What is parallel computing?

- A computational method that utilizes multiple processing elements to solve a problem in tandem
- Implementing parallelism requires modification of an application such that the workload is decomposed into independent tasks

Why bother?

- Faster results
  - N people can dig a hole N times faster than 1 person
- Larger amounts of memory available
  - The price of memory and the underlying architecture increases dramatically as one adds more memory to a computer
- The technological trend is for greater parallelism to be incorporated at all levels of the computing hierarchy.

Parallelism in hardware

Parallelism is abundant in computer hardware and primarily drives increased throughput:

- Parallel instruction scheduling (multiple instructions computed at once)
- Parallel data buses that move many bits of data simultaneously
- Multiple processing cores per computer
- Multiple computers connected by a network that work as a whole

With respect to parallel programming, we are concerned with the last two. The main distinction between them is how each of the processing cores is connected to the memory of the entire system, and hence, how they communicate data amongst one another.

- Multiple cores on the same computer have what is called "shared-memory"
  - all of the cores are able to directly access all of the memory
- Multiple computers on a network have what is called "distributed-memory"
  - each of the cores can directly access only it's local memory

The line between these are often blurred, and modern platforms are primarily a combination of the two.

Parallel programming

Shared-memory systems are usually programmed to take advantage of multiple processing cores by threading applications:

- OpenMP
- POSIX threads (pthreads)

Both are application programming interfaces (APIs). OpenMP is easier to implement and is generally supported by the compiler, whereas pthreads is more complex and is used as an external library. We will only focus on OpenMP.

On distributed memory systems one uses a team of processes that simultaneously run on a network of computers and pass data in the form of messages to one another. This is facilitated by the Message Passing Interface (MPI).

This also an API and is typically available as an external library and runtime programs that are used to compile and execute MPI programs.

There are many MPI implementations, some that are very portable and open-source (Open MPI), and others that are highly-optimized and vendor-specific.

One can run MPI programs on shared-memory systems, or even mix the two approaches.

Breaking down the problem

There are two main approaches to decomposing a problem so that it can run in parallel:

- Data parallelization
  - Data which is being processed is broken up into independent chunks
    - Each compute element only deals with its local data
  - Most common approach for scientific applications
    - eg. Mesh-based calculations, linear algebra
- Task parallelization
  - Different computational tasks performed by a program are done in parallel
  - More useful for programs that have distinct and independant operations
    - eg. Real-time signal correlator
      - One thread to do I/O, one to do message passing, other threads process data
- Fine grained vs coarse grained parallelism
  - Refers to the amount of work that each compute element is given before a synchronization point

The parallelization process in practice

In some cases a major re-design of a code is necessary to convert it to run in parallel. Data structures may have to change and new methods to solving problems may have to be implemented. Once these changes are made, and for more straight-forward problems, the parallelization process follows the same procedure used to optimize a program:

- Identify "ho t-spots" (functions/routines that use the most cycles)
  - Requires profiling / timing instrumentation in the code
- Modify code to parallelize hot-spots
  - Not all operations can be parallelized
- Check for accuracy and correctness, debug any program errors that are introduced
- Tune for performance, repeat process as necessary

Amdahl's Law

In a parallel computational process, the degree of speedup is limited by the fraction of code that can be parallelized:

$$T_{parallel} = \{ (1 - P) + P / NP \} * T_{serial} + overhead$$

Where:

Tparallel	== time to run application in parallel
P	== fraction of serial runtime that can be parallelized
NP	== number of parallel elements (cores/nodes)
Tserial	== time to run application serially
overhead	== overhead introduced by adding the parallelization

To make compare the parallelization to the original serial code, one uses the term scaling, which is just:

$$Scaling = T_{serial} / T_{parallel}$$

*Note that this is for "strong-scaling" applications.

Scaling

- efficiency of a program in utilizing parallel resources
- 2 different measures are often used:
  - Weak scaling
    - Workload / compute element is kept constant as one adds more elements
    - In a linearly scaling case, a problem N times larger takes the same amount of time to do on N processes
    - An example is large N-body simulations
  - Strong scaling
    - Total workload is kept constant as one adds compute elements
    - Linear scaling in this case means that the runtime will decrease in direct proportion to the number of compute elements used
    - An example is inverting a fixed-size matrix

One usually compares parallel efficiency to the expected linear scaling

Processes and threads

- A process is an instance of a computer program that is being executed
  - Includes binary instructions, different memory storage areas, hidden kernel data
  - Performs serial execution of instructions

Depending on the nature of the execution stream, it may be possible to perform many operations simultaneously (the instructions are independent of one another).

