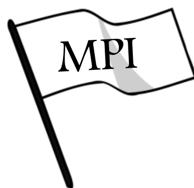
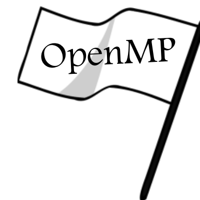
EEE4120F

High Performance Embedded Systems

Lecture 12b:
Distributed Memory Systems &
MPI vs. OpenMP





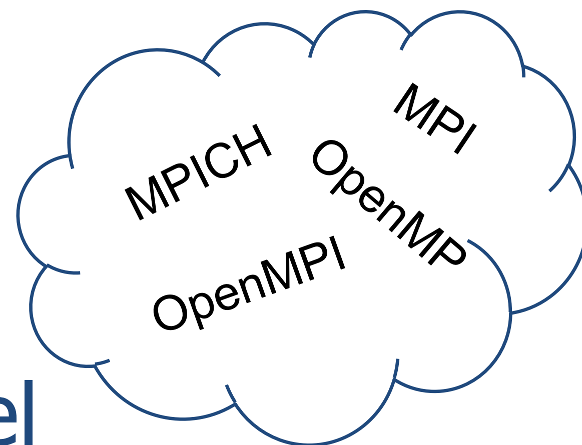
Lecturer:
Simon Winberg

An R+J theme to help memory
+ an observation of 8 March
International Womans Day special

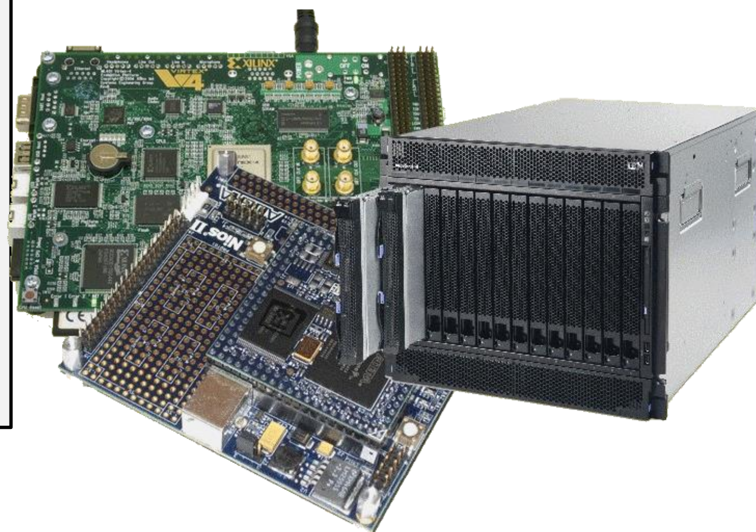
Trimmed down version with fewer mages

public share

Lecture Overview



- Data parallel
- Message Passing (MP) model
- Message passing implementations
- Two ‘MP’ households
  - Intro to MPI & OpenMPI
  - Intro to OpenMP
- MPICH



Data parallel characteristics

- For *shared memory* architectures, all tasks often have access to the same data structure through *global memory*.
  (you probably did this with Pthreads already!)
- For *distributed memory* architectures the data structure is split up and resides as 'chunks' in local task/machines
  (MPI and OpenMPI programs tends to use this method together with messages indicating which data to use)

Message Passing (MP) model

- Involves a set of tasks that use their own local memory during computations
- Multiple tasks can run on the same physical machine, or they could run across multiple machines
- Tasks exchange data by sending and receiving communication messages
- Transfer of data usually needs cooperative or synchronization operations to be performed by each process, i.e., each send operation needs a corresponding receive operation.

Message Passing (MP) model

- Parallelized software applications using MP usually make use of a specialized library of subroutines, linked with the application code.
- Programmer is entirely responsible for deciding and implementing all the parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.

But let’s look at the more exciting aspect of today’s typical alternatives that tends to be considering if one should go for:

MPI (or an open source alternative)

or

OpenMP

They sound similar… but the approaches are actually quite dissimilar!

