Scientific Parallel Programming

by Wadud Miah

Research Computing Services, UEA

Outline of Presentation

- Reasons for Parallelisation;
- Power of Supercomputers;
- Parallel Architectures;
- Parallel Efficiency;
- Important Parallel Terminology;
- Flynn's Taxonomy;
- Patterns for Parallel Programming.

Reasons for Parallelisation

Scientific codes were usually sequential and hardware was not able to keep up with program execution;

With the advent of multi- and many-core architectures, software is lagging behind hardware performance;

Even desktops, laptops and mobile phones have many cores which software is failing to fully utilise;

CPU frequencies have stagnated for many years at ≈ 3.4 GHz and is not going to increase anymore;

The future of scientific computing is in multi- and many-core architectures so we need to parallelise our applications.

Power of Supercomputers

The world's most powerful supercomputer (Japan's K computer) has 705,024 CPU cores;

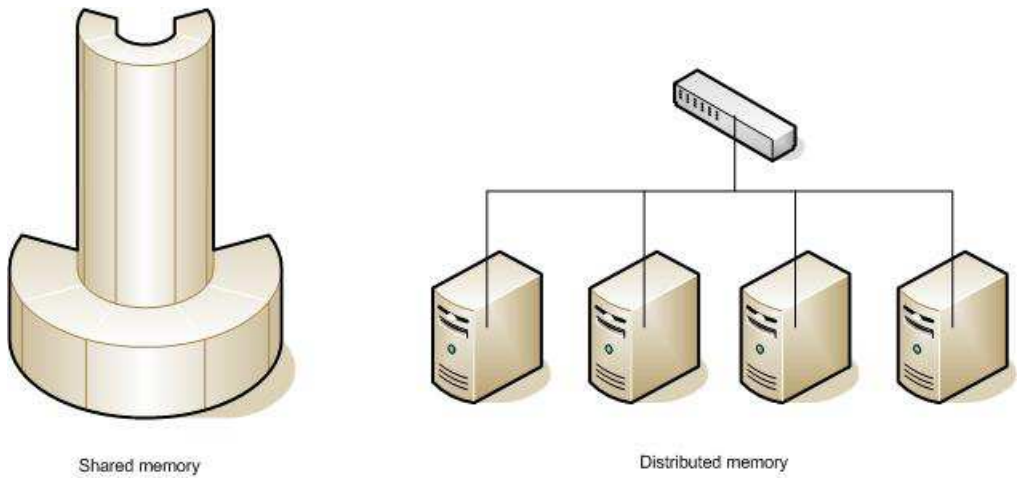
Accelerator cards such as Nvidia Tesla GPUs have about 512 lightweight cores;

The Grace cluster provided by the Research Computing Services group has in total 2,028 cores;

The solution to greater performance is parallel programming in many core environments where number of cores is $\gtrsim 256$.

Parallel Architectures

There are two types of parallel architectures: 1) shared memory and 2) distributed memory;



Each node in the distributed memory architecture has its own memory space;

Every CPU in the shared memory architecture has access to the entire memory space;

The MPI parallel library is mainly used in distributed memory architectures;

The OpenMP library is only used on shared memory architectures;

The OpenMP library is easier to use than MPI, but this is limited to codes on a single computational node on the cluster (48 GB of RAM and 12 CPU cores);

OpenMP, although easier to implement, does not scale as well as MPI codes.

Parallel Efficiency

Let T_1 be the time it takes for a program to complete on a single processor;

Let T_p be the time it takes for a program to complete on p processors;

The dimensionless speed up is defined by:

$$S_p = \frac{T_1}{T_p}$$

Ideally, S_p grows linearly with p. This property is expressed as having good scalability;

If $S_p > p$ then this is known as super-linear speed up;

However, $S_p \to \epsilon p$ (where $\epsilon < 1$) as $p \to \infty$, i.e. it asymptotes to a value less than p;

The dimensionless parallel efficiency is thus defined as:

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

Ideally, $E_p = 1$, namely 100%; However, E_p sometimes peaks at a certain value of p and then decrease $\rightarrow$ poor scalability;

Efficiency E_p depends on many factors: a) network topology b) amount of inter-process communication c) parallel algorithm d) amount of code that can be parallelised.

