Parallel Programming

Libraries and Implementations

Reusing this material

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

http://www.archer.ac.uk
support@archer.ac.uk

Outline

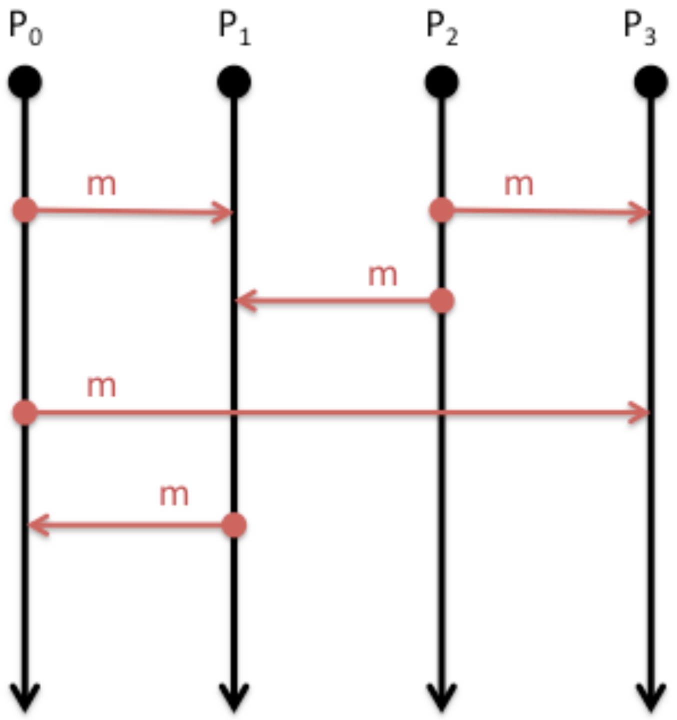
- MPI – de facto standard for distributed memory programming
- OpenMP – de facto standard for shared memory programming
- CUDA – dominant GPGPU programming model & libraries
- Other Approaches
  - PGAS
  - SHMEM

MPI

Distributed memory parallelism using message passing

Message-passing concepts

- Processes can not access each other’s memory spaces
  - Variables are private to each process
  - Processes communicate data by passing messages



What is MPI?

- MPI = Message Passing Interface
- MPI is not a programming *language*
  - There is *no such thing* as an MPI compiler
- MPI is available as a *library* of function/subroutine calls
  - Library implements a communications protocol
  - Follows an agreed-upon standard (see next slide)
- The C or Fortran compiler you invoke knows nothing about what MPI actually does
  - only knows prototype/interface of the function/subroutine calls

The MPI standard

- MPI is a standard
- Agreed upon through extensive joint effort of ~100 representatives from ~40 different organisations (the MPI Forum)
  - Academics
  - Industry experts
  - Vendors
  - Application developers
  - Users
- First version (MPI 1.0) drafted in 1993
- Now on version 3 (version 4 being drafted)

MPI Libraries

- The MPI Forum defines the standard, vendors / open-source developers create libraries that actually implement versions of the standard

- There are a number of different implementations but all should support the MPI standard (version 2 or 3)
  - As with different compilers there will be variations in implementation details but all the features specified in the standard should work.
  - Examples: MPICH2, OpenMPI
  - Cray-MPICH on ARCHER (optimised for interconnect on Cray machines)

Features of MPI

- MPI is a portable library used for writing parallel programs using the message passing model
  - You can expect MPI to be available on any HPC platform you use

- Based on a number of *processes* running independently in parallel
  - HPC resource provides a command to launch multiple processes simultaneously (*e.g.* mpiexec, aprun)
  - Can think of each process as an instance of your executable communicating with other instances

Explicit Parallelism

- In message-passing all the parallelism is explicit
  - The program includes specific instructions for each communication
  - What to send or receive
  - When to send or receive
  - Synchronisation
- It is up to the developer to design the parallel decomposition and implement it
  - How will you divide up the problem?
  - When will you need to communicate between processes?