The 'MP' households : determinately different

EEE4120F

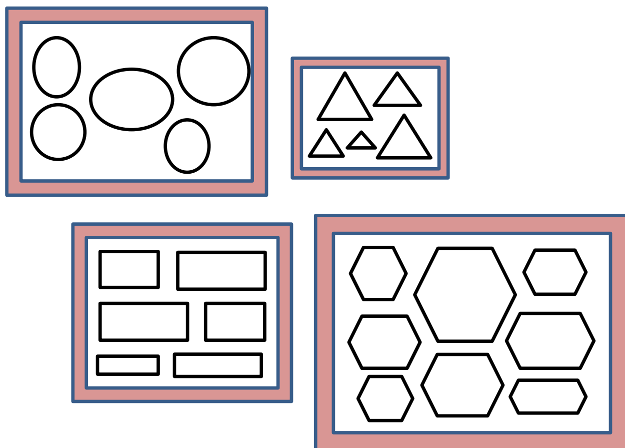
Two households, both alike in dignity… but beware to anyone who moves from one family to the other as much contestation may befall any doing so unprepared…

The big 'MP' families

MPI

MPI =
Message Passing Interface

- Standard language (e.g. pure C) no embellishments
- Designed around heterogeneous clusters



OpenMP

OpenMP =
Open MultiProcessing

- Language embellishments / Pragmas galore
- More for one computer; homogeneous architecture

Indeed my slides are set up as a bit of a play on the Romeo and Juliette, where the Montagues (MPIs) and Capulets (OpenMP) are 'alike in dignity' but are opposing force to one another ... the two 'houses' of thought have quite different paradigms.

Basically the decision about which to use depends what you want to do and the system architecture you have, whether to use OpenMP or MPI (or Open MPI)

let's

BRING IT ON!

MPI

OpenMP

Starting with the Montagues … I mean MPI

MPI – the first (big) 'MP' family

- MPI = Message Passing Interface
- As you may have expected:
  - Yes, this is actually designed for application to distributed computing, one PC speaking to another, setting up servers and the like
  - You can have multiple instances of an MPI program running on one machine (e.g. processes talking to each other via MPI)
- MPI is a standardized API with a library provided.

Message passing implementations

- The MPI Forum was formed in 1992, with the goal of establishing a standard interface for message passing implementations, called the "Message Passing Interface (MPI)" – first released in 1994

Message passing implementations

- MPI is now the most common industry standard for message passing, and has replaced many of the custom-developed and obsolete standards used in legacy systems.
- Most manufacturers of (microprocessor-based) parallel computing platforms offer an implementation of MPI.
- For shared memory architectures, MPI implementations usually do not use the network for inter-task communications, but rather use shared memory (or memory copies) for better performance. Or a combination and shared and distributed memory, i.e. transfers over network if messages are sent to nodes on other platforms.

Using MPI or Open MPI on heterogeneous clusters

- MPI has a significant advantage (over its competitor OpenMP) in that it accommodates heterogeneous computing systems or clusters
- The processors that are involves in running an MPI program do not have to be the same type or be compiled by the same compiler.
- The coordination is through messages which abstracts away the mechanisms of processing.

Short prelude to MPI

```
/*******************************************
   Hello MPI - a simple starting point for
   MPI programmes
*********************************************/

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

/*********************************************/

int main(int argc, char** argv) {
    // Initialize MPI environment
    MPI_Init(NULL, NULL);

    // Determine number of processes started
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of this processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Display 'hello world' message and which processor
    // num this is out of the total num available processors.
    printf("Hello world from node %s, rank %d"
           " out of %d nodes\n",
           processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Using these few fundamental MPI commands you can quickly start developing distributable programs, especially e.g. if you use it in combination with a shared file system (e.g. NFS).
While you could do the message passing just with TCP/IP sockets it is recommendable to use the provided message passing functions instead.

Let's quickly run through these basic MPI commands…

(there will be a prac on MPI where you can get more into the use of the API)

Short prelude to MPI

```
/*******************************************
   Hello MPI - a simple starting point for
   MPI programmes
********************************************/

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

/********************************************/

int main(int argc, char** argv) {
    // Initialize MPI environment
    MPI_Init(NULL, NULL);

    // Determine number of processes started
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of this processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Display 'hello world' message and which processor
    // num this is out of the total num available processors.
    printf("Hello world from node %s, rank %d"
           " out of %d nodes\n",
           processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

```
MPI_Init(NULL, NULL);
```

Initializes the MPI environment. The two parameters are technically two pointers, which could pass argc and argv from main(). It is mainly for future use and generally isn’t used. This command does various things, waits for master to give the go-ahead to start running.