If Q is the fraction of code that can be parallelised, the speed up is then bounded by Amdahl's law:

$$S_p \leq \frac{1}{(1-Q) + Q/p}$$

If $p \to \infty$ then $S_p \leq 1/(1-Q)$. So if parallel portion of code is 75%, then $S_p \leq 4$;

Theoretically, $S_p \to \infty$ if 100% of code can be parallelised. This is generally not the case as some pre- and post-processing is done sequentially;

Important Parallel Terminology

Task: a set of instructions that can be conveniently grouped together;

Unit of execution (UE): a container of a task. In computing terms, this is a process or thread;

Processing element (PE): a hardware element that processes a unit of execution (UE). This is usually a CPU core and can be a GPU core;

Load balancing: when codes are parallelised, UEs are mapped to PEs. This is usually a one-to-one mapping, although you can also have many-to-one mapping. It is important to ensure that all units of execution (UEs) have the same amount of work, otherwise some PEs will be doing nothing;

Synchronisation: this mechanism ensures certain UEs are executed in a certain order for program correctness;

Synchronous and asynchronous: how events are coordinated. If two events must happen at the same time, this execution is known as synchronous. If no coordination is required, then this is asynchronous;

Race conditions: this is an error that occurs when results of a parallel program vary with the execution of the same program with the same data. This usually occurs when multiple UEs share the same data;

Deadlock: Another error where UEs are waiting for each other before proceeding. For example, UE A might be waiting for B and B might be waiting for A before either UE can proceed. This will result in a deadlock where nothing can proceed any further, and the parallel program blocks forever;

Message: as UEs work together to solve a large problem, they will often need to communicate by passing messages. This is the overhead of parallelisation. The time it takes to send a message of size N is given by:

$$T = \alpha + \frac{N}{\beta}$$

where α is known as the latency and β is known as the bandwidth. For scientific programming, it is important that $\alpha \ll 1$ and $\beta \gg 1$;

Granularity: the amount of tasks contained within a unit of execution (UE). If very little work is contained with the UE, then the parallel overhead will dominate. If too much work is contained with a UE, then this will result in poor load balancing.

Flynn's Taxonomy

Parallel computers can be described according to the number of instruction and data streams:

- **Single Instruction Single Data (SISD)**: a single instruction processes a single element of data;
- **Single Instruction Multiple Data (SIMD)**: a single instruction processes a stream of data. This is known as a vector supercomputer;
- **Multiple Instruction Single Data (MISD)**: no such parallel computer exists;
- **Multiple Instruction Multiple Data (MIMD)**: each processing element (PE) processes its own stream of data. This is the widely available type of parallel computer.

Parallel Pattern Language

The design and implementation of parallel applications is an iterative process and the stages are listed below:

Finding Concurrency - identify tasks that can be executed concurrently in your scientific problem. This could be updating cell values, particles, matrix/vector elements;

Algorithmic Structure - developing an algorithm which exploits the identified concurrency using a decomposition technique;

Supporting Structures - using a suitable programming paradigm for your code and data structures;

Implementation Mechanism - how the parallel pattern is then implemented using a suitable programming environment.

Finding Concurrency

The finding concurrency stage involves mapping tasks to units of execution (UEs) in a way that ensures good load balancing;

This is an iterative process and the stages are:

1. Decomposition:
   (a) Task decomposition;
   (b) Data decomposition.
2. Dependency analysis;
   (a) Group tasks;
   (b) Order tasks;
   (c) Data sharing.

Decomposition

The decomposition stage is crucial in designing a parallel algorithm;

This will determine how well the code is load balanced and its scalability;

The two methods of parallel decomposition are:

1. **Data decomposition**: linear algebra (matrices and vectors), finite difference schemes;

2. **Task decomposition**: particle physics, protein folding, medical imaging.