Point-to-point communications

- A message sent by one process and received by another
- Both processes are actively involved in the communication – not necessarily at the same time
- Wide variety of semantics provided:
  - Blocking vs. non-blocking
  - Ready vs. synchronous vs. buffered
  - Tags, communicators, wild-cards
  - Built-in and custom data-types
- Can be used to implement any communication pattern
  - Collective operations, if applicable, can be more efficient

Collective communications

- A communication that involves all processes
  - “all” within a communicator, i.e. a defined sub-set of all processes
- Each collective operation implements a particular communication pattern
  - Easier to program than lots of point-to-point messages
  - Should be more efficient than lots of point-to-point messages
- Commonly used examples:
  - Broadcast
  - Gather
  - Reduce
  - AllToAll

Example: MPI HelloWorld

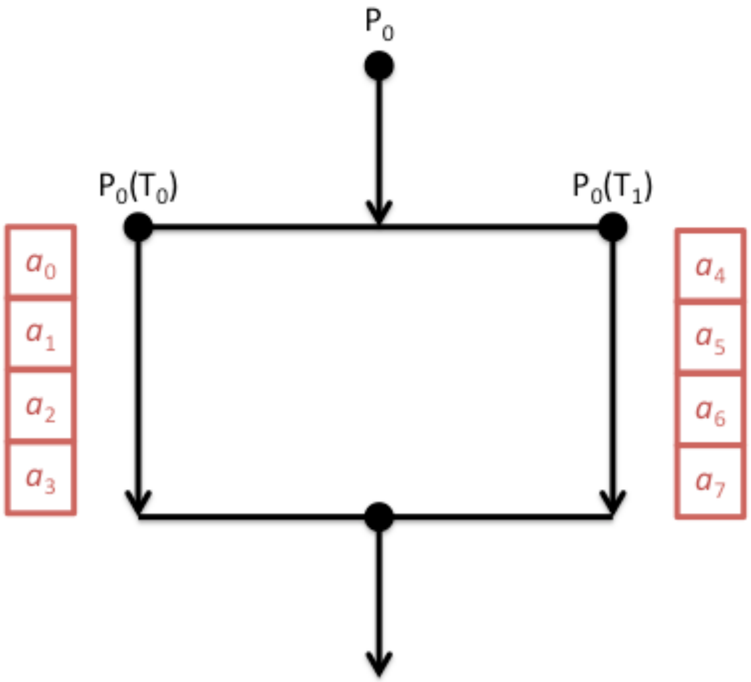
```c
#include <mpi.h>

int main(int argc, char* argv[])
{
   int size,rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("Hello world - I'm rank %d of %d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

OpenMP

Shared-memory parallelism using directives

Shared-memory concepts

- Threads “communicate” by having access to the same memory space
  - Any thread can alter any bit of data
  - No explicit communications between the parallel tasks



OpenMP

- OpenMP = “Open Multi Processing”
  - Application Program Interface (API) for shared memory programming
- OpenMP is a set of extensions to Fortran, C and C++:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Not a library interface, unlike MPI
- A directive is a special line of source code with meaning only to certain compilers thanks to keywords (*sentinels*)
  - Directives are ignored if code is compiled as regular sequential Fortran/C/C++
- OpenMP is also a standard (see http://openmp.org/)

Features of OpenMP

- Directives define parallel regions in code within which OpenMP threads divide work done in the region
  - Should decide which variables are private to each thread or shared
- The compiler needs to know what OpenMP actually does
  - It is responsible for producing the OpenMP-parallel code
  - OpenMP supported by all common compilers used in HPC
  - Compilers should implement the standard
- Parallelism is less explicit than for MPI
  - You specify which parts of the program you want to parallelise and the compiler produces a parallel executable
- Also used for programming Intel Xeon Phi

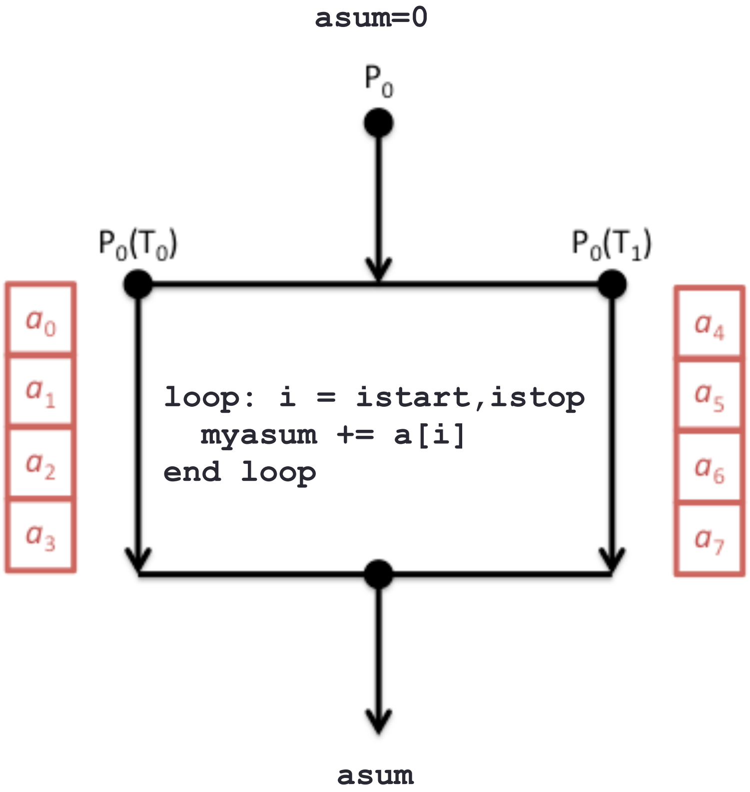
Loop-based parallelism

- A very common form of OpenMP parallelism is to parallelise the work in a loop
  - The OpenMP directives tell the compiler to divide the iterations of the loop between the threads

