Parallel Programming

Libraries and implementations

Reusing this material

Outline

- How we manage software packages & libraries
- MPI – distributed memory de-facto standard
  - Using MPI
- OpenMP – shared memory de-facto standard
  - Using OpenMP
- Other parallel programming technologies
  - CUDA, OpenCL, OpenACC
- Examples of common scientific libraries

The module environment

Managing software packages and libraries

(these examples come from ARCHER but Cirrus is very similar)

Module environment

```
user@eslogin001:~> module list
Currently Loaded Modulefiles:
  1) modules/3.2.10.2                      9) rca/1.0.0-2.0502.57212.ari
  2) eswrap/1.3.3-1.020200.1278.0         10) atp/1.8.3
  3) switch/1.0-1.0502.57058.1.58.ari     11) PrgE56
  4) craype-network-aries                 12) pbs/12.2.401.141761
  5) craype/2.4.2                         13) craype-ivybridge
  6) cce/8.4.1                            14) cray-mpich/7.2.6
  7) cray-libsci/13.2.0                   15) packages-archer
  8) udreg/2.3.2-1.0502.9889.2.20.ari     16) bolt/0.6
```

- The module environment allows you to easily load different packages and manage different versions of packages.
- Via the *module* command
  - *List loaded modules, view available modules, load and unload modules*

Using the module environment

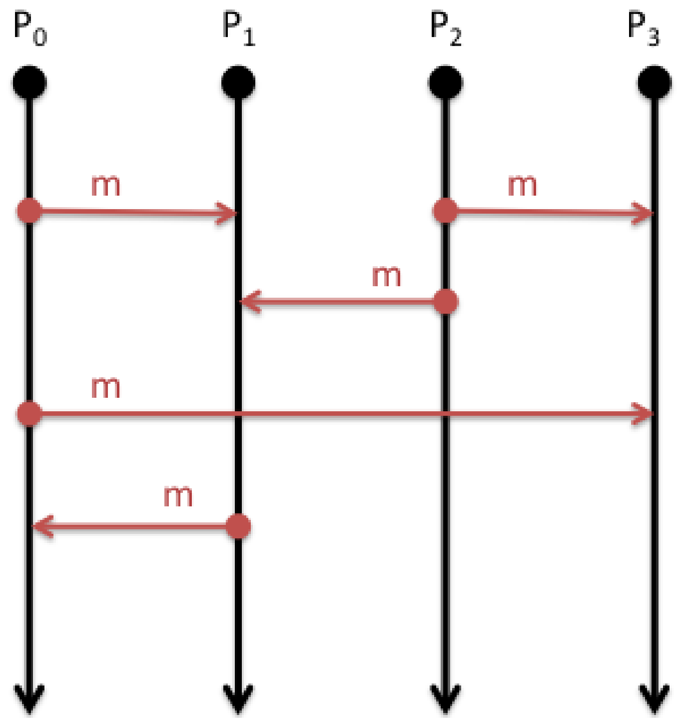
```
user@eslogin001:~> module avail
PrgEnv-cray/5.1.29          PrgEnv-cray/5.2.56(default)          PrgEnv-gnu/5.1.29
PrgEnv-intel/5.1.29         PrgEnv-intel/5.2.56(default)cray-mpich/6.3.1
cray-mpich/7.1.1                        cray-mpich/7.2.6(default)               cray-mpich/7.3.2
cray-netcdf/4.3.3(default)  cray-netcdf/4.4.1                                   cray-petsc/3.5.2.1
cray-petsc/3.6.3.0                      cray-petsc/3.6.1.0 (default)            cray-petsc/3.7.2.0
fftw/2.1.5.7                fftw/2.1.5.9                         fftw/3.3.4.5(default)
fftw/3.3.4.7                fftw/3.3.4.9
```

```
user@eslogin001:~> module load fftw
user@eslogin001:~> module unload fftw

user@eslogin001:~> module load fftw/2.1.5.7
user@eslogin001:~> module switch fftw/2.1.5.7 fftw/3.3.4.9

user@eslogin001:~> module swap PrgEnv-cray PrgEnv-gnu
```

MPI Library

Distributed, message-passing programming

Message-passing concepts



What is 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
  - The library implements the MPI standard
- The C or Fortran compiler knows nothing about what MPI actually does
  - Just the prototype/interfaces of the functions/subroutine
  - It is just another library

