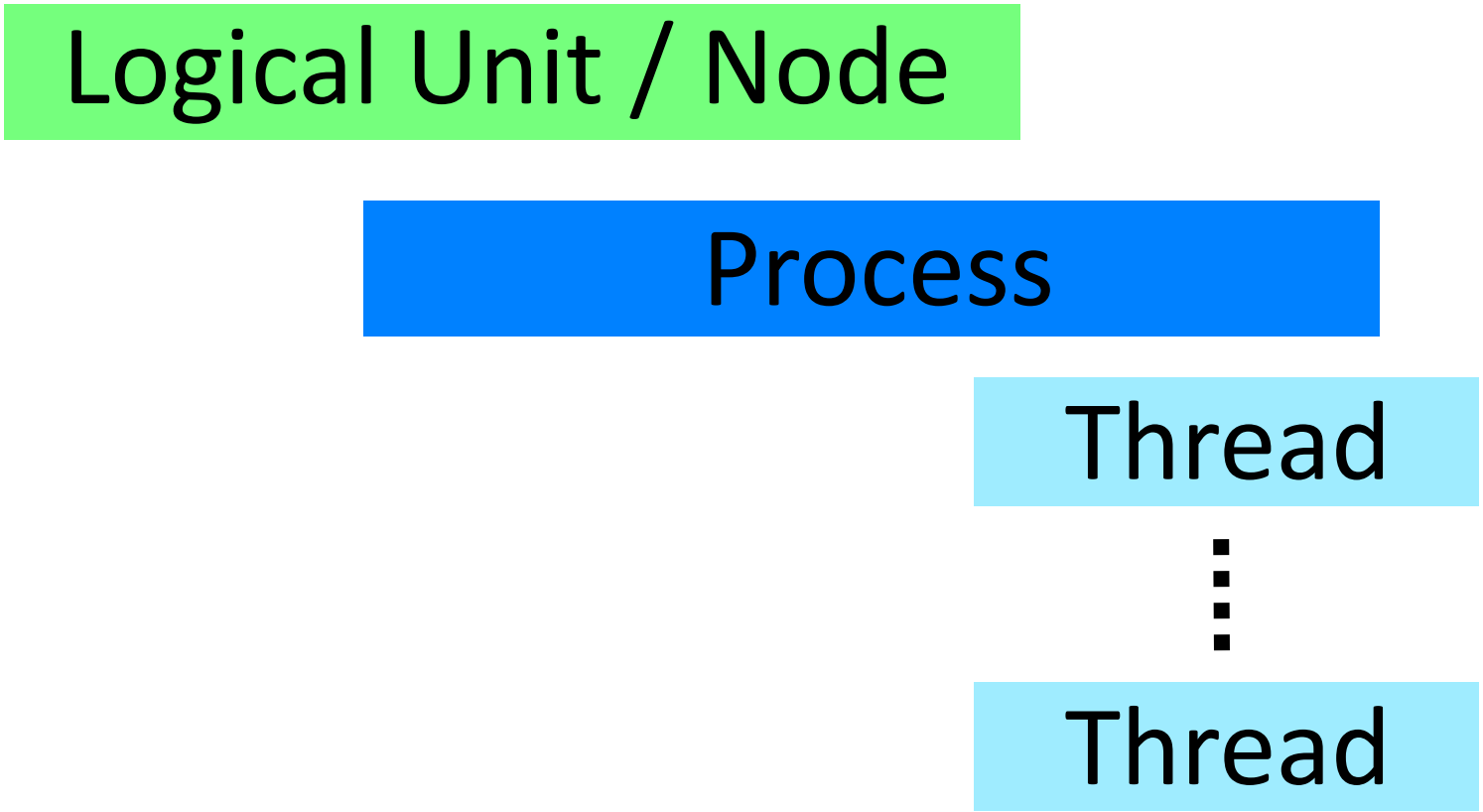
- Iowa's Helium cluster has 2 quad-core CPUs per node.
- MPI: each core runs one process with its own memory. Communication via network stack within node and with other nodes.
- This seems like a great waste! Why not share the memory within a CPU or even a node and bypass the interconnect?