The characteristics of your algorithm should include:

- **Flexibility**: the code should be sufficiently portable;
- **Efficiency**: the code scales well for different number of processing elements (PEs);
- **Simplicity**: the code should be simple enough for the purpose of maintainability and debugging.

You will need to find the right balance for your application.

Dependency Analysis

To satisfy problem constrains, the following should be considered:

1. **Group tasks**: it is sometimes convenient to group tasks. For example, in molecular dynamics, the tasks can be grouped in the following way:

   - vibrational forces on an atom;
   - rotational forces on an atom;
   - non-bonded forces on an atom;
   - update the velocity and position of an atom.

2. **Order tasks**: the ordering of tasks are required to be satisfied for program correctness;

3. **Data sharing**: when tasks are being executed concurrently, they sometimes share data structures which need to be co-ordinated. Data sharing inhibits scalability, so avoid this if you can. Data can be:

   - **Read only**: data is read by UEs, therefore little protection is required. If it is updated by a single UE, then the remaining UEs will need an updated copy of the data;

   - **Local only**: data is local to UEs, therefore little protection is required. If another UE requires this data, it has to be sent as a message;

   - **Read write**: data is read and written to by multiple UEs. This co-ordination is the most complex and the data requires protection to avoid race conditions. There are techniques to protect data structures (called mutual exclusion where only one UE can update it at one time).

Algorithmic Structure

The algorithmic structure of a parallel program is concerned with ensuring tasks are efficiently balanced across the units of execution (UEs);

The resulting algorithm should have the following characteristics some of which conflict:

1. **Efficiency**: it is important that the code finishes sufficiently quickly by making efficient usage of architecture;

2. **Simplicity**: a simple algorithm is easy to understand, subsequently making it easy to develop, debug and maintain;

3. **Portability**: ideally, the code should run on a large number of parallel architectures, particularly if you are going to release

it for other users. If it is only going to run on a local cluster, then this may not be so important;

4. **Scalability**: with the advent of petascale architectures, it is important that the parallel algorithm scales for a large number of PEs $\gtrsim 256$. The bottleneck today is in applications.

It is up to the application programmer to find the right balance;

There are two ways to decompose a parallel algorithm and this should be carefully chosen:

- Organising by data;

- Organising by tasks.

Data Parallelism

Data parallelism is where UEs are mapped to data. Each UE processes its block of data which is part of the larger problem;

A lot of scientific codes fit into this paradigm, where problems are mapped onto data structures that are arrays which represent the computational domain of a scientific problem.

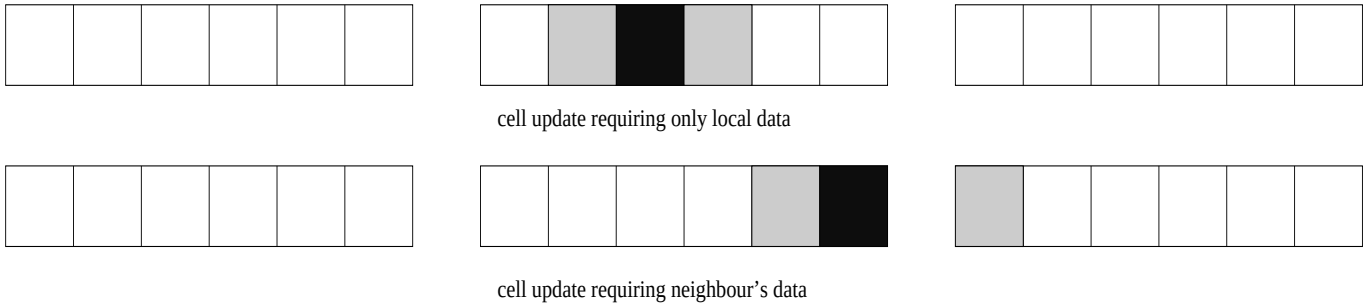
Heat Diffusion

The equation that governs heat diffusion across a medium is governed by the differential equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

$$\text{discretisation} \Rightarrow u_i^{n+1} = u_i^n + \frac{\Delta t}{\Delta x}\left\{u_{i+1}^n - 2u_i^n + u_{i-1}^n\right\}$$

This problem can be easily decomposed into chunks of data where each UE works on its chunk which is shown below:



For cells on the edge of the chunk, it has to communicate with its neighbour;

The number of chunks is given by N/p, where N is the number of cells and p is the number of processing elements (PEs).

