PRACE

PARTNERSHIP FOR ADVANCED COMPUTING IN EUROPE

Using HPCFS

Leon Kos, UL

PRACE Autumn School 2013 - Industry Oriented HPC Simulations, September 21-27, University of Ljubljana, Faculty of Mechanical Engineering, Ljubljana, Slovenia

University *of Ljubljana*
Faculty *of Mechanical Engineering*

CAPACITIES

e-infrastructure

Basic HPCFS cluster usage

- Setting GNOME or KDE desktop locale preferences for keyboard, LANG environment
- Using NX client (Disconnect, Terminate, Logout)
- Console commands in Linux
- Editors for programming (emacs, gedit, kate, eclipse, vi, pico, …)

Modules

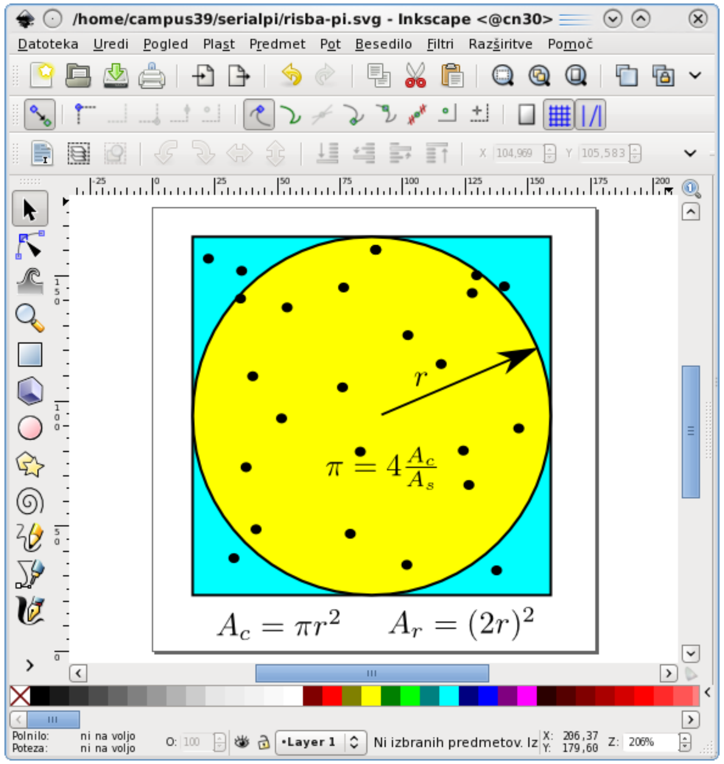
- module avail
- module help/info
- module display
- module load/unload
- module list
- module purge

Pi example

emacs pi.py

single python pi.py

```
import random, math
total=100000
in_circle=0
for i in range(total):
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)
    r = math.sqrt(x*x+y*y)
    if r < 1.0:
        in_circle += 1
print 'Pi =', 4.0*in_circle/total
```


/home/campus39/serialpi/risba-pi.svg - Inkscape <@cn30>
Datoteka Uredi Pogled Plast Predmet Pot Besedilo Filtri Razširitve Pomoč
r
$\pi \doteq 4\frac{A_c}{A_s}$
$A_c = \pi r^2$
$A_r = (2r)^2$
Polnilo: ni na voljo
Poteza: ni na voljo
•Layer 1
Ni izbranih predmetov. Iz
X: 206,37 Y: 179,60 Z: 206%

Load Sharing Facility (LSF)

- Batch scheduler for all programs
- Compiled-in OpenMPI support
- bsub
- bjobs
- bkill
- bpeek
- Aliases for interactive usage of nodes
  - node, single

An Introduction to MPI

Leon Kos, UL

PRACE Autumn School 2013 - Industry Oriented HPC Simulations, September 21-27, University of Ljubljana, Faculty of Mechanical Engineering, Ljubljana, Slovenia

The Message-Passing Model

- Unlike the shared memory model, resources are local
- MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

Why MPI

- Scalable to thousands of processes
- MPI provides a powerful, efficient, and *portable* way to express parallel programs
- Many libraries use MPI and thus programs eliminate the need of knowing programming in MPI.