Threaded Shared-Memory Parallelism



- A **process** may have multiple parallel **threads** sharing the same memory (multi-threading). Each process has at least one thread.
- One thread per physical core (note: Intel Hyper-Threading – 2 virtual cores per physical core)

Parallelism Hierarchy



- Only one thread communicates with the “outside”.
- At least one thread per process (exactly 1 in classical MPI setup).
- No shared memory between typical cluster nodes.
- Number of cores per node keeps increasing.

Architectures for Shared-Memory Parallelism

Multi-core laptop

SGI Altix UV "Blacklight" at PSC

- Any shared memory symmetric multiprocessor machine (SMP). $\rightarrow$ Any modern laptop/desktop; any one cluster compute node. Limited to physical unit (cluster node).
- Non-Uniform Memory Access machines – system-wide shared memory architectures $\rightarrow$ 1000s of cores.

Routes to Shared-Memory Multi-Threading

Routes to Shared-Memory Multi-Threading

- Compiler-based automatic parallelization.
  - Code unchanged.
  - Compiler specific. Results vary greatly.
    Won't parallelize complex loops.
  - Number of threads per process usually set at compile time.

Routes to Shared-Memory Multi-Threading

- Compiler-based automatic parallelization.
  - Code unchanged.
  - Compiler specific. Results vary greatly.
    Won't parallelize complex loops.
  - Number of threads per process usually set at compile time.
- PThreads library
  - Provides full control and run-time allocation of threads.
  - Requires major code re-write from single-thread version.
    Lots of pedestrian work.
  - Available only for C.

Routes to Shared-Memory Multi-Threading

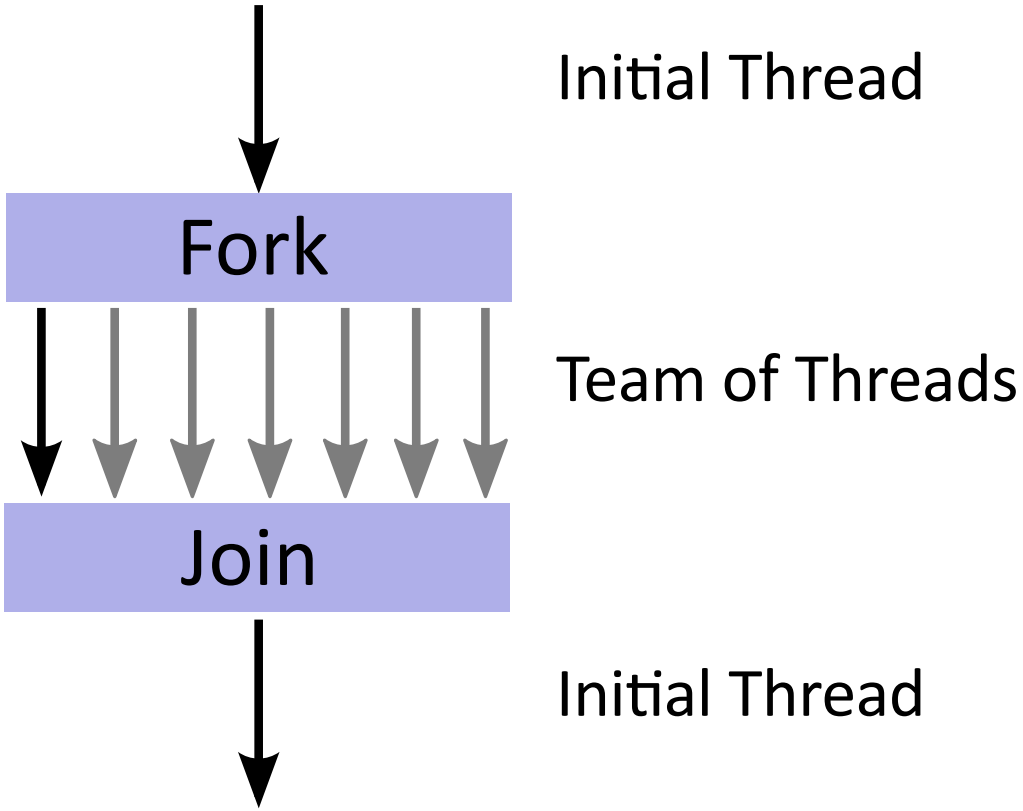
- Compiler-based automatic parallelization.
  - Code unchanged.
  - Compiler specific. Results vary greatly.
    Won't parallelize complex loops.
  - Number of threads per process usually set at compile time.
- PThreads library
  - Provides full control and run-time allocation of threads.
  - Requires major code re-write from single-thread version.
    Lots of pedestrian work.
  - Available only for C.
- **OpenMP**
  - Full control, run-time thread allocation.
  - Only small code changes needed.
  - Convenient, high-level interface.