The MPI standard

- MPI itself is a standard
- Agreed upon by approx 100 representatives from about 40 organisations (the MPI forum)
  - Academics
  - Industry
  - Vendors
  - Application developers

- First standard (MPI version 1.0) drafted in 1993
  - We are currently on version 3
  - Version 4 is being drafted

MPI Libraries

- The MPI forum defines the standard and vendors/open source developers then actually implement this

- There are a number of different implementations but all should support version 2.0 or 3.0
  - As with compilers there are variations in implementation details but all features in the standards should work
  - Examples: MPICH and OpenMPI
  - Cray-MPICH on ARCHER which implements version 3.1 of the standard (optimised for Cray machines, specifically the interconnect)

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
  - Aids portability between HPC machines and is trivial to install on local clusters

- Based on a number of processes running independently in parallel
  - The HPC resource provides the command to launch the processes in parallel (i.e. *aprun* or *mpiexec*)
  - 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?

Supported features

- Point to point communications
  - Communications involving two processes; a sender and receiver
  - Wide variety of semantics involving non-blocking communications
  - Other aspects such as wildcards & custom data types

- Collective communications
  - Communication that involves many processes
  - Implements all the collective communications we saw in the programming models lecture and many more
  - Also supports non-blocking communications and custom data types

Example: MPI HelloWorld

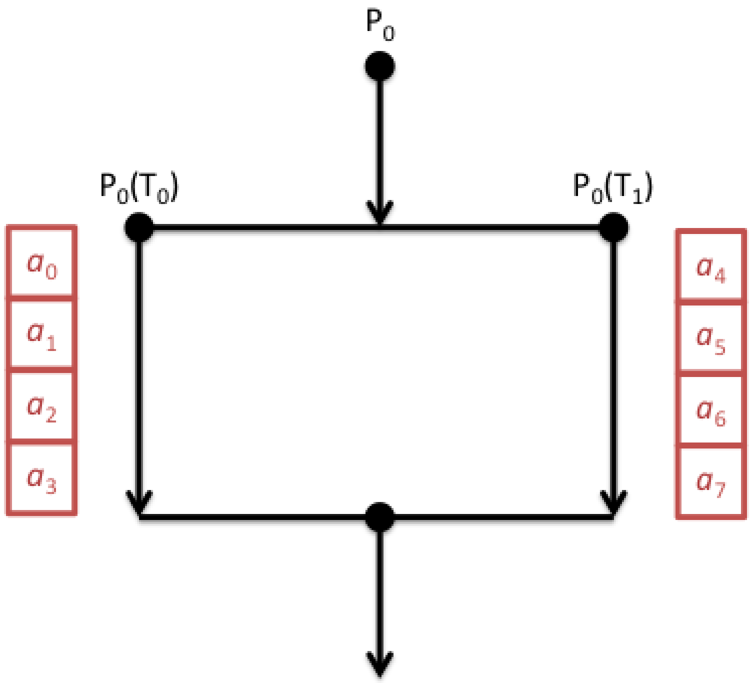
```
#include "mpi.h"

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

- Open Multi Processing
  - Application programming interface (API) for shared variable programming
- Set of extensions to C, C++ and Fortran
  - Compiler directives
  - Runtime library functions
  - Environment variables
- Not a library interface like MPI
- Uses directives, which are a special line in the source code with a meaning understood by the compilers
  - Ignored if OpenMP is disabled and it becomes regular sequential code
- This is also a standard (http://openmp.org)

Features of OpenMP

- Directives define parallel regions in the code
  - OpenMP threads are active in these regions and divide the workload amongst themselves
- The compiler needs to understand what OpenMP does
  - It is responsible for producing the parallel code
  - OpenMP supported by all common compilers used in HPC
- Parallelism less explicit than MPI
  - You just specify what parts of the program you want to run in parallel
- OpenMP version 4.5 is the latest version

- Can be used to program the Xeon Phi

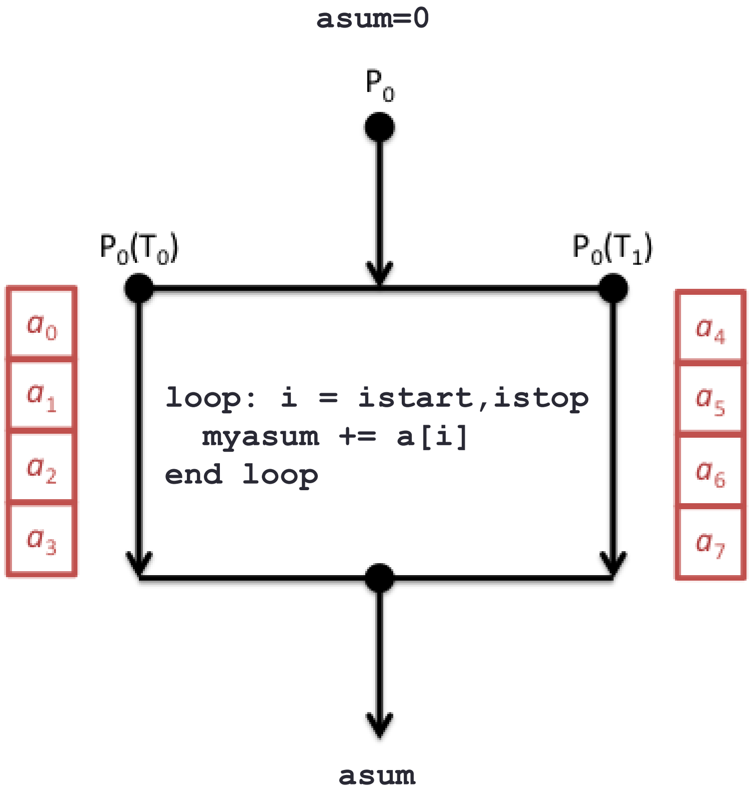
Loop-based parallelism

- The most 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) private(i)
{
   #pragma omp for schedule(dynamic) 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);
```



