OpenMP

There are several useful functions

OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region

OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number

OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
- `int omp_get_max_threads(void)` the maximum number of threads available (often defaults to the number of cores)

OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
- `int omp_get_max_threads(void)` the maximum number of threads available (often defaults to the number of cores)
- `void omp_set_num_threads(int)` set the number of threads OpenMP can use

OpenMP

There are several useful functions

- `int omp_get_num_threads(void)` returns the number of threads in this parallel region
- `int omp_get_thread_num(void)` returns a per-thread unique number
- `int omp_get_max_threads(void)` the maximum number of threads available (often defaults to the number of cores)
- `void omp_set_num_threads(int)` set the number of threads OpenMP can use
- `int omp_get_num_procs(void)` number of processors in this system

OpenMP

And lots more functionality

OpenMP

And lots more functionality

For example, setting the environment variable `OMP_NUM_THREADS` before running the program sets the default number of threads

OpenMP

And lots more functionality

For example, setting the environment variable `OMP_NUM_THREADS` before running the program sets the default number of threads

```
OMP_NUM_THREADS=7 ./prog
```

OpenMP

And lots more functionality

For example, setting the environment variable `OMP_NUM_THREADS` before running the program sets the default number of threads

```
OMP_NUM_THREADS=7 ./prog
```

OpenMP is widely supported. For example, to compile under GCC:

```
cc -fopenmp -Wall -o prog prog.c
```

OpenMP

OpenMP is clearly naturally associated with shared memory

OpenMP

OpenMP is clearly naturally associated with shared memory

There is a distributed memory version from Intel, called *Cluster OpenMP*

OpenMP

OpenMP is clearly naturally associated with shared memory

There is a distributed memory version from Intel, called *Cluster OpenMP*

There is an undercurrent of “if your program doesn’t work well on normal OpenMP, then it won’t work well on Cluster OpenMP”

OpenMP

OpenMP

OpenMP

OpenMP

- is an evolving standard

OpenMP

OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally

OpenMP

OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling

OpenMP

OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling
- needs compiler support, unlike pthreads (but is supported by the mainstream compilers, in particular GCC, Clang and MSVC)

OpenMP

OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling
- needs compiler support, unlike pthreads (but is supported by the mainstream compilers, in particular GCC, Clang and MSVC)
- is dependent on good implementation of the compiler: if you pass control of the parallelism to a compiler you need that compiler to be good at it

OpenMP

OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling
- needs compiler support, unlike pthreads (but is supported by the mainstream compilers, in particular GCC, Clang and MSVC)
- is dependent on good implementation of the compiler: if you pass control of the parallelism to a compiler you need that compiler to be good at it
- is very large and complicated in scope

OpenMP

OpenMP

- is an evolving standard
- is easy to use; you can modify existing programs incrementally
- hides messy threads fiddling
- needs compiler support, unlike pthreads (but is supported by the mainstream compilers, in particular GCC, Clang and MSVC)
- is dependent on good implementation of the compiler: if you pass control of the parallelism to a compiler you need that compiler to be good at it
- is very large and complicated in scope
- still allows trivially buggy programs

OpenMP

Exercise Would the coursework be easier using OpenMP?

Cilk Plus

Of course, OpenMP is not the only way of tweaking C

Cilk Plus

Of course, OpenMP is not the only way of tweaking C

Cilk Plus is somewhat similar in that it adds annotations and is based on fork and join

Cilk Plus

Of course, OpenMP is not the only way of tweaking C

Cilk Plus is somewhat similar in that it adds annotations and is based on fork and join

But as new keywords in C, not as pragmas (mostly)

Cilk Plus

Of course, OpenMP is not the only way of tweaking C

Cilk Plus is somewhat similar in that it adds annotations and is based on fork and join

But as new keywords in C, not as pragmas (mostly)

Cilk Plus is intended as an extension to C++, but works for C, too

Cilk Plus

Of course, OpenMP is not the only way of tweaking C

Cilk Plus is somewhat similar in that it adds annotations and is based on fork and join

But as new keywords in C, not as pragmas (mostly)

Cilk Plus is intended as an extension to C++, but works for C, too

You may come across other versions named “Cilk” and “Cilk++”

Cilk Plus

Of course, OpenMP is not the only way of tweaking C

Cilk Plus is somewhat similar in that it adds annotations and is based on fork and join

But as new keywords in C, not as pragmas (mostly)

Cilk Plus is intended as an extension to C++, but works for C, too

You may come across other versions named “Cilk” and “Cilk++”

We may have time to talk about Cilk later

Shared Memory

This concludes our discussion of the shared memory world

Shared Memory

This concludes our discussion of the shared memory world

For now

Distributed Memory

We now turn to distributed memory programming

Distributed Memory

We now turn to distributed memory programming

We could use interfaces like threads or OpenMP and have an underlying or virtualising infrastructure that converts them to message passing between processors over a network