```
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
   #pragma omp for schedule(dynamic,chunk) nowait
   for (i=0; i < N; i++) {
     c[i] = a[i] + b[i];
   }
}
```

Addition example

```
asum = 0.0
#pragma omp parallel \
shared(a,N) private(i) \
reduction(+:asum)
{
   #pragma omp for
   for (i=0; i < N; i++)
   {
     asum += a[i];
   }
}
printf("asum = %f\n", asum);
```



CUDA

Programming GPGPU Accelerators

CUDA

- CUDA is an Application Program Interface (API) for programming NVIDIA GPU accelerators
  - Proprietary software provided by NVIDIA. Should be available on all systems with NVIDIA GPU accelerators
  - Write GPU specific functions called *kernels*
  - Launch kernels using syntax within standard C programs
  - Includes functions to shift data between CPU and GPU memory
- Similar to OpenMP programming in many ways in that the parallelism is implicit in the kernel design and launch
- More recent versions of CUDA include ways to communicate directly between multiple GPU accelerators (*GPUdirect*)

Example:

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

// Called with
vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
```

OpenCL

- An open, cross-platform standard for programming accelerators
  - includes GPUs, e.g. from both NVIDIA and AMD
  - also Xeon Phi, Digital Signal Processors, ...

- Comprises a language + library

- Harder to write than CUDA if you have NVIDIA GPUs
  - but portable across multiple platforms
  - although maintaining performance is difficult

Others

Niche and future implementations

Other parallel implementations

- Partitioned Global Address Space (PGAS)
  - Coarray Fortran, Unified Parallel C, Chapel
- Cray SHMEM, OpenSHMEM
  - Single-sided communication library
- OpenACC
  - Directive-based approach for programming accelerators

Summary

Parallel Implementations

- Distributed memory programmed using MPI
- Shared memory programmed using OpenMP
- GPU accelerators most often programmed using CUDA

- Hybrid programming approaches very common in HPC, especially MPI + X (where X is usually OpenMP)
  - Hybrid approaches matches the hardware layout more closely

- A number of other, more experimental approaches are available