Introduction to OpenMP

- Introduction
- OpenMP basics
- OpenMP directives, clauses, and library routines

What is OpenMP?

- What does OpenMP stands for?
  - Open specifications for Multi Processing 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 shared-memory multiprocessors

What is OpenMP?

- OpenMP API is specified for C/C++ and Fortran.
- OpenMP is not intrusive to the orginal 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 in the website and other places.

Why OpenMP?

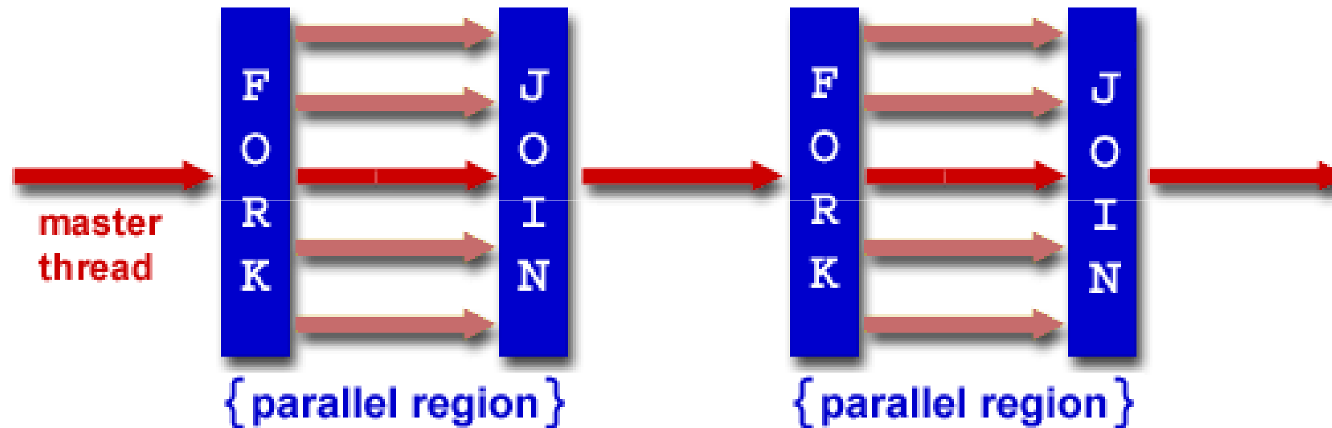
- OpenMP is portable: supported by HP, IBM, Intel, SGI, SUN, and others
  - It is the de facto standard for writing shared memory programs.
- OpenMP can be implemented incrementally, one function or even one loop at a time.
  - A nice way to get a parallel program from a sequential program.

How to compile and run OpenMP programs?

- GNU and Intel compilers support OpenMP.
- To compile OpenMP programs:

  ‘icc example1.c -openmp’
- To run: ‘a.out’
  - In the default setting, the number of threads used is equal to the number of processors in the system.
  - To change the number of threads:
    - export OMP_NUM_THREADS=8 (bash)

OpenMP execution model

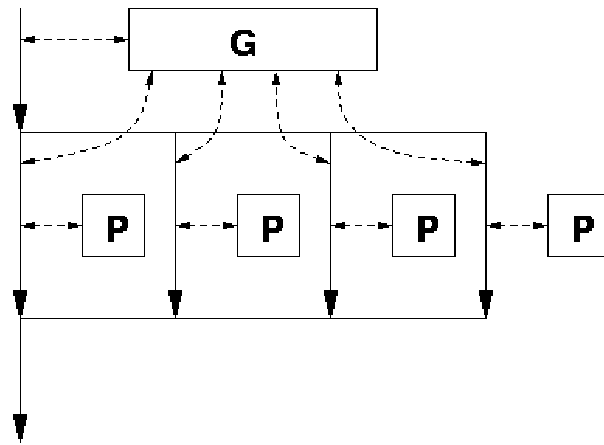


- OpenMP uses the fork-join model of parallel execution.
  - All OpenMP programs begin with a single master thread.
  - The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK).
  - When the team threads complete the parallel region,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