Distributed Memory

We now turn to distributed memory programming

We could use interfaces like threads or OpenMP and have an underlying or virtualising infrastructure that converts them to message passing between processors over a network

Good programmers don't like that as it hides the source of the cost of distributed parallelism from the programmer, making it harder to design and write efficient programs

Distributed Memory

We now turn to distributed memory programming

We could use interfaces like threads or OpenMP and have an underlying or virtualising infrastructure that converts them to message passing between processors over a network

Good programmers don't like that as it hides the source of the cost of distributed parallelism from the programmer, making it harder to design and write efficient programs

So most distributed programs are explicitly message passing, or have some other way of making the cost of an operation more clear

Distributed Memory

The big player in this field is *Message Passing Interface* (MPI)

Distributed Memory

The big player in this field is *Message Passing Interface* (MPI)

You may hear about

- PVM: Parallel Virtual Machine, a predecessor to MPI
- SHMEM: SHared MEMory, only on Cray (SGI) machines
- UPC: Unified Parallel C, a supposed successor to MPI

MPI

MPI is what Big Science uses, when terabytes of data crunching is needed

MPI

MPI is what Big Science uses, when terabytes of data crunching is needed

And remember distributed systems are not good for small programs due to the overhead of the messaging outweighing the parallelism gained

MPI

MPI is what Big Science uses, when terabytes of data crunching is needed

And remember distributed systems are not good for small programs due to the overhead of the messaging outweighing the parallelism gained

MPI runs the same program on multiple processors (SPMD), but definitely not in lockstep

MPI

MPI is what Big Science uses, when terabytes of data crunching is needed

And remember distributed systems are not good for small programs due to the overhead of the messaging outweighing the parallelism gained

MPI runs the same program on multiple processors (SPMD), but definitely not in lockstep

The processes communicate via messages

MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

Code written to the standard should run on any implementation

MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

Code written to the standard should run on any implementation

But frequently doesn’t

MPI

MPI is “simply” a library of functions to do messaging; you can use it with normal (unmodified) C, Fortran, etc.

Even Java, Python and other languages less suited to high performance systems

MPI is actually a standard with several competing implementations

Code written to the standard should run on any implementation

But frequently doesn’t

The MPI standard specifies a huge number of functions, covering a wide range of different types of messaging

MPI

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

int main(int argc, char **argv)
{
  int rc, myrank, nproc, namelen;
  char name[MPI_MAX_PROCESSOR_NAME];

  rc = MPI_Init(&argc, &argv);
  if (rc != MPI_SUCCESS) {
     printf ("Error starting MPI program\n");
     MPI_Abort(MPI_COMM_WORLD, rc);
  }

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

continued

MPI

```
  if (myrank == 0) {
    printf("main reports %d procs\n", nproc);
  }

  namelen = MPI_MAX_PROCESSOR_NAME;
  MPI_Get_processor_name(name, &namelen);
  printf("hello world %d from '%s'\n", myrank, name);

  MPI_Finalize();
  return 0;
}
```

MPI

Notes:

MPI

Notes:

- `MPI_Init(&argc, &argv);` Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while `MPI_Init` sets up the connections between them

MPI

Notes:

- `MPI_Init(&argc, &argv);` Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while `MPI_Init` sets up the connections between them
- Later versions of MPI allow `MPI_Init(NULL, NULL)` but the above is preferable as it provides more information to the MPI system

MPI

Notes:

- `MPI_Init(&argc, &argv);` Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while `MPI_Init` sets up the connections between them
- Later versions of MPI allow `MPI_Init(NULL, NULL)` but the above is preferable as it provides more information to the MPI system
- `rc` Always check to make sure it worked

MPI

Notes:

- `MPI_Init(&argc, &argv);` Set up the system: you must always do this. A batch processing system (e.g., SLURM) starts the processes on all the processors, while `MPI_Init` sets up the connections between them
- Later versions of MPI allow `MPI_Init(NULL, NULL)` but the above is preferable as it provides more information to the MPI system
- `rc` Always check to make sure it worked
- `MPI_COMM_WORLD` The system can be sub-divided into subsets of processors called *communicators*. The `WORLD` communicator is all processors; `MPI_COMM_SELF` refers to just the calling processor

MPI

- `MPI_Comm_rank` Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* -1. So, for `WORLD` the rank ranges from 0 to *total number of processors* -1

MPI

- `MPI_Comm_rank` Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* -1. So, for `WORLD` the rank ranges from 0 to *total number of processors* -1
- `MPI_Comm_size` Get the size of the communicator

MPI

- `MPI_Comm_rank` Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* -1. So, for `WORLD` the rank ranges from 0 to *total number of processors* -1
- `MPI_Comm_size` Get the size of the communicator
- `if (myrank == 0)` All processors run the same code (SPMD). This is how we get different things happening on different processors