To process these instructions in parallel, the process can spawn a number of threads.

- A thread is a light-weight process
  - Inherently connected to the parent process, sharing it's memory space
  - Threads each have their own list of instructions and independent stack memory
- The process of splitting up the execution stream is typically called a "fork-join" model
  - Execution progresses serially, then a team of threads is created, work in parallel, and return execution flow to the parent process

OpenMP

- programmed by utilizing special compiler directives and functions in the OpenMP runtime library
  - the compiler will only consider the directives if it is directed to do so
- In simple cases (loop-level parallelism), one doesn't have to make any code modifications, only addition of these directives
  - It's easy to incrementally add parallelization
- Blocks of code that are to be done in parallel are considered parallel regions, and must have only 1 entry and 1 exit point.
- Threaded programs are executed in the same fashion as serial programs
  - OMP_NUM_THREADS environment variable to set number of threads, or else hardwire it in your code

Data Scoping

- The concept of data scoping is crucial to using OpenMP
- Scoping refers to how different variables should be accessed by the threads. Basic scoping types are:
  - Private: each thread gets a copy of the variable that is stored on it's private stack
    - Values in private variables are not returned to the parent process
  - Shared: all threads access the same variable
    - If variables are only read from, it's safe to declare them as shared
  - Reduction: like private, but the value is reduced (sum,max,min,etc.) and returned to the parent process.

A simple OpenMP example

```
#pragma omp parallel default(shared) private(i,x) reduction(+:pi_int)
  {
#ifdef _OPENMP
  printf("thread %d of %d started\n", omp_get_thread_num(),
    omp_get_num_threads());
#endif
#pragma omp for
  for (i = 1; i <= n; ++i) {
      x = h * ( i - 0.5 ) ;    //calculate at center of interval
      pi_int += 4.0 / ( 1.0 + pow(x,2) ) ;
    }
 }
```

What is a message?

- a way to transfer the value of local memory to a remote process
- Consists of network packets
  - Sensitive to the network bandwidth (how much data can be transferred in a given time) and latency (how long it takes for each packet to move from source to destination)
- Can be synchronously or asynchronously sent between processes
- May require buffer memory usage to store outgoing/incoming messages, can be explicitely stated

MPI

- programmed by utilizing functions/routines in the MPI library
- Requires more work than implementing OpenMP
  - involves new data structures and increased algorithmic complexity
- portable
  - runs on both shared and distributed memory
- MPI programs are executed on top of an underlying MPI framework. Typically each computing element (network node) runs an MPI daemon that facilitates the passing of messages
- Usually executed with a special mpirun command
  - takes the number of processes to be used as arguments

The 6 basic MPI routines

MPI essentially boils down to only 6 different subroutines:

- MPI_Init()
- MPI_Comm_size(MPI_COMM_WORLD,&numprocs)
- MPI_Comm_rank(MPI_COMM_WORLD,&rank)
- MPI_Send(buffer,buffer_size,MPI_TYPE,to_rank,tag,mpi_comm)
- MPI_Recv(buffer,buffer_size,MPI_TYPE,from_rank,tag,mpi_comm,status)
- MPI_Finalize()

All other communication patterns fundamentally consist of sends and receives.

There are different types of sends and recieves

- Buffered, asynchronous, synchronous

More complex communication routines exist

- Reductions, broadcasts, barriers, alltoall

The simplest MPI example

```
#include "mpi.h"
int main( int argc, char *argv[])
{
  int numprocs, myrank;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  printf("Hello World from rank %d of %d\n", myrank, numprocs);

  MPI_Finalize();
  return 0;
}
```

Parallel Libraries

There are many useful parallel libraries that internally use OpenMP / MPI:

- Intel MKL
  - Threaded FFTs, linear algebra, random number generators
- FFTW
  - Threaded and MPI parallel FFT routines
- Scalapack
  - MPI routines for doing linear algebra

These can relieve a great deal of the burden of parallel programming – use them!

Be careful not to oversubscribe processors by calling threaded library routines from inside parallel regions.

Debugging

- While there are powerful commercial parallel debugging tools (Totalview, Intel Thread Analyzer, etc) we are cheap and don't have any
- The usual route is to use a lot of print statements
- Both gdb and idb support debugging threaded applications
- One can also run a separate debugger for each MPI process, which is sometimes helpful, but it is only limited to running on one node
- Many problems result from incorrect synchronization
  - Deadlocking (posting a receive before a send)