Short prelude to MPI

```
/*********************************************
   Hello MPI - a simple starting point for
   MPI programmes
*********************************************/

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

/*********************************************/

int main(int argc, char** argv) {
    // Initialize MPI environment
    MPI_Init(NULL, NULL);

    // Determine number of processes started
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of this processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Display 'hello world' message and which processor
    // num this is out of the total num available processors.
    printf("Hello world from node %s, rank %d"
           " out of %d nodes\n",
           processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

```
int world_size;
MPI_Comm_size(MPI_COMM_WORLD,
                    &world_size);
```

This gets the total number of nodes that are to be started. When you start an MPI program you tell it how many nodes you want to spawn.

Short prelude to MPI

```
/*******************************************
   Hello MPI - a simple starting point for
   MPI programmes
********************************************/

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

/********************************************/

int main(int argc, char** argv) {
    // Initialize MPI environment
    MPI_Init(NULL, NULL);

    // Determine number of processes started
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of this processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Display 'hello world' message and which processor
    // num this is out of the total num available processors.
    printf("Hello world from node %s, rank %d"
           " out of %d nodes\n",
           processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

```
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD,
              &world_rank);
```

This gets the nodes number for this instance, and is 0 for the first or master node. This is clearly used to determine which instance is which.

Short prelude to MPI

```
/********************************************
   Hello MPI - a simple starting point for
   MPI programmes
********************************************/

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

/********************************************/

int main(int argc, char** argv) {
    // Initialize MPI environment
    MPI_Init(NULL, NULL);

    // Determine number of processes started
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of this processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Display 'hello world' message and which processor
    // num this is out of the total num available processors.
    printf("Hello world from node %s, rank %d"
           " out of %d nodes\n",
           processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

```
MPI_Get_processor_name(
          processor_name,
          &name_len);
```

This gets the full name of the node, which can be useful (e.g. for debugging) to determine which machine it is running on.

Short prelude to MPI

```
/*********************************************
   Hello MPI - a simple starting point for
   MPI programmes
*********************************************/

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

/*********************************************/

int main(int argc, char** argv) {
    // Initialize MPI environment
    MPI_Init(NULL, NULL);

    // Determine number of processes started
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of this processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Display 'hello world' message and which processor
    // num this is out of the total num available processors.
    printf("Hello world from node %s, rank %d"
           " out of %d nodes\n",
           processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

```
printf("Hello world …\n",…);
```

You can use printf in an MPI program to display info and is useful for debugging. The stdout is connected to the master (thread process that started the MPI program). The MPI standard doesn't stipulate in which order printouts from nodes should appear – generally they displayed soon as possibly. So if node 2 were to start sooner than node 1 the output might show node 2's greeting before the node 1 greeting.

Short prelude to MPI

```
/*******************************************
   Hello MPI - a simple starting point for
   MPI programmes
********************************************/

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

/********************************************/

int main(int argc, char** argv) {
    // Initialize MPI environment
    MPI_Init(NULL, NULL);

    // Determine number of processes started
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of this process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of this processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Display 'hello world' message and which processor
    // num this is out of the total num available processors.
    printf("Hello world from node %s, rank %d"
           " out of %d nodes\n",
           processor_name, world_rank, world_size);
    // Finalize the MPI environment.
    MPI_Finalize();
}
```

```
MPI_Finalize();
```

This closes the MPI application, essentially joins the threads and shuts down each thread running on the various node elegantly. Also flushes IO buffers). Master (node 0) shuts down last.

Compiling & running?... usually:

```
mpicc hellompi.c -o hellompi
mpirun -np 4 hellompi
```

(starts total of 4 processes; note cannot be guaranteed it will start given number of processes, you need to check how many started in your code and report errors if needed.)

Compiling and Running MPI

- To compile an MPI program use:
  - mpicc -o exename codename.c
  - This compiles codename.c to generate executable file named exename
- To run an MPI program use:
  - mpirun ./exename # uses default num nodes
  - mpirun -n 3 ./exename # uses 3 nodes
- mpicc is essentially a wrapper around a standard C compiler like gcc. It's not a completely separate, specialized compiler in itself.
- More info/tips at:

  http://wiki.ee.uct.ac.za/Message_Passing_Interface_(MPI)

Now you're prepared and ready to take on Prac3 !