Minimal MPI

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

Try to run it with LSF

1. module load intel/11.1 openmpi/1.4.4
2. mpicc hello-mpi.c
3. bsub –n 6 mpirun a.out
4. mail

- Fortran example uses `mpif90 hello-mpi.f90` instead

```
program main
include 'mpif.h'
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

Rank and communicator

- A process is identified by its *rank* in the group associated with a communicator
- **`MPI_Comm_size`** reports the number of processes.
- **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

Updated hello-mpi.{c,f90}

```c
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

```fortran
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Point-To-Point Message Passing – Data transfer and Synchronization

- The sender process cooperates with the destination process
- The communication system must allow the following three operations
  - send(message)
  - receive (message)
  - synchronisation

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- Point-to-point (send/recv) isn’t the only way

Send/Receive P-t-P

```
program main
implicit none
include 'mpif.h'
integer ierr, rank, size
integer status(MPI_STATUS_SIZE)
real data(2)

call MPI_INIT( ierr )
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
if (rank .eq. 0) then
  data(1)=1
  data(2)=2
  call MPI_SEND(data, 2, MPI_REAL,  1,  2929, MPI_COMM_WORLD, ierr)
else if (rank.eq.1) then
  call MPI_RECV(data, 2, MPI_REAL,  0, 2929, MPI_COMM_WORLD, status, ierr)
  print *, data(1), data(2)
endif
call MPI_FINALIZE( ierr )
end
```

Standard Send and Receive in C

-
  ```
  int MPI_Send(void *buf, int count,
      MPI_Datatype, type, int dest, int tag,
      MPI_Comm comm);
  ```

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

C example

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char * argv[])
{
         int err, size, rank;
        MPI_Status status;
        float data[2];
        err = MPI_Init(&argc, &argv);
        Andrew Emerson
        err = MPI_Init(&argc, &argv);
        err = MPI_Comm_size(MPI_COMM_WORLD, &size);
        err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if( rank == 0 ) {
                data[0] = 1.0, data[1] = 2.0;
                MPI_Send(data, 2, MPI_FLOAT, 1, 1230, MPI_COMM_WORLD);
        } else if( rank == 1 ) {
                MPI_Recv(data, 2, MPI_FLOAT, 0, 1230, MPI_COMM_WORLD, &status);
                printf(”%d: a[0]=%f a[1]=%f\n”, rank, a[0], a[1]);
        }
        err = MPI_Finalize();
}
```

Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency

Summary

- MPI is a **standard** for message-passing and has numerous implementations (OpenMPI, IntelMPI, MPICH, etc)
- MPI uses send and receive calls to manage communications between two processes (point-to-point)
- The calls can be blocking or non-blocking.
- Non-blocking calls can be used to overlap communication with computation but wait routines are needed for synchronization.
- Deadlock is a common error and is due to incorrect order of send/receive

Introduction to OpenMP

Introduction to OpenMP

Outline

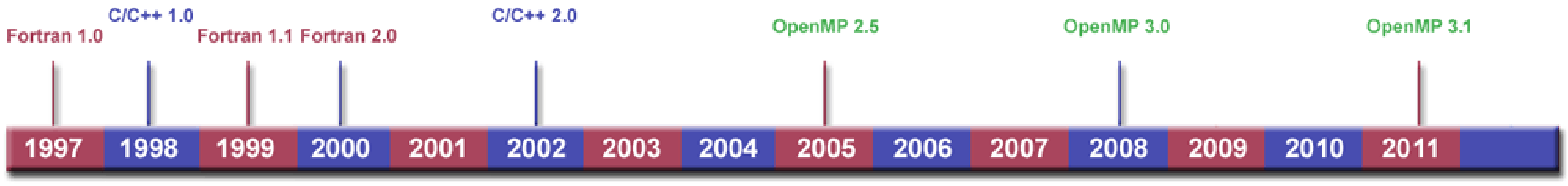
- What is OpenMP?
- Timeline
- Main Terminology
- OpenMP Programming Model
- Main Components
- Parallel Construct
- Work-sharing Constructs
  - sections, single, workshare
- Data Clauses
  - default, shared, private, firstprivate, lastprivate, threadprivate, copyin

What is OpenMP?

