Distributed Systems

Subheadline

- Stefan Henkler **E-Mail: stefan.henkler@hshl.de**

OpenMP

- Explicit thread management is too intrusive
- Explicit thread management is often unnecessary
  - Consider the matrix multiplication
- Solution
  - OpenMP is a nonintrusive framework for parallel programming

What is OpenMP

- Open specifications for multiprocessing via collaborative work between interested parties from the hardware and software industry, government and academia.
- OpenMP is an Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory parallelism.***
  - API components: Compiler Directives, Runtime Library Routines. Environment Variables
- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors

What is OpenMP?

- OpenMP API is specified for C/C++ and Fortran.
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for fortran and pragmas for C/C++.
- OpenMP website: http://www.openmp.org
  - Materials in this lecture are taken from various OpenMP tutorials on the website and other places.

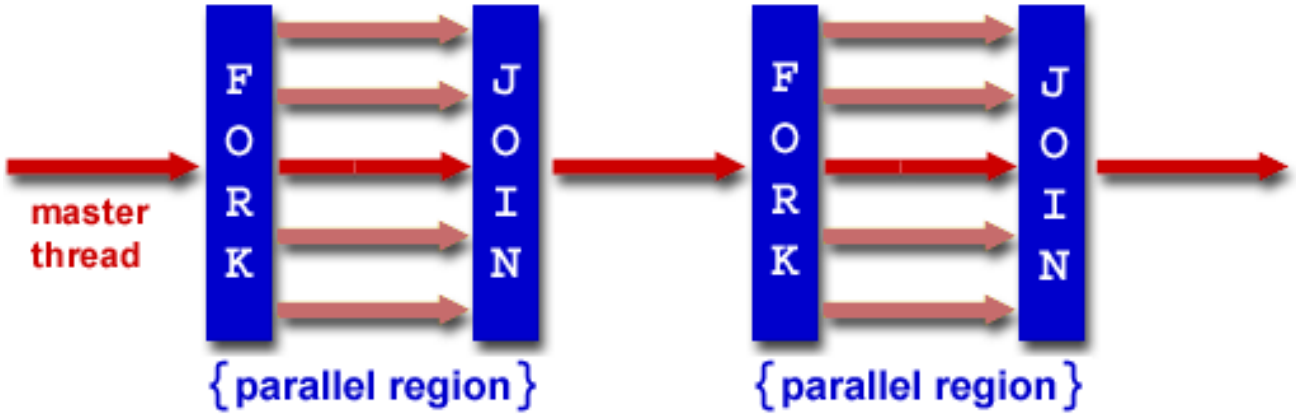
Why OpenMP?

- OpenMP is portable: supported by HP, IBM, Intel, SGI, SUN, and others
  - It is the de facto standard for writing shared memory programs.
- OpenMP can be implemented incrementally

How to compile and run OpenMP programs?

- Gcc 4.2 and above supports OpenMP 3.0
  - gcc –**fopenmp** a.c
- To run: ‘a.out’
  - To change the number of threads:
    - setenv OMP_NUM_THREADS 4 (tcsh) or export OMP_NUM_THREADS=4(bash)

OpenMP execution model

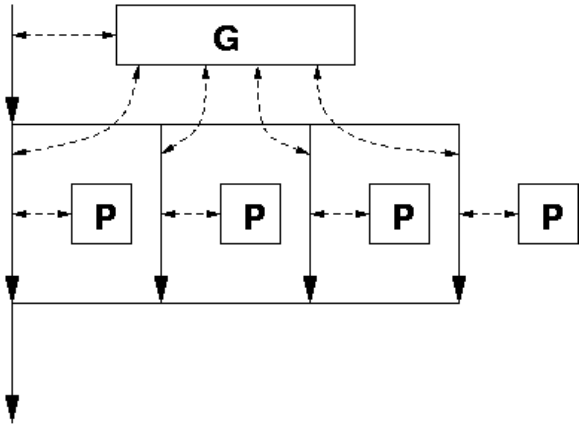


- OpenMP uses the fork-join model of parallel execution.
  - All OpenMP programs begin with a single master thread.
  - The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK).
  - When the team threads complete the parallel region,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



P = private data space
G = global data space

- 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)

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)

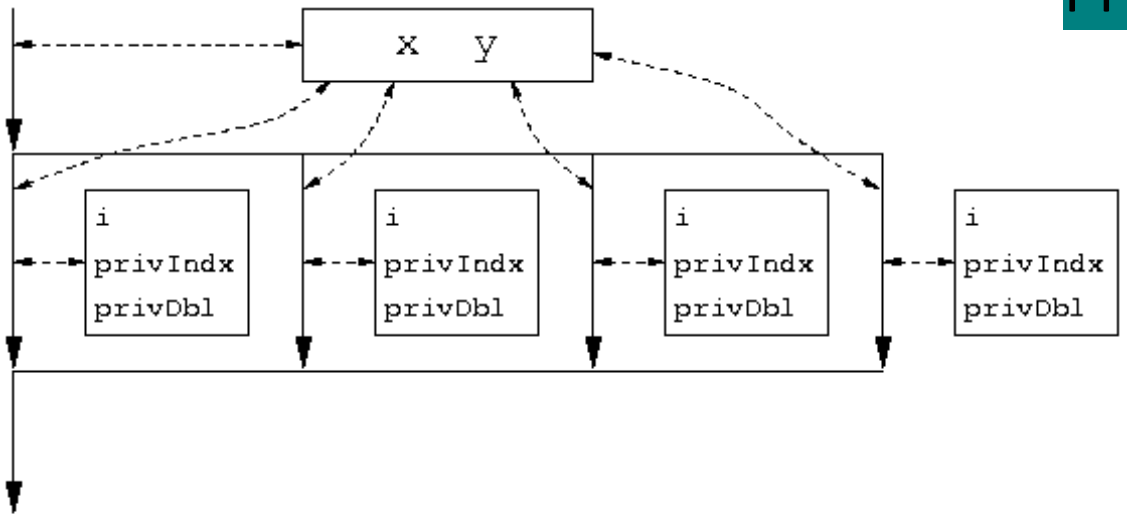
  Clauses: if (expression), private *(list),* shared *(list),* default (shared | none), reduction *(operator: list),* firstprivate*(list),* lastprivate*(list)*

  - 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)

Loop - index

```
#pragma omp parallel for private( privIndx, privDbl )
  for ( i = 0; i < arraySize; i++ ) {
     for ( privIndx = 0; privIndx < 16; privIndx++ ) {
    privDbl = ( (double) privIndx ) / 16;
      y[i] = sin( exp( cos( - exp( sin(x[i]) ) ) ) ) + cos(
    privDbl );
   }
}
```

Parallel for loop index is Private by default.



execution context for "arrayUpdate_II"

Reduction

```
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.

Work-sharing constructs

- #pragma omp for [clause …]
  - … as discussed
- #pragma omp section [clause …]
  - Only one thread per section
- #pragma omp single [clause …]
  - Only one thread executes block.
  - If team of threads: All other ones wait until single block is executed

- The work is distributed over the threads
- Must be enclosed in parallel region

The omp for directive: example

Disable synchronization after for

```
#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)

The omp sections clause - example

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

      #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/

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

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.
Problem: dependencies between both loops
- > race condition!

Add explicit barrier!

```
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 section

```
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
- OMP_SCHEDULE

OpenMP runtime environment

- omp_get_num_threads
- omp_get_thread_num
- omp_in_parallel
- …

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];
```

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