Which is about MPI

OpenMP – the other 'family' with 'MP' in its name

NOTE: OpenMP coding not included in exam curriculum

- OpenMP = Open MultiProcessing
- It is not (really) planned around being a message passing framework!
- OpenMP is a shared memory system
- It needs to be compiled using a specialized compiler

A nice thing (and an major purpose of OpenMP) is that it does much of the parallelization for you. It provides pragmas (compiler directives) to tell the compiler how to treat certain blocks of code. It has much scope for increasing development speed without the developer necessarily needing to know much about developing parallel software.

OpenMP – simple example

NOTE: OpenMP coding not included in exam curriculum

```
/*******************************************************
 OpenMP Example
 *******************************************************/

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

/*******************************************************/

int main (int argc, char *argv[])
{
 int nthreads, tid;
 /* Fork a team of threads giving them their own copies of variables */
 #pragma omp parallel private(nthreads, tid)
 { /* >>> the thread starts here... */

   /* Obtain thread number */
   tid = omp_get_thread_num();
   printf("Hello World from thread = %d\n", tid);

   /* Only master thread does this */
   if (tid == 0)
     {
     nthreads = omp_get_num_threads();
     printf("Number of threads = %d\n", nthreads);
     }

     /* the thread ends here... <<<< */
   } /* The threads all join at this point */

}
```

The OpenMP code may look a whole lot nicer and easier. And it has many powerful features that can produce very efficient code and at the same time save a lot of time in coding. But there can be quite a lot of documentation and coding practice to get through to do so.

MPI vs. OpenMP

MPI	OpenMP
Defines an API, vendors provide an optimized (usually binary) library implementation that is linked in using your choice of compiler.	OpenMP has its own compiler or is hooked in to a compiler (e.g. gcc). User not have much option in changing compiler and OS unless a OpenMP compiler exists for platform to use.
Has support for C, C++, Fortran, many others. (Not so difficult to port, just need to at least develop a wrapper API interface for a pre-compiled MPI implementation in a different language)	Has support for C, C++ and Fortran but probably not many other options.
Can be used for distributed and shared memory (e.g. SMP) systems	Can only be used for shared memory system.
Processes and thread based parallelism.	Only thread based parallelism.
Potentially high cost in creating process/thread instance and comms	Lower overheads, very little as inter-process comms done by shared mem

OpenMPI vs. OpenMP ?

- OpenMPI is simply an open implementation of the MPI framework
- OpenMP is not OpenMPI

MPICH

- MPICH is probably the most commonly used form of MPI
- Initially developed (and many still maintained) by US governmental organisations
- The standard (and many versions are) public domain/free software
- What is the relevant of ‘CH’ in the name?...

MPICH

- Historical aspect of MPICH
  - Original version (MPICH-1) developed (starting in 1992) by the Argonne National Laboratory as public domain software.
  - The 'CH' part stands for "Chameleon", referring to the portable parallel programming library incorporated into MPICH (Chameleon was developed by William Gropp, a MPICH founder)
- Good support and lots of documentation available
  - See: https://www.mpich.org/documentation/guides/

Sources and more info: https://www.mpich.org/, https://en.wikipedia.org/wiki/MPICH

Conclusion of the MPI vs OpenMP

- So, it is commonly a confusion of acronyms… one that you're now unlikely to forget!

The metaphors aside, it may nevertheless be advisable to try some OpenMP programming as well as some MPI, especially if you are planning a career in parallel computing, as they have dissimilar benefits and drawbacks – and indeed it does happen that some people find the one approach preferably to the other.

Combining OpenMP and MPI ?!

Why not have the best of both?!

Combining OpenMP and MPI ?!

- Parts of your processing could use OpenMP, using shared memory without copies of data structures
- Designs that overlap computation and communication
- Groups of co-located processors on different machines working together on a common task
- Could have combinations like 5x2 where you have 5 groups (5 OpenMP instances on different machines) of two processors (the groups communicating via MPI)

Optional Extensions to Prac3

- Setting up the prac on *multiple* machines (you might combine forces with another prac group for this part – you want to have the image files tested copied to all PCs)
- Combine MPI and OpenMP to see how much better a hybrid MPI+OpenMP system could work compared to standard MPI on a cluster

Suggested further reading

- Have a look at MPI examples, e.g.
  - http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml
- Installing MPI on WSL:
  - https://tutorialsheet.com/mpi-programming/how-to-install-and-setup-mpi-environment-in-wsl/*

* WSL = Windows Subsystem for Linux, suggest installing Ubuntu on it (works pretty well!!)

End of lecture

Ready for Prac3 !!!