OpenMP (*Open specifications for Multi Processing*)

- is an API for shared-memory parallel computing;
- is an open standard for portable and scalable parallel programming;
- is flexible and easy to implement;
- is a specification for a set of compiler directives, library routines, and environment variables;
- is designed for C, C++ and Fortran.

Timeline



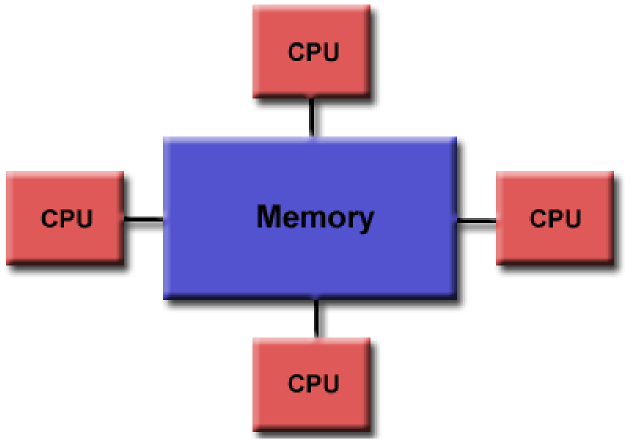
- OpenMP 4.0 Release Candidate 1 was released in November 2012.
- http://openmp.org/

Main Terminology

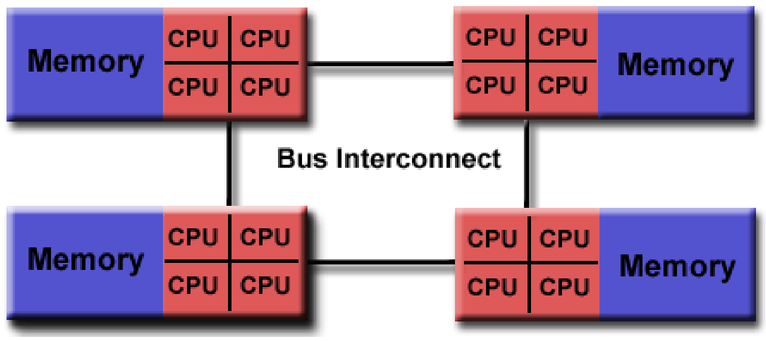
1. OpenMP thread: a lightweight process
2. thread team: a set of threads which co-operate on a task
3. master thread: the thread which co-ordinates the team
4. thread-safety: correctly executed by multiple threads
5. OpenMP directive: line of code with meaning only to certain compilers
6. construct: an OpenMP executable directive
7. clause: controls the scoping of variables during the execution

OpenMP Programming Model

OpenMP is designed for multi-processor/core UMA or NUMA shared memory systems.



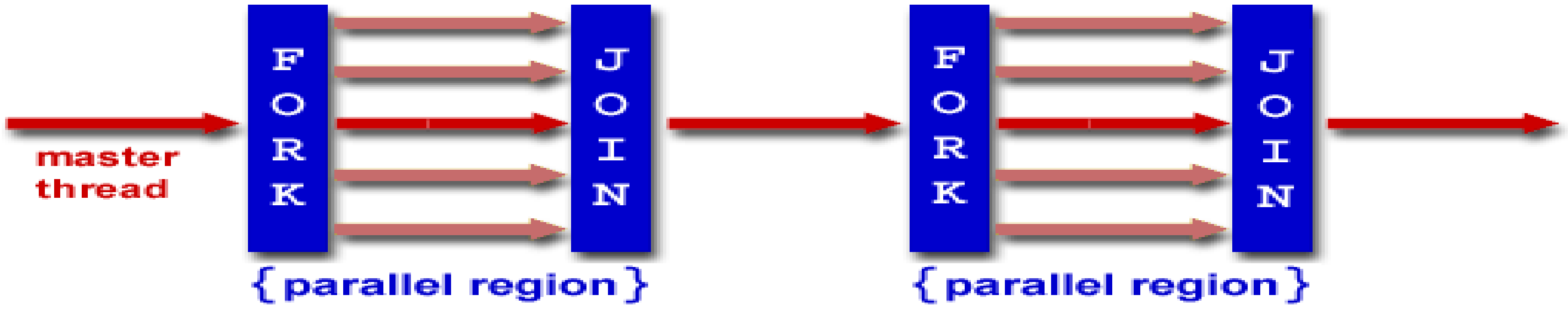
UMA



NUMA

Execution Model:

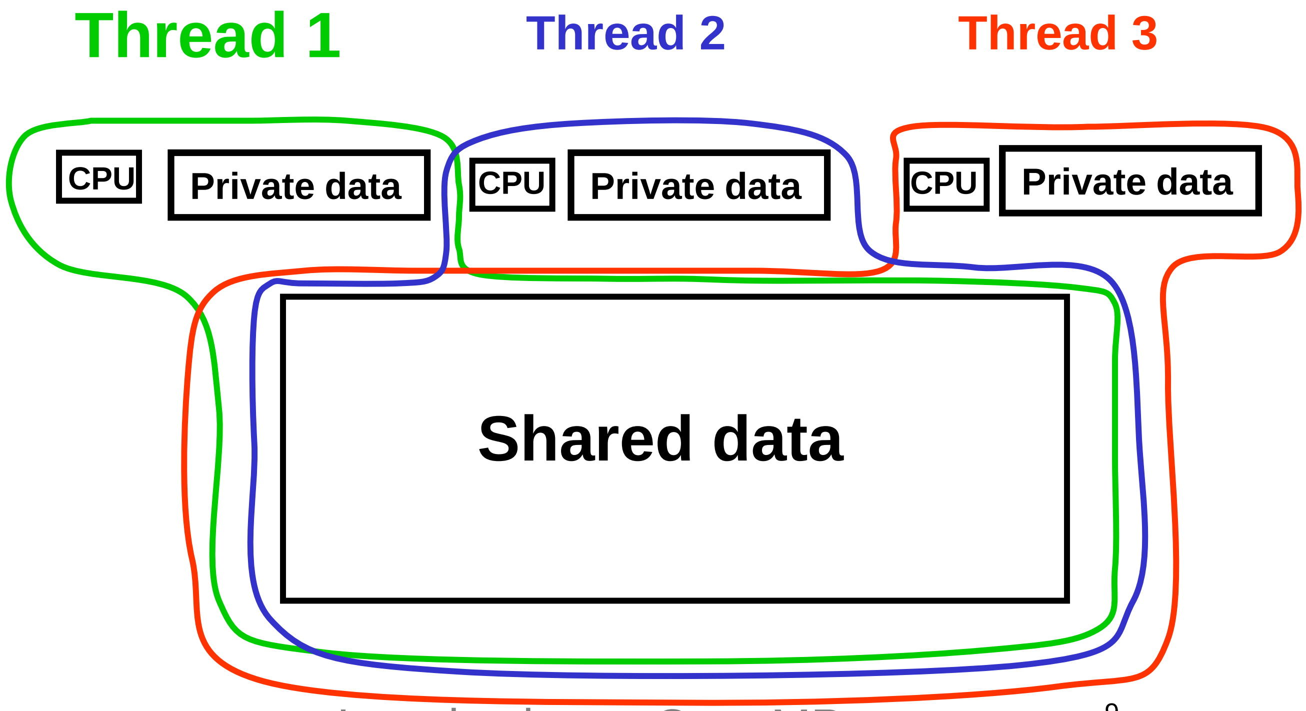
- Thread-based Parallelism
- Compiler Directive Based
- Explicit Parallelism
- Fork-Join Model



- Dynamic Threads
- Nested Parallelism

Memory Model:

- All threads have access to the shared memory.
- Threads can share data with other threads, but also have private data.
- Threads sometimes synchronise against data race.
- Threads cache their data; Use OpenMP flush



Main Components

- Compiler Directives and Clauses: appear as comments, executed when the appropriate OpenMP flag is specified
  - Parallel construct
  - Work-sharing constructs
  - Synchronization constructs
  - Data Attribute clauses

C/C++:#pragma omp *directive-name [clause[clause]...]*

Fortran free form: !$omp *directive-name [clause[clause]...]*

Fortran fixed form: !$omp | c$omp | *$omp *directive-name [clause[clause]...]*

Compiling:

	Compiler	Flag
Intel	icc (C) icpc (C++) ifort (Fortran)	-openmp
GNU	gcc (C) g++ (C++) g77/gfortran (Fortran)	-fopenmp
PGI	pgcc (C) pgCC (C++) pg77/pgfortran (Fortran)	-mp

See: http://openmp.org/wp/openmp-compilers/ for the full list.