Matrix Multiplication

The matrix multiplication problem is another candidate for data parallelism. This is described by the equation:

$$C_{i,j} = \sum_{k} A_{i,k} B_{k,j}$$

The matrix C can be decomposed into either square or rectangular blocks;

This problem requires no communication between the UEs, and this is called "embarrassingly parallel".

Task Parallelism

Task parallelism is where UEs are mapped to tasks. Each UE processes a task which is part of the larger problem;

A lot of physics and chemistry codes fit into this paradigm where particles and atoms are processed by UEs and are advanced in time;

If data parallelism is selected for such codes, this will lead to load imbalance as the atoms (or any other physical element) are likely to be irregularly scattered over the computational domain.

Molecular Dynamics

The equation that governs molecular modelling is expressed by Newton's second law of motion:

$$m_i \frac{\partial^2 \mathbf{r}_i}{\partial t^2} = \mathbf{F}_i$$

where $\mathbf{r}$ is the position vector and $\mathbf{F}$ is the force vector for atom i;

Data sharing between UEs: non-bonded forces;

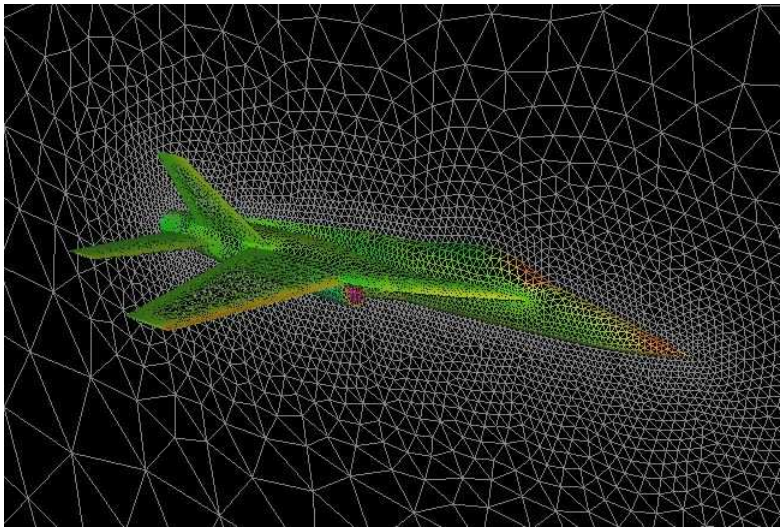
The number of atoms that each UE processes is p/N, where N is the number of atoms.

Computational Fluid Dynamics

A lot of CFD calculations are based on iterative methods where a UE processes each iterative problem. The governing equations are expressed as:

$$\frac{\partial}{\partial t} \int \mathbf{U} \, \mathrm{d}t = \oint_{\partial \Omega_i} \mathbf{F}_i \, \mathrm{d}\Omega_i$$

where $\mathbf{F}_i$ is the flux function across interface i. Cells are assigned to UEs, instead of regions to ensure a good load balance.



Processes and Threads

A unit of execution (UE) can be represented by either a process or thread;

There are differences between the two:

- A process is defined as a program in execution. It contains the program memory/data, register values and program counter which points to the next instruction to execute;

- Process that need to communicate must send messages to each other;

- Process creation and switching between them is expensive as the entire process data structure has to be dealt with;

- A thread is defined as a independent execution path of a process and shares the memory space of a process;

- Threads can easily communicate with each other via the shared memory;

- Thread creation and switching is not an expensive task.