MPI

- `MPI_Comm_rank` Each process in a communicator has a unique rank within that communicator: this is just an integer from 0 to *size of the communicator* -1. So, for `WORLD` the rank ranges from 0 to *total number of processors* -1
- `MPI_Comm_size` Get the size of the communicator
- `if (myrank == 0)` All processors run the same code (SPMD). This is how we get different things happening on different processors
- `MPI_Finalize` All procs must always call this to tidy up their MPI state

MPI

Compile using `mpicc`:

```
mpicc -Wall -o hellompi hellompi.c
```

MPI

Batch file `runnit.slm`:

```
#!/bin/sh
#SBATCH --account=cm30225
#SBATCH --partition=teaching
#SBATCH --job-name=HelloMPI
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8

mpirun ./hellompi
```

MPI

The lines of note here are:

MPI

The lines of note here are:

- `--nodes=2` we want two nodes

MPI

The lines of note here are:

- `--nodes=2` we want two nodes
- `--ntasks-per-node=8` we will be using just 8 of the 44 cores on each node

Recall we had:

```
if (myrank == 0) {
  printf("main reports %d procs\n", nproc);
}

namelen = MPI_MAX_PROCESSOR_NAME;
MPI_Get_processor_name(name, &namelen);
printf("hello world %d from '%s'\n", myrank, name);
```

MPI

Output:

```
hello world 3 from 'ip-AC125409'
hello world 5 from 'ip-AC125409'
hello world 4 from 'ip-AC125409'
hello world 11 from 'ip-AC125408'
hello world 6 from 'ip-AC125409'
hello world 9 from 'ip-AC125408'
hello world 1 from 'ip-AC125409'
hello world 15 from 'ip-AC125408'
hello world 7 from 'ip-AC125409'
hello world 12 from 'ip-AC125408'
hello world 2 from 'ip-AC125409'
hello world 10 from 'ip-AC125408'
main reports 16 procs
hello world 0 from 'ip-AC125409'
hello world 14 from 'ip-AC125408'
hello world 13 from 'ip-AC125408'
hello world 8 from 'ip-AC125408'
```

MPI

Notes:

MPI

Notes:

- `ip-AC125408` and `ip-AC125409` are the names of the two nodes that happened to be allocated; the next run may well get different nodes

MPI

Notes:

- `ip-AC125408` and `ip-AC125409` are the names of the two nodes that happened to be allocated; the next run may well get different nodes
- Processes 0–8 are on `ip-AC125409` while processes 9-15 are on `ip-AC125408`, but it might happen the other way around

MPI

Notes:

- `ip-AC125408` and `ip-AC125409` are the names of the two nodes that happened to be allocated; the next run may well get different nodes
- Processes 0–8 are on `ip-AC125409` while processes 9-15 are on `ip-AC125408`, but it might happen the other way around
- `ntasks-per-node` is important here as sometimes you want fewer MPI tasks on a node than there are cores on that node: an MPI task can itself be multithreaded (not your coursework!)

MPI

- Output in a random order, even for the “main reports 16 procs” which we might think happens first!

MPI

- Output in a random order, even for the “main reports 16 procs” which we might think happens first!
- We *do* see “main reports” before “hello world 0”, though!

MPI

- Output in a random order, even for the “main reports 16 procs” which we might think happens first!
- We *do* see “main reports” before “hello world 0”, though!
- MPI has a mechanism for routing prints on any node back via the network to a single point: this results in all kinds of timing variations in output

MPI

- MPI is SPMD, so this code is *not* synchronised across processors

MPI

- MPI is SPMD, so this code is *not* synchronised across processors
- For example, when proc 0 is doing its `printf` the other processors may well already be doing `MPI_Get_processor_name`

MPI

- MPI is SPMD, so this code is *not* synchronised across processors
- For example, when proc 0 is doing its `printf` the other processors may well already be doing `MPI_Get_processor_name`
- Or perhaps still `MPI_Comm_size`

MPI

- MPI is SPMD, so this code is *not* synchronised across processors
- For example, when proc 0 is doing its `printf` the other processors may well already be doing `MPI_Get_processor_name`
- Or perhaps still `MPI_Comm_size`
- But many MPI function calls do have a built-in synchronisation and block the calling processor until all processors involved in that call are done

MPI

- MPI is SPMD, so this code is *not* synchronised across processors
- For example, when proc 0 is doing its `printf` the other processors may well already be doing `MPI_Get_processor_name`
- Or perhaps still `MPI_Comm_size`
- But many MPI function calls do have a built-in synchronisation and block the calling processor until all processors involved in that call are done
- Each MPI "task" is a separate *process*, not sharing anything with any other task: in particular, not sharing any variables (e.g., `myrank`), even if the tasks happen to be on the same node

MPI

Exercise Does adding a `MPI_Barrier` after the “main reports” conditional *ensure* the message comes out first?