- Runtime Functions: for managing the parallel program
  - omp_set_num_threads(n) - set the desired number of threads
  - omp_get_num_threads() - returns the current number of threads
  - omp_get_thread_num() - returns the id of this thread
  - omp_in_parallel() – returns .true. if inside parallel region

  and more.

  For C/C++: Add #include<omp.h>

  For Fortran: Add use omp_lib

- Environment Variables: for controlling the execution of parallel program at run-time.
  - csh/tcsh: setenv OMP_NUM_THREADS n
  - ksh/sh/bash: export OMP_NUM_THREADS=n

  and more.

Parallel Construct

- The fundamental construct in OpenMP.
- Every thread executes the same statements which are inside the parallel region simultaneously.
- At the end of the parallel region there is an implicit barrier for synchronization

C/C++:

```
#pragma omp parallel [clauses]
{
   …
}
```

Fortran:

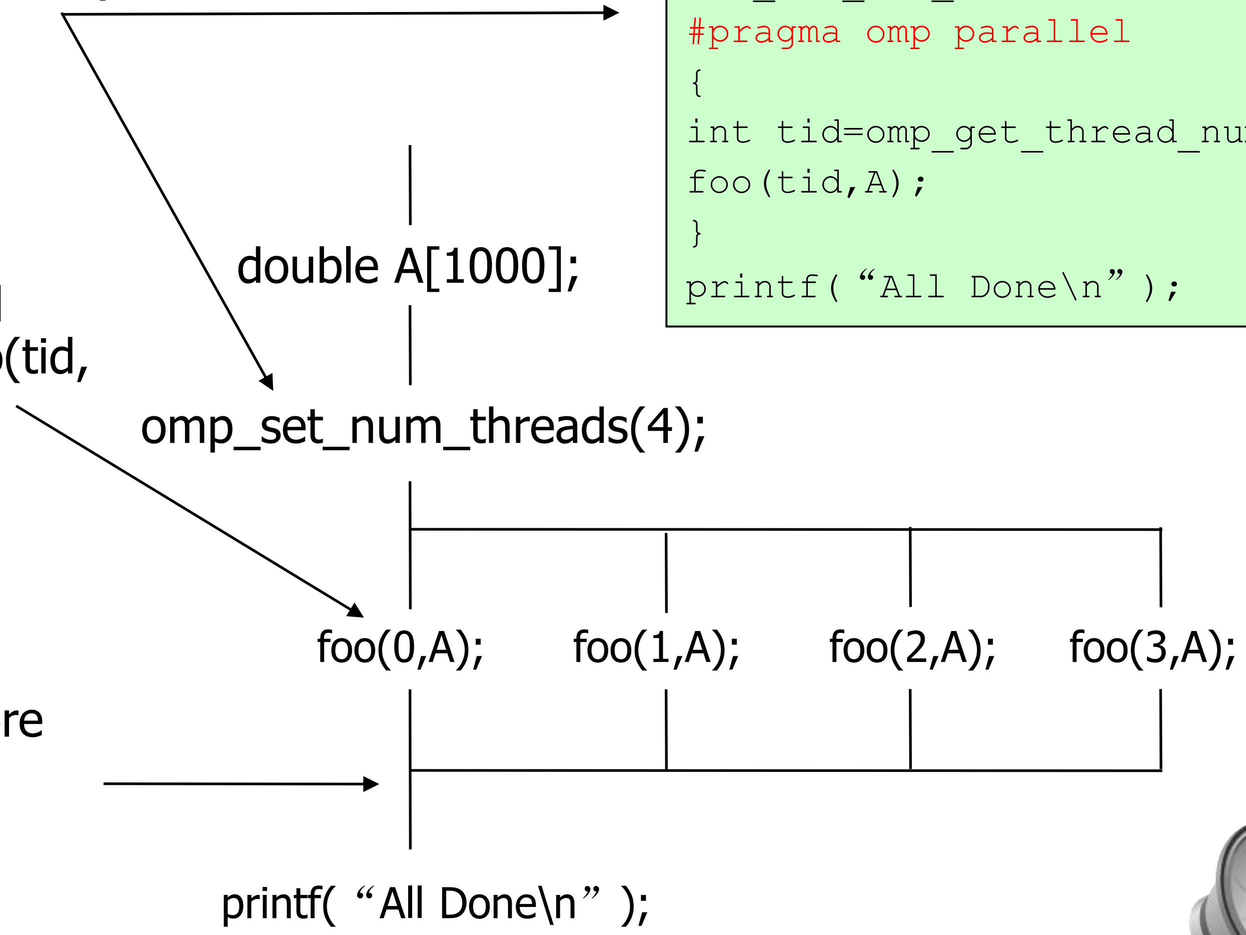
```
!$omp parallel [clauses]
   ...
!$omp end
parallel
```