The characteristics of processes and threads will determine which programming model to use. The two primary models are:

- **MPI**: processes are used. This is mainly used on a distributed memory architecture such as computational clusters;

- **OpenMP**: threads are used. This is used mainly on a shared memory architecture such as a massive parallel machine, e.g. SGI Ultix UV.

MPI codes are more complicated to program but scale better than OpenMP codes;

OpenMP codes are easier to program but do not scale as well as MPI codes;

The model you adopt will depend on the available architecture and performance requirements of your code.

Mapping UEs to PEs

A compute bound UE maps to a single PE;

IO and compute bound UEs can map to a single PE. When a UE waits for IO, another UE can consume the PE (context switching). This ensures that the PE is sufficiently busy;

The cost of switching between threads is very low, whereas the cost of switching between processes are high;

Programs that carry out data mining on large data sets are implemented using the thread-based model, namely OpenMP.

Supporting Structures

The supporting structures phase is concerned with the structure of the parallel code before it is finally implemented;

The parallel program is supported by data structures which is considered here;

The available program structures for parallel programming are:

- **Single Program Multiple Data (SPMD)**: all units of execution (UEs) execute the same program and has its own data which is operates on;

- **Loop Parallelism**: parallelising computational intensive loops;

- **Master/Worker**: a master UE sets up a set of worker UEs and partitions the work equally amongst the workers.

The available data structures for parallel programming are:

- **Shared Data**: data that is shared between the UEs. This is a classic cause of poor scalability as the data has to be protected from race conditions;

- **Distributed Array**: the global problem is represented by an array which is decomposed and is processed by UEs.

The suitability of the program structures are listed below:

	Task Parallelism	Data Parallelism
SMPD	****	****
Loop Parallelism	****	***
Master/Worker	****	*

The suitability of an implementation model is shown below:

	OpenMP	MPI
SMPD	***	****
Loop Parallelism	****	*
Master/Worker	**	***

The master/worker structure should only be used if it is difficult to load balance amongst worker UEs - it is a bottleneck that negatively effects scalability.

SPMD Structure

Initialise: the parallel environment is initialised and handles are created so that UEs can communicate;

UE identifier: the UE obtains a unique identifier which is in the range $0 \leq i < p$;

Execute program for each UE: the program is executed by each UE with its own memory space. Branches can be controlled by the identifier i;

Distribute data: the UEs work on its chunk of data from distributed array using the identifier i. This is stored in memory local to the UE. The UEs communicate with each other using explicit messages;

Finalise: the parallel program completes and closes down. All execution terminates and data is recombined and saved.

Loop Parallelism Structure

Identify CPU intense loops: parallelise loops that iterate a large number of times and execute a number of tasks in the body of the loop;

Eliminate dependencies: loops that contain dependencies in the form $u^{n+1} = f(u^n, u^{n-1})$ can not be parallelised. Remove them if you can, or mitigate using synchronisation techniques. Note: synchronisation reduces scalability of codes;

Parallelise loops: using programming pragmas, parallelise loops;

Optimise schedule: optimise the schedule on how UEs are mapped to PEs to increase load balancing. This will increase the scalability of the code.

Master/Worker Structure

The master/worker structure is mainly used when tasks are created dynamically and require co-ordination for workers. This structure has the following template:

Initialise: the parallel environment is initialised and handles are created so that UEs and communicate;

UE identifier: the UE obtains a unique identifier which is in the range $0 \leq i < p$. The master is usually the UE with identifier $i = 0$;

Create tasks: tasks are created and are mapped to a fixed number of UEs;

Launch worker UEs: worker UEs are then executed PEs;

Worker completes work: a worker UE completes the task whilst other UEs are still processing their tasks;

Launch worker UEs until work is done: the completed UE is assigned more tasks. This is how good load balancing is achieved. This is repeated until all the work is completed;

Collect results: collect all completed work from worker UEs;

Finalise: the parallel program completes and closes down. All execution terminates and data is recombined and saved.

