Parallel Programming

Introduction

- Parallel programming is using multiple cpus concurrently.

- Reasons for parallel execution:
  1. shorten execution time
  2. to permit a larger problem (memory resources)

- The days of waiting for the next-generation chip to improve your serial-code throughput are gone.
Amdahl’s Law

* Describes the time speedup one can expect as a function of the number of processors used and the fraction of parallel code:

\[ \text{speedup} = \frac{1}{1-p + p/N} \]

- \( N \) - number of procs
- \( p \) - fraction of parallel code
Types of Parallel Machines

* Symmetric Multiprocessors (SMP) - multiple cpus sharing memory resource, bus connection - kaibab, desktop Macs
Types of Parallel Machines

* Distributed computing - individual computing elements each with their own memory, and network connection - Cray T3E
Types of Parallel Machines

- Clusters - combine the above two models. SMP nodes can be connected by network - **slikrock, saddleback**
Types of Parallelism

* Process Parallelism (MPMD) - a code may contain different segments that can be computed concurrently. Example: ocean, land, and ice parts of climate model, or convection and radiation parameterizations in an atmosphere.

* Low overhead, but often limits on how many procs can be used.
Types of Parallelism

* Data Parallelism (SPMD) - the same code works on different data streams. For example, dividing a global domain into subdomains - each processor executes all the code for an individual subdomain.

* Data and process parallelism may be employed together.
Parallel Programming
Paradigms: Shared Memory

- Shared memory techniques launch threads during execution.
- Automatic Parallelizers - just turn on the compiler switch - it finds the do loops that can be done in parallel.
- Compiler Directives - Open MP is the current standard. User inserts ‘comments’ in code that compiler recognizes as parallelization instructions. Modest changes to code.
- Only works with shared memory.
Parallel Programming Paradigms: Message Passing

- Can work with both distributed and shared memory.
- MPI is the standard, several packages exist: MPICH2, lam-mpi, open-mpi.
- Library calls explicitly control the parallel behavior - extensive user rewrite of code. Code is explicitly instructed to send and receive messages from the other processes.
- Ross will discuss in much more detail next few weeks.
- Message passing and shared memory techniques can be used in a hybrid-mode.
Parallel Programming Concepts

- Synchronization - making sure all code gets to a certain point before proceeding.
- Load balancing - trying to keep processes from being idle while others are computing.
- Granularity - how much work is in each parallel section.
OpenMP - a Brief Intro

- OpenMP is an API for writing multithreaded applications in a shared memory environment
- It consists of a set of compiler directives and library routines
- Relatively easy to create multi-threaded applications in Fortran, C and C++
- Standardizes the last 15 or so years of SMP development and practice

Tutorial: http://www.osc.edu/hpc/training/openmp/big/fslde.002.html
OpenMP: http://www.openmp.org/
Open MP - first steps

* Identify parallel do-loops. Each do loop carries overhead so it can be helpful to have a larger outer do-loop for parallelism.

* Identify functionally parallel regions (Think F90 case construct as an analog).

* Identify shared and private data

* Identify 'race conditions' where shared data can change program output unexpectedly.
OpenMP - parallel do loop

```c
omp do shared(x) private(i)
omp schedule(static)
    do i = 1, 1000
      x(i) = a
    enddo
```

Diagram:

- thread 0
  - thread 0 (i = 1, 250)
  - thread 1 (i = 251, 500)
  - thread 2 (i = 501, 750)
  - thread 3 (i = 751, 1000)

- thread 0

...
Open MP - reduction

- Allows safe global calculation or comparison.
- A private copy of each listed variable is created and initialized depending on operator or intrinsic (e.g., 0 for +).
- Partial sums and local mins are determined by the threads in parallel.
- Partial sums are added together from one thread at a time to get global sum.
- Local mins are compared from one thread at a time to get gmin.

```c
#pragma omp do shared(x) private(i)
#pragma omp reduction(+:sum)
    do i = 1, N
        sum = sum + x(i)
    enddo

#pragma omp do shared(x) private(i)
#pragma omp reduction(min:gmin)
    do i = 1,N
        gmin = min(gmin,x(i))
    end do
```
Open MP - sections

```c
#pragma omp parallel
#pragma omp sections

#pragma omp section
  call computeXpart()
#pragma omp section
  call computeYpart()
#pragma omp section
  call computeZpart()

#pragma omp end sections
#pragma omp end parallel

  call sum()
```

- Each parallel section is run on a separate thread.
- Allows functional decomposition.
Open MP - data dependency

- Only variables that are **written** in one iteration and **read** in another iteration will create data dependencies.
- A variable cannot create a dependency unless it is **shared**.
- Often data dependencies are difficult to identify. APO can help by identifying the dependencies automatically.

**Recurrence:**

```fortran
do i = 2, N, 2
  a(i) = c * a(i-1)
enddo
```

**Is there a dependency here?**

```fortran
do i = 2, N, 2
  a(i) = c * a(i-1)
enddo
```

**Thread**

- Thread 0:
  - `a(2) = c * a(1)`

- Thread 1:
  - `a(3) = c * a(2)`

- Thread 2:
  - `a(4) = c * a(3)`

- Thread 3:
  - `a(5) = c * c(4)`

**Time**
OpenMP - run time

OpenMP Environment Variables

- **OMP_NUM_THREADS**
  - Sets the number of threads requested for parallel regions.

- **OMP_SCHEDULE**
  - Set to a string value which controls parallel loop scheduling at runtime.
  - Only loops that have schedule type **RUNTIME** are affected.

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads actually used in a parallel region (due to system load).
  - Default value is implementation dependent.