Other parallel programming technologies

Programming accelerators and less common technologies

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

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

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

Directives for Accelerators

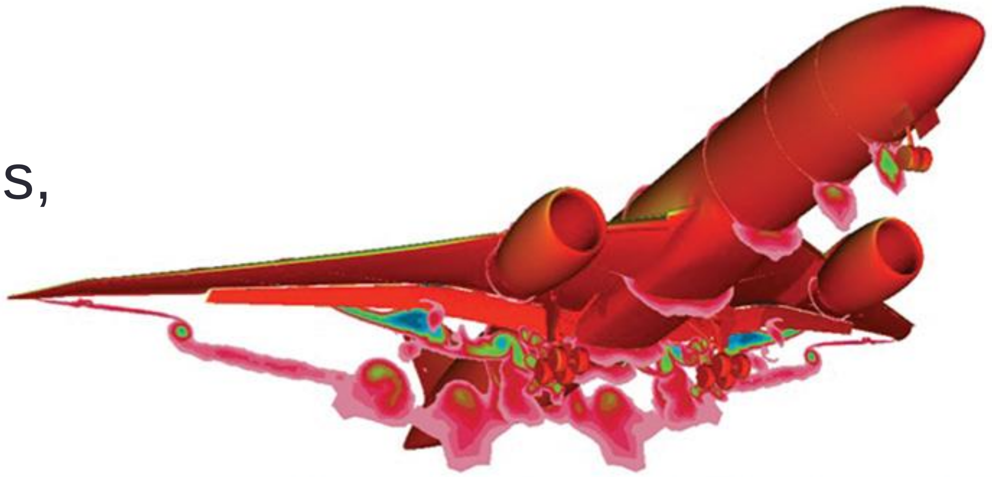
Common scientific parallel libraries

Two examples commonly used on HPC machines

PETSc

- Portable Extensible Toolkit for Scientific Computation
  - Suite of data structures & routines for the parallel and scalable solution of PDEs
  - The programmer uses the library framework itself which under the hood will use parallel technologies MPI, OpenMP and/or CUDA.

- Unlike many serial libraries, you the programmer are responsible for performance & scalability.



NetCDF

- Network Common Data Form
  - Self describing, machine independent file data format and implementation that is very common for writing and reading scientific data
- Parallel version supporting parallel IO
  - Multiple processes/threads can read and write to a file concurrently
  - Built on top of MPI
- Many third party tools such as visualisation suites

- Again requires user understanding, both from the programmer and also the user (file configuration options)

Summary

Parallel and scientific libraries

- The module environment is an easy way of managing many different software packages, their dependencies and different versions.

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

- There are very many software packages installed on Cirrus, but scientific libraries often require in-depth knowledge and understanding to get good performance.