Chapter 3
OpenMP Introduction

OpenMP Introduction

- OpenMP (Open Multi-Processing)
  - OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.
- There are several reasons to use OpenMP:
  - OpenMP is the most widely standard for SMP systems, it supports 3 different languages (Fortran, C, C++), and it has been implemented by many vendors.
  - OpenMP is a relatively small and simple specification, and it supports incremental parallelism.
  - A lot of research is done on OpenMP, keeping it up to date with the latest hardware developments.
- A Few OpenMP References
  - The OpenMP API specification for parallel programming (OpenMP.org)
  - OpenMP (Lawrence Livermore National Lab)
  - OpenMP (Wikipedia)

Motivation

- Pthread is too tedious: explicit thread management is often unnecessary
  - If we have a sequential code and know which loop can be executed in parallel; the program conversion is quite mechanic
  - We should just say that the loop is to be executed in parallel and let the compiler do the rest.
  - OpenMP does exactly that!!!
- Can be implemented incrementally, one function or even one loop at a time.
  - A nice way to get a parallel program from a sequential program.
- OpenMP is portable: supported by Industry (e.g., HP, IBM, Intel, etc.) and academia
- Why are standards useful?

OpenMP Introduction

- The primary way programmers use OpenMP is to use directives inserted into the source code of their program. The directive in OpenMP follows the format:

  #pragma omp directive-name [clause [[,] clause] . . .] new-line

- For example, to express a DOALL parallelism for a loop, the following directive is inserted above the loop

  #pragma omp for [clause [[,] clause] . . .] new-line

- Where a clause is one of the following:
  - private (variable-list)
  - firstprivate (variable-list)
  - lastprivate (variable-list)
  - reduction (operator: variable-list)
  - ordered
  - schedule(kind [, chunk_size])
  - nowait

OpenMP Introduction

- Initially, designed for expressing loop-level parallelism (i.e. parallelism between loop iterations)

Sequential Program

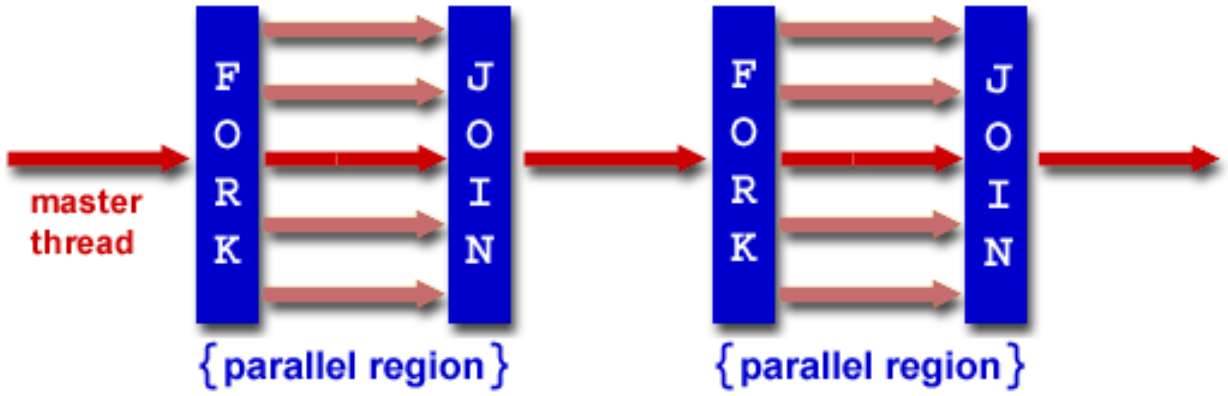
```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

Directives: #include and #pragma communicate info to compiler

OpenMP execution model



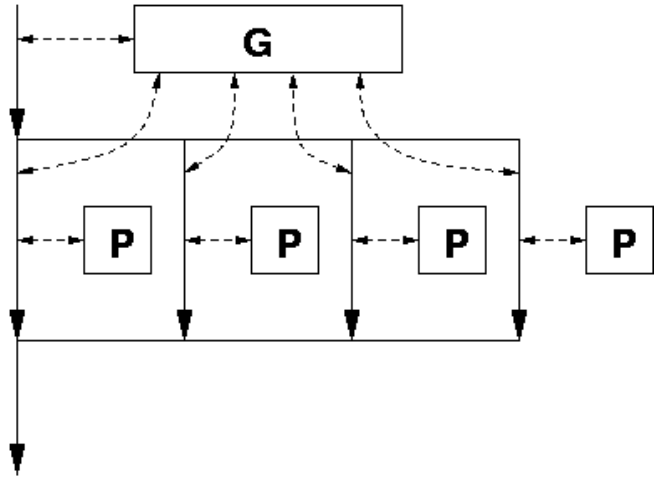
- OpenMP uses the fork-join model of parallel execution
  - All OpenMP programs begin with a single master thread.
  - The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK).
  - When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).

OpenMP general code structure