- Create a 4-thread parallel region

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
int tid=omp_get_thread_num();
foo(tid,A);
}
printf(“All Done\n”);
```

- Each thread with tid from 0 to 3 calls foo(tid, A)

- Threads wait for all treads to finish before proceeding



Hello World Example:

C:

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

int main(){
#pragma omp parallel

printf("Hello from thread %d out
of %d\n", omp_get_thread_num(),
omp_get_num_threads());
}
```

Fortran:

```
program hello
use omp_lib

implicit none
!$omp parallel

PRINT*, 'Hello from
thread',omp_get_thread_num(),'out
of',omp_get_num_threads()

!$omp end parallel

end program hello
```

Compile: (Intel)

```
>icc -openmp hello.c -o a.out
>ifort -openmp hello.f90 -o a.out
```

Execute:

```
>export OMP_NUM_THREADS=4
>./a.out
Hello from thread 0 out of 4
Hello from thread 3 out of 4
Hello from thread 1 out of 4
Hello from thread 2 out of 4
```

- **Dynamic threads:**
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - omp_set_dynamic(), OMP_DYNAMIC
  - omp_get_dynamic()

- **Nested parallel regions:**
  - If a parallel directive is encountered within another parallel directive, a new team of threads will be created.
  - omp_set_nested(), OMP_NESTED
  - omp_get_nested()

- **If Clause:**
  - Used to make the parallel region directive itself conditional.
  - Only execute in parallel if expression is true.

C/C++:

(Checks the size of the data)

```
#pragma omp parallel if(n>100)
{
    …
}
```

Fortran:

```
!$omp parallel if(n>100)
   ...
!$omp end parallel
```

- **nowait Clause:**
  - allows threads that finish earlier to proceed without waiting

C/C++:

```
#pragma omp parallel nowait
{
    …
}
```

Fortran:

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

Data Clauses

- Used in conjunction with several directives to control the scoping of enclosed variables.
  - default(*shared|private|none*): The default scope for all of the variables in the parallel region.

  - shared(*list*): Variable is shared by all threads in the team. All threads can read or write to that variable.

C: #pragma omp parallel default(none), shared(n)

Fortran: !$omp parallel default(none), shared(n)

  - private(*list*): Each thread has a private copy of variable. It can only be read or written by its own thread.

C: #pragma omp parallel default(none), shared(n), private(tid)

Fortran: !$omp parallel default(none), shared(n), private(tid)

- Most variables are shared by default
  - C/C++: File scope variables, static
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - Both: dynamically allocated variables
- Variables declared in parallel region are always private

- How do we decide which variables should be shared and which private?
  - Loop indices - private
  - Loop temporaries - private
  - Read-only variables - shared
  - Main arrays - shared

Example:

C:

```
#include<omp.h>
#include<stdio.h>
int tid, nthreads;
int main(){

#pragma omp parallel private(tid),
shared(nthreads)
{
tid=omp_get_thread_num();
nthreads=omp_get_num_threads();
printf("Hello from thread %d out
of %d\n", tid, nthreads);
}
}
```

Fortran:

```
program hello
use omp_lib
implicit none
integer tid, nthreads

!$omp parallel private(tid),
shared(nthreads)
tid=omp_get_thread_num()
nthreads=omp_get_num_threads()
PRINT*, 'Hello from
thread',tid,'out of',nthreads
!$omp end parallel

end program hello
```

Some Additional Data Clauses:

- firstprivate(*list*): Private copies of a variable are initialized from the original global object.
- lastprivate(*list*): On exiting the parallel region, variable has the value that it would have had in the case of serial execution.
- threadprivate(*list*): Used to make global file scope variables (C/C++) or common blocks (Fortran) local.
- copyin(*list*): Copies the threadprivate variables from master thread to the team threads.