Introduction: What is OpenMP?

`http://www.openmp.org`

- OpenMP us an application programming interface (API) for shared-memory multi-threading.
- OpenMP is not an external library. It is implemented directly by the compiler.
- OpenMP 1.0 in 1997; Current: OpenMP 3.0 (3.1 coming).
- OpenMP works with C/C++ and Fortran.

Basic: The OpenMP Concept



- Fork-Join multi-threading model for mixing serial with shared-memory-parallel program sections.

Basics: Compiling with OpenMP

- The compiler must be told to use OpenMP. This is accomplished via compiler flags that differ between compilers.

GNU gcc/gfortran (open source)

```
gcc -fopenmp ...   , g++ -fopenmp ...,
gfortran -fopenmp ...
```

Intel Compilers

```
icc -openmp ..., icpc -openmp ...,
ifort -openmp ...
```

Basics: OpenMP API Components

OpenMP has three basic components:

- Pre-processor directives.

```
C/C++:  #pragma omp parallel ...
   F90 :  !$OMP PARALLEL ... !$OMP END PARALLEL
```

- Runtime library routines.

```
C/C++:  #include <omp.h>
   F90 :  use module omp_lib
```

- Environment variables. To be set in the shell, e.g.,

```
OMP_NUM_THREADS=8
```

A First OpenMP Program in F90

A first OpenMP Program in Fortran 90:

```
program omp1
  use omp_lib
  implicit none

  !$OMP PARALLEL          ! Fork threads
  write(6,*) "my thread id: ", omp_get_thread_num()
  !$OMP END PARALLEL      ! Join threads

end program omp1
```

A First OpenMP Program in C

A first OpenMP Program in C:

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

int main(void) {

  #pragma omp parallel   // Fork threads
  {
   printf("my thread id: %d\n", omp_get_thread_num());
  }                      // Join threads

}
```

In this example (and in the previous Fortran one), every thread redundantly executes the code in the parallel region.

Useful OpenMP Library Functions

- `omp_get_thread_num`: current thread index (0, 1, . . .)
- `omp_get_num_threads`: size of the active team
- `omp_set_num_threads`: set size of the thread team (make this call outside of a parallel region)
- `omp_get_max_threads`: maximum number of threads
- `omp_get_num_procs`: number of cores available

There are a couple more – see the OpenMP reference manual for a full list and description.

Parallelizing Loops

Finally doing something useful...

The compute-intense parts of most codes are loops over large datasets that carry out many floating point operations.

```
do k=1,nz
  do j=1,ny
    do i=1,nx

      [do something crazy complicated]

    enddo
  enddo
enddo
```

`[do something crazy complicated]` is executed nx*ny*nz times!

Basic Worksharing: Parallel do/for

Parallelizing "do/for" loops:

- C `for` loop

```
#pragma omp parallel
#pragma omp for
for(i=0;i<n;i++) {
  // do something in parallel
}
```

or, using a combined directive:

```
#pragma omp parallel for
for(i=0;i<n;i++) {
  // do something in parallel
}
```

Basic Worksharing: Parallel do/for

Parallelizing "do/for" loops:

- Fortran do loop

```
!$OMP PARALLEL
!$OMP DO
do i=1,n
  ! do something in parallel
enddo
!$OMP END DO
!$OMP END PARALLEL
```

or, using a combined directive:

```
!$OMP PARALLEL DO
do i=1,n
  ! do something in parallel
enddo
!$OMP END PARALLEL DO
```

Basic Worksharing with Loops: Rules

- Only standard `for`/`do` loops can be parallelized. `while` loops cannot.
- Program correctness must not depend upon which thread executes a particular iteration. For example:

Does not work:

```
x(1) = 0
do i=2,n
   x(i) = x(i-1) + f
enddo
```

Works:

```
do i=1,n
  x(i) = (i-1)*f
enddo
```

- Branching statements such as `break`, `exit`, `continue`, `goto`, `return` etc. are not allowed.

Basic Worksharing: Basic loop example

A somewhat mindless example:

Let's write a simple code that fills an array of length n with numbers and see how this can be sped up with OpenMP.

Basic Worksharing: Basic loop example – serial

```
program omp2ser
  implicit none
  integer :: i
  integer, parameter :: n = 260000000
  real*8,allocatable :: myarray(:)

  allocate(myarray(n))
  do i=1,n
    myarray(i) = 5*i**3 + i**2 + i + sin(1.0*i)**2
  enddo
  deallocate(myarray)

end program omp2ser
```

Basic Worksharing: Basic loop example – parallel

```
program omp2
  implicit none
  integer :: i
  integer, parameter :: n = 260000000
  real*8,allocatable :: myarray(:)

  allocate(myarray(n))
  !$OMP PARALLEL DO
  do i=1,n
     myarray(i) = 5*i**3 + i**2 + i + sin(1.0*i)**2
  enddo
  !$OMP END PARALLEL DO
  deallocate(myarray)

end program omp2
```

Basic Worksharing: Basic loop example – parallel

Let's try this on `helium`:

```
export OMP_NUM_THREADS=1;time ./omp2
export OMP_NUM_THREADS=2;time ./omp2
export OMP_NUM_THREADS=4;time ./omp2
export OMP_NUM_THREADS=8;time ./omp2
```

$\rightarrow$ Won't see major improvement at more than 8 threads, since `helium` only has 8 physical cores.

Basic Worksharing: Basic loop example – parallel

Let's reduce the problem size n by a factor of 100 and try again:

```
export OMP_NUM_THREADS=1;time ./omp2b100
export OMP_NUM_THREADS=2;time ./omp2b100
export OMP_NUM_THREADS=4;time ./omp2b100
export OMP_NUM_THREADS=8;time ./omp2b100
```

Basic Worksharing: Basic loop example – parallel

Let's reduce the problem size n by a factor of 100 and try again:

```
export OMP_NUM_THREADS=1;time ./omp2b100
export OMP_NUM_THREADS=2;time ./omp2b100
export OMP_NUM_THREADS=4;time ./omp2b100
export OMP_NUM_THREADS=8;time ./omp2b100
```

$\rightarrow$ No improvement with increasing number of threads! Why?

Basic Worksharing: Basic loop example – parallel

Let's reduce the problem size n by a factor of 100 and try again:

```
export OMP_NUM_THREADS=1;time ./omp2b100
export OMP_NUM_THREADS=2;time ./omp2b100
export OMP_NUM_THREADS=4;time ./omp2b100
export OMP_NUM_THREADS=8;time ./omp2b100
```

$\rightarrow$ No improvement with increasing number of threads! Why? OpenMP "fork/join" process requires time. If problem size too small, forking/joining dominates compute time.

Basic Worksharing: Private/Shared Vars

In more complex settings, it becomes necessary to tell OpenMP what variables are **private** to each thread and which are **shared**.

By default, all variables are assumed to be **shared**. Exceptions: Loop counters of the outermost loop and variables declared inside the parallel region (only in C).

Basic Worksharing: Private/Shared Vars

In more complex settings, it becomes necessary to tell OpenMP what variables are **private** to each thread and which are **shared**.

By default, all variables are assumed to be **shared**. Exceptions: Loop counters of the outermost loop and variables declared inside the parallel region (only in C).

Declaring vars private/shared:

Basic Worksharing: Private/Shared Vars

In more complex settings, it becomes necessary to tell OpenMP what variables are **private** to each thread and which are **shared**.

By default, all variables are assumed to be **shared**. Exceptions: Loop counters of the outermost loop and variables declared inside the parallel region (only in C).

Declaring vars private/shared:

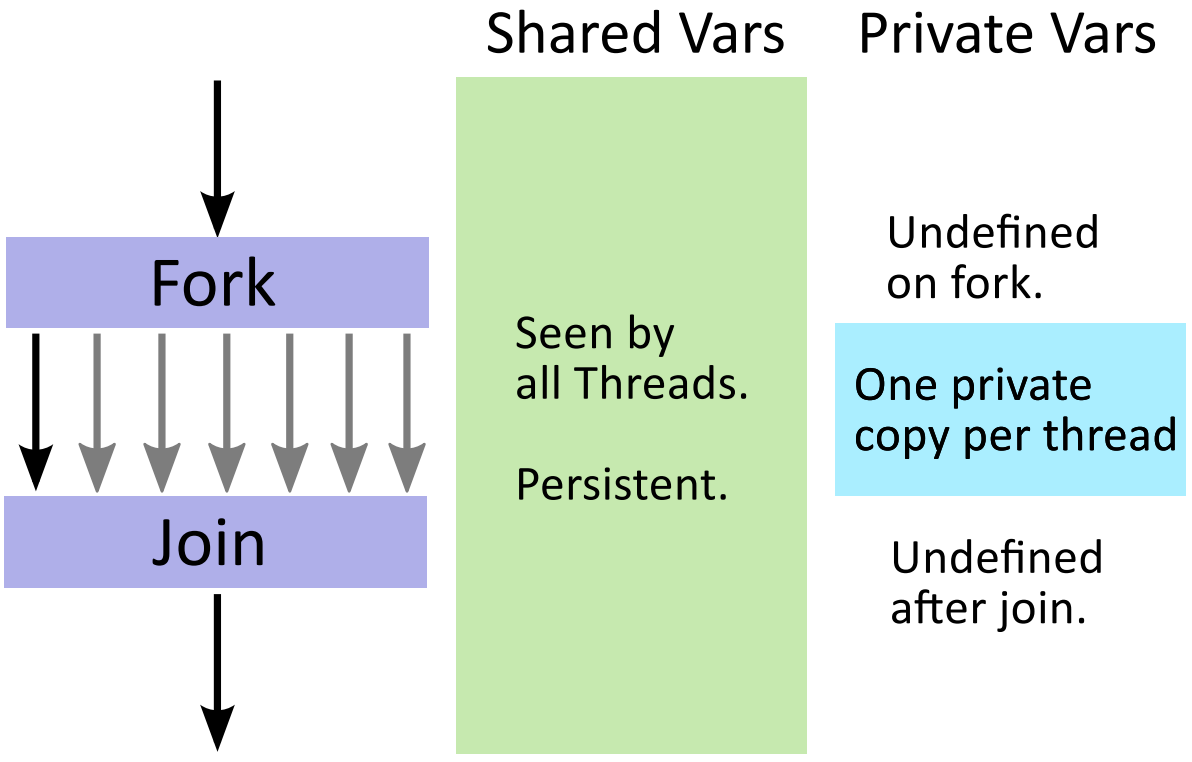
→ In C:

```
#pragma omp parallel for private(pvar1,pvar2) shared(svar)
```

→ In Fortran:

```
!$OMP PARALLEL DO PRIVATE(pvar1,pvar2) SHARED(svar)
[...]
!$OMP END PARALLEL DO
```

Basic Worksharing: Private/Shared Vars



- Shared vars: Seen by all threads, but not more than one thread must write to a shared var at a time. Persistent.
- Private vars: Private "copy" for each thread. Undefined when the thread team is created; undefined after parallel region.

Private/Shared Example

Consider this code snippet:

```
[...]
!$OMP PARALLEL DO
do i=1,n
   x = 5*i**3 + i**2 + i + sin(1.0*i)**2
   myarray(i) = x
enddo
!$OMP END PARALLEL DO
[...]
```

`i` is private, but `n`, `x`, and `myarray` are shared.

Private/Shared Example

Consider this code snippet:

```
[...]
!$OMP PARALLEL DO
do i=1,n
   x = 5*i**3 + i**2 + i + sin(1.0*i)**2
   myarray(i) = x
enddo
!$OMP END PARALLEL DO
[...]
```

`i` is private, but `n`, `x`, and `myarray` are shared.

data race condition: `x` **is updated inconsistently and uncontrollably by multiple threads!**

Private/Shared Example: fixed

Fixed:

```
[...]
!$OMP PARALLEL DO PRIVATE(x)
do i=1,n
   x = 5*i**3 + i**2 + i + sin(1.0*i)**2
   myarray(i) = x
enddo
!$OMP END PARALLEL DO
[...]
```

`i` and `x` are private. `n`, `myarray` are shared.
Outside the parallel segment, `i` and `x` are undefined.

Another Loop Example

Suppose we wanted to parallelize