```
#include <omp.h>
main () {
  int var1, var2, var3;
  Serial code
  . . .
 /* Beginning of parallel section. Fork a team of threads. Specify variable scoping*/
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    /* Parallel section executed by all threads */
    . . .
    /* All threads join master thread and disband*/
   }
   Resume serial code
   . . .
}
```

Data model

- Private and shared variables
  - Variables in the global data space are accessed by all parallel threads (shared variables)
  - Variables in a thread's private space can only be accessed by the thread (private variables)



P = private data space
G = global data space

OpenMP directives

- Format:

  #pragma omp directive-name [clause,..] newline

  (use '\' for multiple lines)

- Example:

  #pragma omp parallel default(shared) private(beta,pi)

- Scope of a directive is one block of statements { …}

Parallel region construct

- A block of code that will be executed by multiple threads

#pragma omp parallel *[clause …]*
{
……
} (implied barrier)

Clauses: if (expression), private *(list),* shared *(list),* default (shared | none), reduction *(operator: list),* firstprivate*(list),* lastprivate*(list)*

- if (expression): only in parallel if expression evaluates to true
- private(list): everything private and local (no relation with variables outside the block).
- shared(list): data accessed by all threads
- default (none|shared)

The Reduction Clause

```
Sum = 0.0;
#pragma parallel default(none) shared (n, x) private (I) reduction(+ : sum)
{
  For(I=0; I<n; I++) sum = sum + x(I);
}
```

  - Updating sum must avoid racing condition
  - With the reduction clause, OpenMP generates code such that the race condition is avoided

- Firstprivate(list): variables are initialized with the value before entering the block
- Lastprivate(list): variables are updated going out of the block.

The omp for directive: example

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)

Synchronization: Barrier

```
For(I=0; I<N; I++)
    a[I] = b[I] + c[I];

For(I=0; I<N; I++)
    d[I] = a[I] + b[I]
```

Both loops are in parallel region
With no synchronization in between.
What is the problem?

Fix:

```
For(I=0; I<N; I++)
    a[I] = b[I] + c[I];

#pragma omp barrier

For(I=0; I<N; I++)
    d[I] = a[I] + b[I]
```

Critical session

```
For(I=0; I<N; I++) {
   ……
   sum += A[I];
   ……
}
```

Cannot be parallelized if sum is shared.

Fix:

```
For(I=0; I<N; I++) {
   ……
   #pragma omp critical
  {
     sum += A[I];
   }
   ……
}
```

OpenMP environment variables

- OMP_NUM_THREADS
  - Specifies the number of threads to use for parallel regions
- OMP_SCHEDULE
  - Specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm
  - Valid options for algorithm are:
    - Auto
    - Dynamic [, n]
    - Guided [, n]
    - Runtime
    - Static [, n]
  - n is chunk size
  - Default is auto

OpenMP runtime environment

- omp_get_num_threads
  - Returns number of threads currently executing
- omp_get_thread_num
  - Returns the number of the currently executing thread
- omp_in_parallel
  - Returns .TRUE. if called in parallel region
  - Returns .FALSE. otherwise

Sequential Matrix Multiply

```
For (I=0; I<n; I++)
    for (j=0; j<n; j++)
        c[I][j] = 0;
        for (k=0; k<n; k++)
            c[I][j] = c[I][j] + a[I][k] * b[k][j];
```

OpenMP Matrix Multiply

```
#pragma omp parallel for private(j, k)
For (I=0; I<n; I++)
    for (j=0; j<n; j++)
        c[I][j] = 0;
        for (k=0; k<n; k++)
            c[I][j] = c[I][j] + a[I][k] * b[k][j];
```

Ease of Use

- OpenMP takes cares of the thread maintenance
  - Big improvement over pthread
- Synchronization
  - Much higher constructs (critical section, barrier)
  - Big improvement over pthread

Summary

- OpenMP provides a compact, yet powerful programming model for shared memory programming
  - It is very easy to use OpenMP to create parallel programs.
- OpenMP preserves the sequential version of the program
- Developing an OpenMP program:
  - Start from a sequential program
  - Identify the code segment that takes most of the time
  - Determine whether the important loops can be parallelized
    - The loops may have critical sections, reduction variables, etc
  - Determine the shared and private variables.
  - Add directives