- copyprivate and reduction clauses will be described later.

Work-Sharing Constructs

- To distribute the execution of the associated region among threads in the team
- An implicit barrier at the end of the worksharing region, unless the nowait clause is added

- Work-sharing Constructs:
  - Loop
  - Sections
  - Single
  - Workshare

Sections Construct

- A non-iterative work-sharing construct.
- Specifies that the enclosed section(s) of code are to be executed by different threads.
- Each section is executed by one thread.

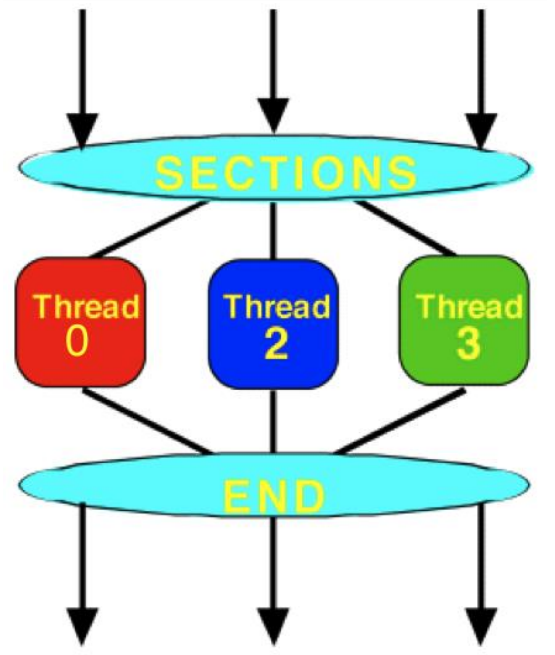
C/C++:

```
#pragma omp sections [clauses] nowait
{
   #pragma omp section
   …
   #pragma omp section
   …
}
```

Fortran:

```
!$omp sections [clauses]
   !$omp section
   ...
   !$omp section
   ...
!$omp end sections
[nowait]
```

```c
#include <stdio.h>
#include <omp.h>
int main(){
int tid;
#pragma omp parallel private(tid)
{
   tid=omp_get_thread_num();
   #pragma omp sections
   {
   #pragma omp section
   printf("Hello from thread %d \n", tid);
   #pragma omp section
   printf("Hello from thread %d \n", tid);
   #pragma omp section
   printf("Hello from thread %d \n", tid);
   }
}
}
```



```
>export
OMP_NUM_THREADS=4

Hello from thread 0
Hello from thread 2
Hello from thread
3
```

Single Construct

- Specifies a block of code that is executed by only one of the threads in the team.
- May be useful when dealing with sections of code that are not thread-safe.
- Copyprivate(*list*): used to broadcast values obtained by a single thread directly to all instances of the private variables in the other threads.

C/C++:

```
#pragma omp parallel [clauses]
{
   #pragma omp single [clauses]
   …
}
```

Fortran:

```
!$omp parallel [clauses]
   !$omp single [clauses]
   ...
   !$omp end single
!$omp end
parallel
```

Workshare Construct

- Fortran only
- Divides the execution of the enclosed structured block into separate units of work
- Threads of the team share the work
- Each unit is executed only once by one thread
- Allows parallelisation of
  - array and scalar assignments
  - WHERE statements and constructs
  - FORALL statements and constructs
  - parallel, atomic, critical constructs

```
!$omp workshare
   ...
!$omp end workshare
[nowait]
```

```
Program WSex

use omp_lib
implicit none

integer i
real a(10), b(10), c(10)
do i=1,10
   a(i)=i
   b(i)=i+1
enddo

!$omp parallel shared(a, b, c)
!$omp workshare
   c=a+b
!$omp end workshare nowait
!$omp end parallel

end program WSex
```

References

1. http://openmp.org
2. https://computing.llnl.gov/tutorials/openMP
3. http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf
4. Michael J. Quinn, Parallel Programming in C with MPI and OpenMP, Mc Graw Hill, 2003.

Thank you!