Shared Data

Data sharing amongst units of execution (UEs) is sometimes inevitable and this reduces scalability;

When data is shared, it must be protected to prevent race conditions $\Rightarrow$ unexpected program behaviour;

This is more of an issue with shared memory programs. In single program multiple data structures (SPMD), data is local to the UE, so sharing is almost eliminated;

When shared data is accessed by threads they must be contained within critical regions to ensure data consistency;

Ensure synchronisation is carried to so that UEs have the latest copy of the data.

Distributed Array

A distributed array represents a discretised problem and is then distributed to the units of execution (UEs);

The array data is usually read and written to a single file, and this is a major cause of bottlenecks and reduced scalability. The solution to this is to use a parallel IO library so that each UE can operate on its portion of the file;

Available parallel IO libraries:

- MPI-IO: the MPI 2 standard implements this feature;
- Parallel NetCDF: a parallel version of the popular NetCDF data format;

- Parallel HDF5: a parallel version of the popular HDF data format for data that has a hierarchical structure, e.g. adaptive mesh refinement computational domains.

Use a binary file format such as NetCDF or HDF5 as reading data in this format is much quicker;

If implementing check-pointing, using a parallel IO library will significantly increase code scalability and performance.

Implementation Mechanisms

The implementation mechanisms stage involves using the correct parallel implementation for the problem at hand;

The previous stages of 1) finding concurrency, 2) algorithm structure and 3) supporting structures stages usually determine which implementation mechanism to choose;

The implementation mechanism involve the following:

- UE management - creation and finalising units of execution;
- Synchronisation - enforces the ordering of events in a parallel program;
- Communication - the communication between UEs.

There are two types of parallel computers: shared memory and distributed memory;

OpenMP is a shared memory implementation that can be used to program on a single node;

MPI is a distributed implementation that can be used to program on a number of nodes (e.g. computational cluster).

UE Management

An example MPI code is shown below:

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

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

  MPI_Init( &argc, &argv ); /* processes are created */
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  printf( ``Hello from process %d of %d\n'', rank, size );

  MPI_Finalize(); /* processes are closed down */
  return 0;
}
```

An example OpenMP code is shown below:

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

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

#pragma omp parallel
{ /* processes are created */
  printf(‘‘Hello, world\n’’);
} /* processes are destroyed */

  return 0;
}
```

To execute an MPI program with 4 units of execution (processes), type the command:

```
mpirun -n 4 ./xhpl
```

To execute the OpenMP program with 4 units of execution (threads), type the command:

```
icc -openmp xhpl.c -o xhpl
export OMP_NUM_THREADS=4
./xhpl
```

Use `-fopenmp` for GNU compilers and `-mp` for PGI compilers.

Synchronisation

To synchronise processes in MPI, use the following subroutine:

```
MPI_Barrier( MPI_COMM_WORLD );
```

To synchronise processes in OpenMP, use the following pragma:

```
#pragma omp barrier
```

The above directive must be contained within a parallel region.

Communication

Unless the algorithm is embarrassingly parallel, UEs will need to communicate with each other;

In OpenMP, threads can communicate with each other easily using shared memory directly;

In MPI, processes need to communicate with each other by passing messages using:

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm );
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status );
```

There are is a catalogue of point-to-point and collective MPI communication subroutines.

Questions and Discussion

Further Sessions

- Debugging, profiling and optimisation with workshop;
- Introduction to OpenMP with workshop;
- Introduction to MPI with workshop;
- GPU programming in CUDA with workshop.

Further Reading

1. *Parallel Programming for Multicore and Cluster Systems* - T. Rauber and G. Runger. Springer

2. *Patterns for Parallel Programming* - T. Mattson, B. Sanders and B. Massingill. Addison Wesley

3. *Parallel Programming with MPI* - P. Pacheco. Morgan Kaufmann

4. *Parallel Programming in OpenMP* - R. Chandra, R. Menon and J. McDonald. Morgan Kaufmann