WORKSHOP: Introduction to High-Performance Computing (HPC)

Paul Coddington, Deputy Director eRSA

9 March 2016

Introduction to HPC

- Overview of High-Performance Computing
- Different ways of using HPC
- Porting application programs to use HPC
- Shared memory vs distributed memory
- GPUs
- Speedup

Supercomputers

- Modern supercomputers are parallel (multi-processor) computers with hundreds or thousands of processors.
- Usually commodity processors – similar to those in a desktop PC.
- Usually commodity compute servers connected by fast networks (10Gbit Ethernet, Infiniband)
- This is a change from early custom-built supercomputers

Supercomputers

- Increase in speed of supercomputers over desktop computers is from using multiple CPUs at once, not from faster CPUs.
- This approach is now moving to desktop PCs with multicore processors.
- Servers and HPC moving to “manycore” processors
- So to gain benefit from supercomputers requires getting your application to run on multiple processors – *parallel computing*
- No free lunch!

Parallel Computing

- Two basic options for efficient parallel computing.
- **Reduce completion time of a single run**
  - Speed up the execution time of a single program run by dividing up the computation among the processors.
  - *High-performance computing*.*
  - Need to modify (parallelize) the program.
- **Reduce total completion time of many runs**
  - Run many instances of the same program concurrently, each on a different processor.
  - *High-throughput computing*.
  - Don’t need to change the program.

High-throughput Computing

- Many researchers want to run the same program many times with many different input parameters and/or input data files.
- These are often called *parameter sweep* applications.
- If each program run is independent, then different runs can be run at the same time on different processors.
- Each program run is on a single processor, so takes about the same time to execute as on a PC.
- However if you use 20 processors at once, results from all runs can be completed 20 times faster.