```
[...]
sum = 0.0d0
do i=1,n
  val = f(i)
  sum = sum + val
enddo
[...]
```

Another Loop Example

Suppose we wanted to parallelize

```
[...]
sum = 0.0d0
do i=1,n
  val = f(i)
  sum = sum + val
enddo
[...]
```

First attempt:

```
[...]
sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(val)
do i=1,n
  val = f(i)
  sum = sum + val
enddo
!$OMP END PARALLEL DO
[...]
```

Another Loop Example

Suppose we wanted to parallelize

```
[...]
sum = 0.0d0
do i=1,n
  val = f(i)
  sum = sum + val
enddo
[...]
```

First attempt:

```
[...]
sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(val)
do i=1,n
  val = f(i)
  sum = sum + val
enddo
!$OMP END PARALLEL DO
[...]
```

Problem: Race condition in the updating of sum!

Another Loop Example: fixed (1: CRITICAL)

One way of fixing this is the !$OMP CRITICAL directive:

```
[...]
sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(val)
do i=1,n
  val = f(i)
  !$OMP CRITICAL
  sum = sum + val
  !$OMP END CRITICAL
enddo
!$OMP END PARALLEL DO
[...]
```

The CRITICAL directive ensures that only one thread accesses sum at a time.

Another Loop Example: fixed (2: reduction)

An even better way of dealing with this issue is a **sum reduction**:

```
[...]
sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(val) REDUCTION(+:sum)
do i=1,n
  val = f(i)
  sum = sum + val
enddo
!$OMP END PARALLEL DO
[...]
```

The REDUCTION clause tells OpenMP that the team of threads must safely add to sum so that it assumes the same value as in the serial case.

Numerical reduction operators: +, -, *

See reference manual/sheet for more.

Advanced Stuff: Ask Google!

Some more useful clauses that modify OpenMP behavior

- `NO WAIT` clause – don't wait after a loop (or other directive) *inside* a parallel section until all threads are done.
- `SCHEDULE(STATIC)` – evenly divide iterations of a loop among threads.
- `SCHEDULE(DYNAMIC[,chunk])` – divide work into `chunk`-sized parcels. If a thread is done with a chunk, it grabs another one. Default `chunk` size is 1.
- `SCHEDULE(GUIDED[,chunk])` – divide work into chunks of exponentially decreasing size. `chunk` is the minimum chunk size. Default is 1.

Advanced Stuff: Ask Google!

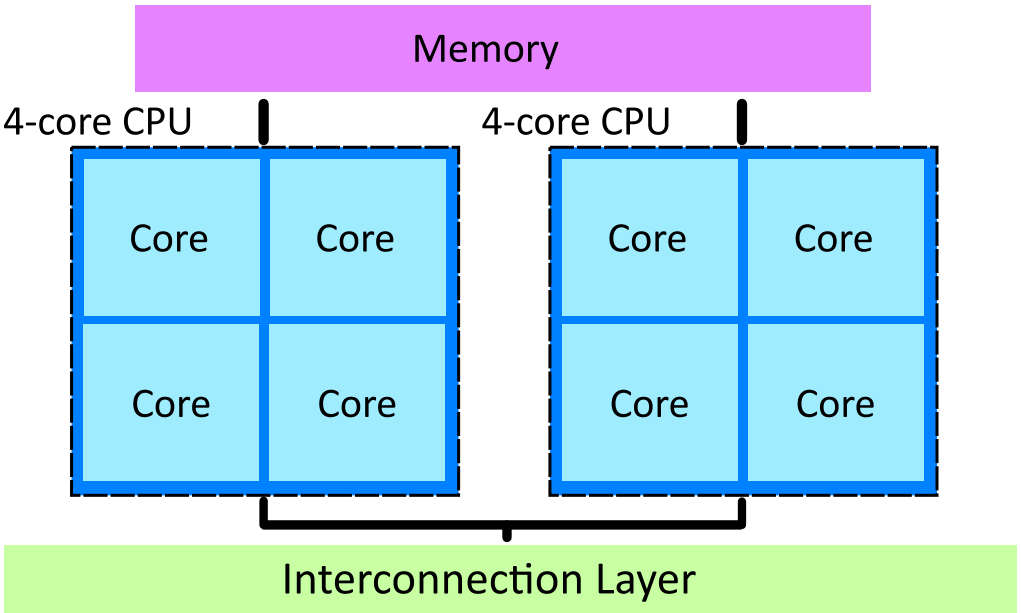
Some more useful clauses that modify OpenMP behavior

- `NO WAIT` clause – don't wait after a loop (or other directive) *inside* a parallel section until all threads are done.
- `SCHEDULE(STATIC)` – evenly divide iterations of a loop among threads.
- `SCHEDULE(DYNAMIC[,chunk])` – divide work into `chunk`-sized parcels. If a thread is done with a chunk, it grabs another one. Default `chunk` size is 1.
- `SCHEDULE(GUIDED[,chunk])` – divide work into chunks of exponentially decreasing size. `chunk` is the minimum chunk size. Default is 1.

There are a more OpenMP directives.

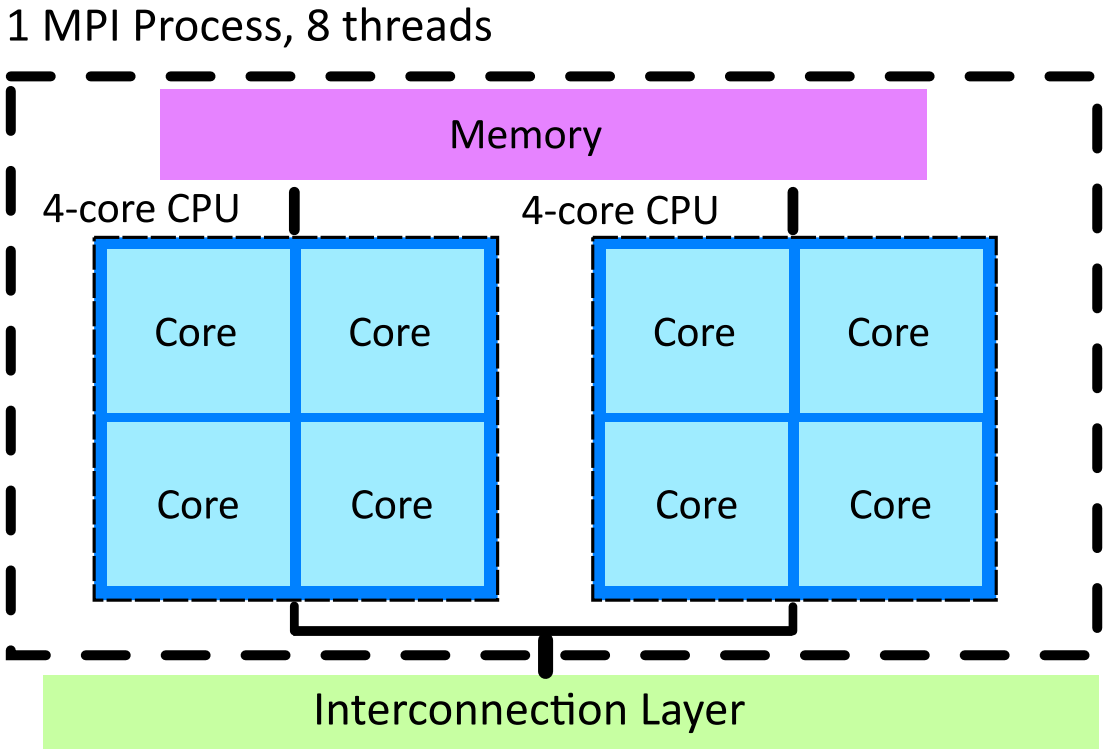
- `SECTIONS` – non-iterative work sharing.
- `BARRIER` – force threads to wait for each other.
- `ORDERED` – force sequential order in a loop.
- `MASTER` – section in a loop executed only by the master.
- `SINGLE` – section in a loop executed only by one thread.

Hybrid Parallelism



- Modern cluster supercomputers have nodes with an increasing number of cores. `Helium`: 8 cores per node (two 4-core CPUs).
- All cores within a node share the same main memory.

Hybrid Parallelism



Hybrid Parallelism:

- Node-local OpenMP.
- Internode MPI.
- Reduces communication overhead. Optimal number of MPI processes per node depends on software & hardware.