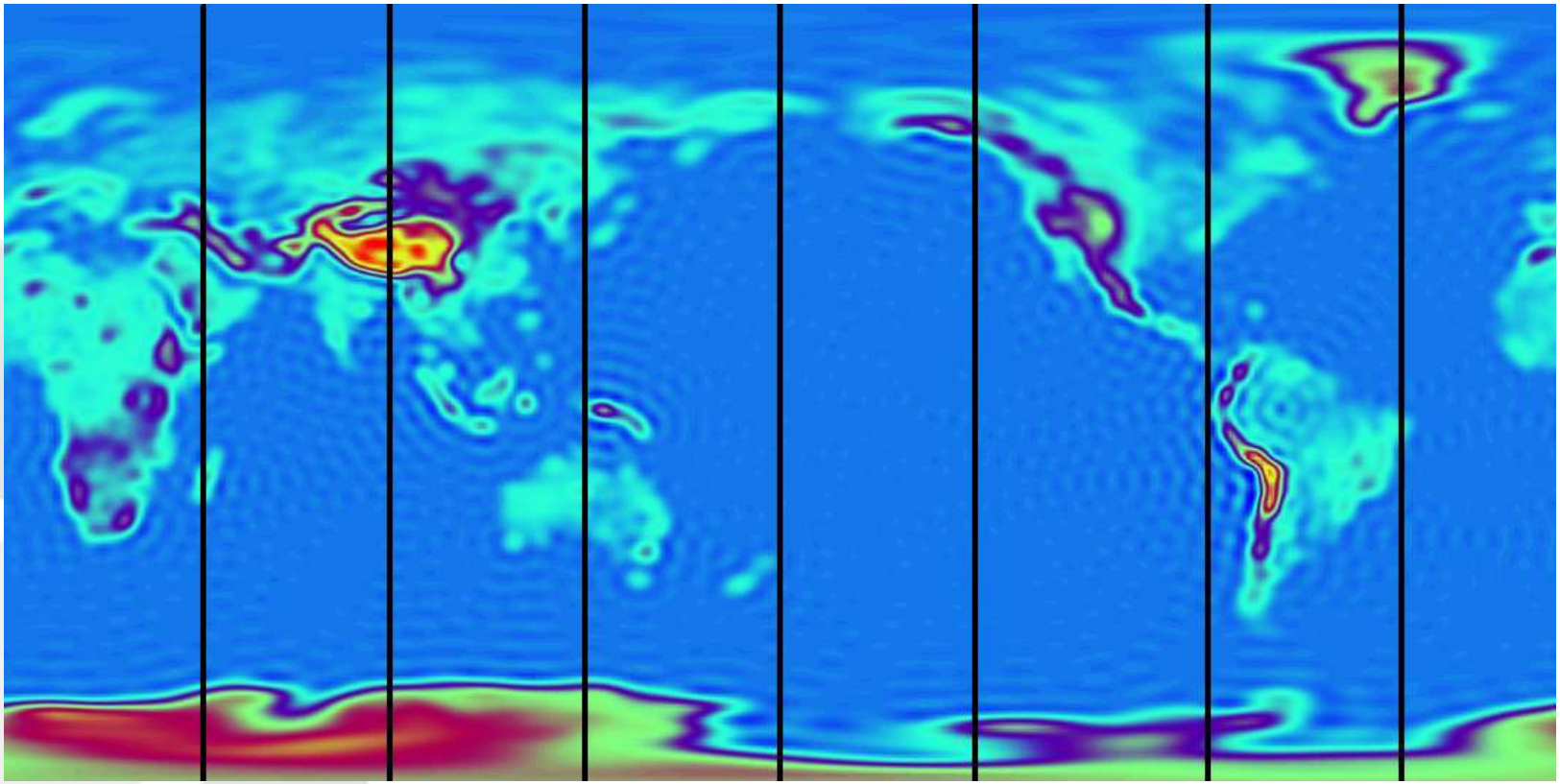
High-throughput Computing

- Multiple runs can be set up using shell scripts or submitting *array jobs* to the HPC system.
- Or can use a software package to manage setting up the runs and submitting them to the parallel computer.
  - or to a cloud, or a network of workstations.
- Nimrod, Condor, etc.
- Nimrod/O used for optimization
  - gets results for many input parameters and uses optimisation algorithm to find an optimal solution

High Performance Computing

- Use multiple processors to speed up program run-time, by dividing up the computation among CPUs.
- Requires changing the program - *parallel programming*.
- Usually achieved by splitting up the data to process to different processors - *data parallel computing*.
- Each processor does processing on its section of the data, concurrently with all the other processors.
- Processors may need to access data stored in the memory of another processor (on a cluster), or in banks of memory shared between processors (SMP).

Data Parallelism



Parallel Computers

- Many compute nodes (or servers) connected by a fast network
  - Usually Infiniband or 10 Gbit Ethernet
- Each node has multiple processors
  - Usually 2 or 4
- Each processor has multiple processing cores
  - Usually 4 to 16
  - Manycore (32 or more) coming soon
  - GPUs or custom chips have hundreds of cores
- A single compute node can have lots of cores
  - 32 or 64 now inexpensive and common
  - About 15 years ago 64-processor SMP was Top 500!

Shared and Distributed Memory

Shared memory (SMP)

- Processing cores on a compute node all have shared access to all the memory on the node
- Parallel programs often written using multiple *threads*, usually with one thread per processing core

Distributed memory (cluster)

- Processing cores on one compute node can't directly access memory from other nodes
- Program needs to send information using *message passing*
- Parallel programs often written using Message Passing Interface (MPI) standard

Developing Parallel Programs

- To develop a parallel algorithm or a parallel program, or to parallelise an existing sequential program, we need to:
  - Identify and exploit potential parallelism
  - Try to balance workload between processors
  - Minimize inter-processor communication or synchronization overhead
- Sometimes tasks are independent so program is “embarrassingly parallel”
- If not, this will require some thought, and some modifications to the sequential algorithm and/or program

Easy Parallel Programming

- Parallel programming is (usually) difficult.
- But you might get lucky and have little or no work to do.
- Many commonly used programs already have parallel versions, e.g. BLAST, Gaussian, OpenFOAM, ...
  - Some will only run on multiple cores on a single compute node
  - Some will run across multiple compute nodes (usually MPI)
- Some commonly used math libraries (e.g. LAPACK), ODE and PDE solvers, FFT libraries, etc, have parallel versions
  - So use those instead of sequential (single processor) versions.
  - Works well ifmost of the compute time is in these routines.
- Compilers can automatically generate parallel code.
  - But it may not be very efficient
- Or you may be able to split the work into multiple jobs.

Hard Parallel Programming

Four main approaches to parallelizing a program:

- Threads
  - Only works with shared memory; hard unless you use Java or other language with high-level thread library, or compiler does it for you.
- OpenMP
  - Only works with shared memory; easy for C, C++, Fortran (compiler directives)
- High Performance Fortran
  - Works on clusters or SMPs; easy, but only for Fortran 90 (mostly compiler directives)
- Message Passing Interface (MPI)
  - Works on clusters or SMPs; works for C, C++, Fortran, Java; best performance (OpenMP and HPF don't give good performance for some applications); but hardest to program

GPUs

- Consumer computer gaming market has driven huge performance increases in GPUs, which now have higher performance than CPUs.
- Modern GPUs are like parallel computers on a chip.
- GeFORCE GTX580 GPU has 512 cores
  - 1.3 TFlops single precision
  - 0.33 TFlops double precision

GP-GPUs

- High performance of GPUs for gaming has led to development of general purpose GPUs (GP-GPUs)
- High-performance GPUs aimed at technical computing rather than gaming
  - More general processing capability, fast double precision floating point, large error-correcting GPU memory
- But much more expensive than standard GPUs
- nVIDIA Tesla M2090 GPU has 512 cores
  - 1.3 TFlops single precision
  - 0.66 TFlops double precision

GPUs

- But GPUs have a specialised processor architecture and programming model, so programs need to be rewritten for GPUs.
- Many applications have now been ported to GPUs.
- Some applications run very well on GPUs and can even scale across multiple GPUs.
- However some give little or no performance benefit over a compute node with many cores.
- Your mileage may vary – check speedups for the application you are interested in.

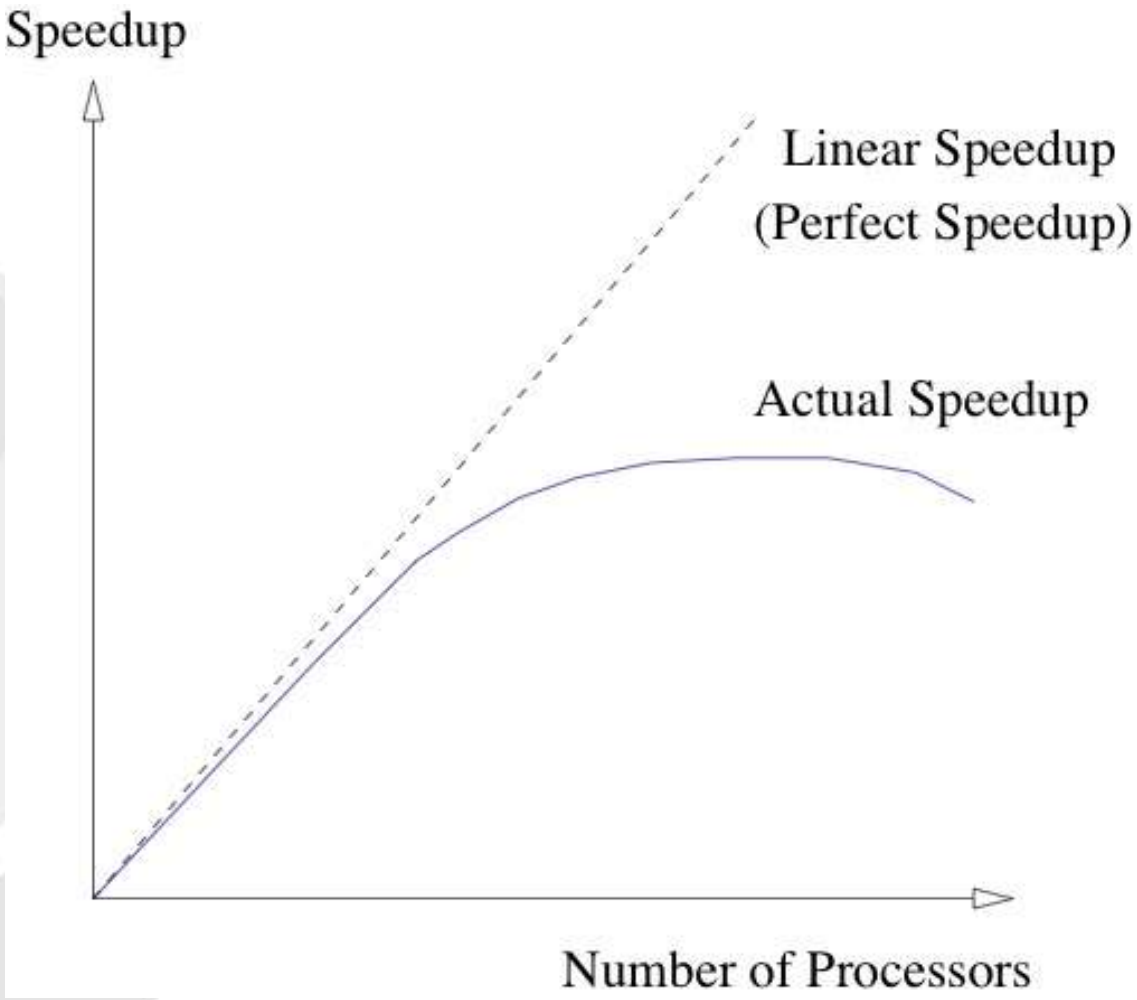
Programming Guides

- eRSA user guides have links to books, online information and training material for parallel programming and GPU programming
- MPI and OpenMP training material and courses available from NCI
  - Held in Adelaide occasionally if enough interest

Speedup

- Aim to get speedup of program run time on multiple processors
- Speedup = Time on 1 core / Time on N cores
- Would like to get speedup of N on N cores
- But not guaranteed
- Parallel overheads
  - Load imbalance
  - Communication
  - Sequential parts

Speedup


Speedup
Linear Speedup
(Perfect Speedup)
Actual Speedup
Number of Processors

Find the right number of cores

- Run your program and measure speedup on different numbers of cores
  - e.g. 1,2,4,8,16,32,48 on Tizard CPU nodes
  - And more cores on multiple nodes if possible
- Find the best number of cores to use
  - Where adding more cores doesn't give much additional speedup
  - e.g. using 8x more cores for 2x more speedup is wasteful!
- Note that best number of cores will vary depending on the problem size and complexity, i.e. the amount of computation required
  - So optimum cores may vary for different problem sizes

eRSA

Advancing Research Innovation

Upcoming workshops

How to use HPC and Cloud Clusters to enhance your research outcomes
9 March, University of Adelaide

R-Studio in the Cloud
15 April, University of Adelaide

Talk to us after the workshop to register