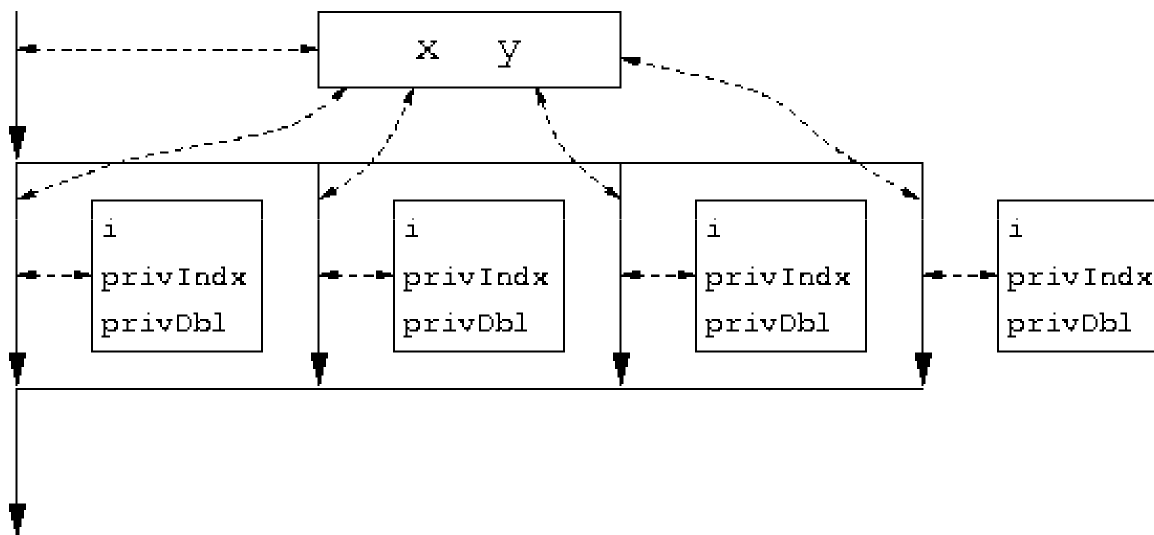
Data model



- 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)

```
#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"

OpenMP directives

- Format:

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

  (use '\' for multiple lines)
- Example:

  #pragma omp parallel default(shared)private(beta,pi)
- Scope of a directive is a block of statements { …}

Parallel region construct

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

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

Example clauses: if (expression), private *(list),* shared *(list),* default (shared | private | 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|private)

- The reduction clause:

```
sum = 0.0;
#pragma parallel shared (n, x) private (I) reduction(+ : sum)
{
  For(I=0; I<n; I++) sum = sum + x(I);
}
```

  - Without the reduction clause, race condition occurs.
  - 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.

Work-sharing constructs

- #pragma omp for [clause …]
- #pragma omp section [clause …]
- #pragma omp single [clause …]

- The work is distributed over the threads
- Must be enclosed in parallel region
- No implied barrier on entry, implied barrier on exit (unless specified otherwise)

The omp for directive: example

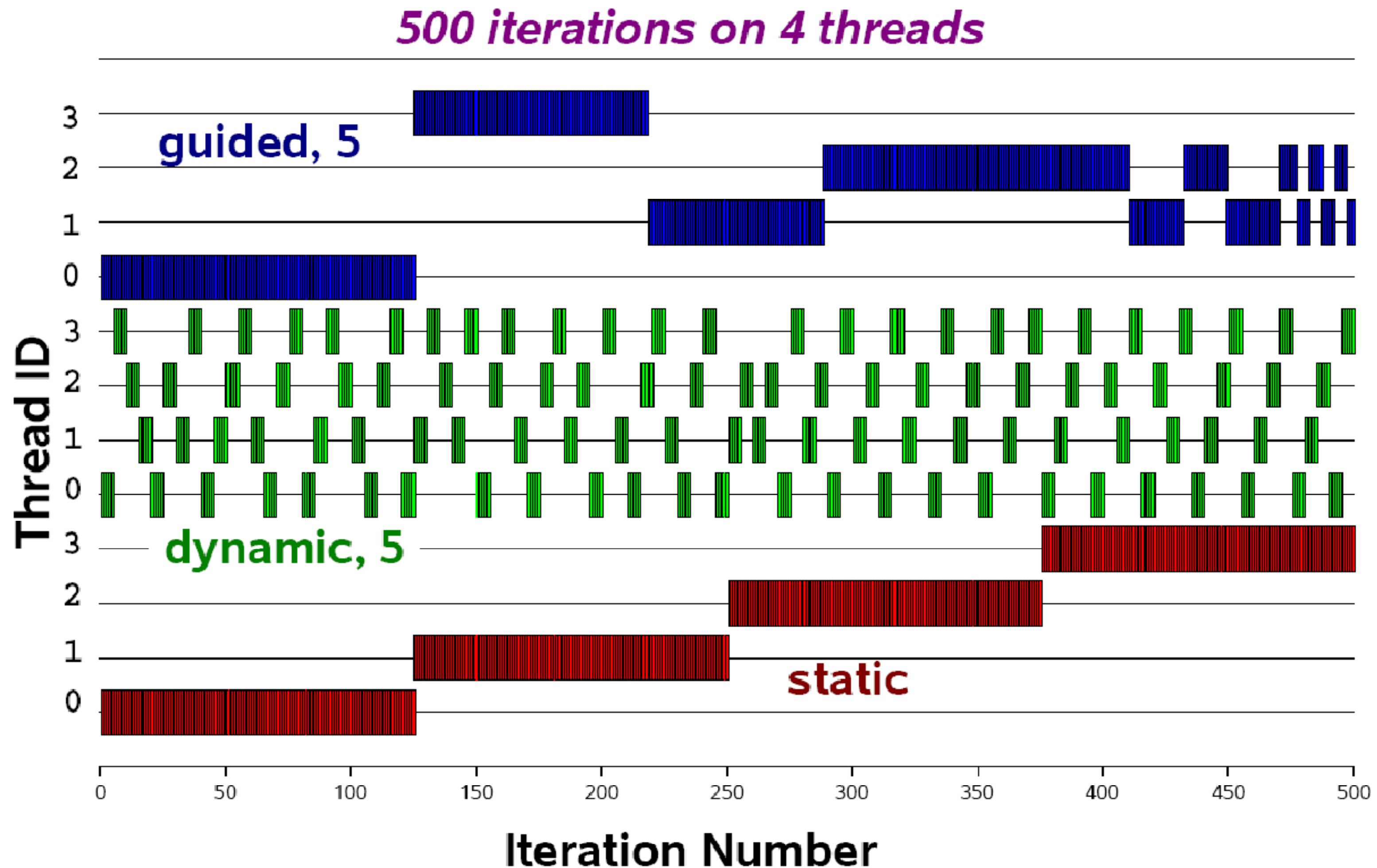
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

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

(implied barrier)

- Schedule clause (decide how the iterations are executed in parallel):

schedule (static | dynamic | guided [, chunk])



The omp session clause - example

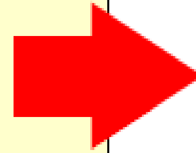
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

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

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

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

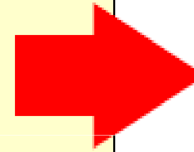
```
#pragma omp parallel
#pragma omp for
   for (...)
```



```
#pragma omp parallel for
for (....)
```

Single PARALLEL loop

```
#pragma omp parallel
#pragma omp sections
{ ...}
```



```
#pragma omp parallel sections
{ ... }
```

Single PARALLEL sections

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
- 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

MPI vs. OpenMP

- Pure MPI Pro:
  - Portable to distributed and shared memory machines.
  - Scales beyond one node
  - No data placement problem
- Pure MPI Con:
  - Difficult to develop and debug
  - High latency, low bandwidth
  - Explicit communication
  - Large granularity
  - Difficult load balancing
- Pure OpenMP Pro:
  - Easy to implement parallelism
  - Low latency, high bandwidth
  - Implicit Communication
  - Coarse and fine granularity
  - Dynamic load balancing
- Pure OpenMP Con:
  - Only on shared memory machines
  - Scale within one node
  - Possible data placement problem
  - No specific thread order

Why Hybrid

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures.
- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes. Good usage of shared memory system resource (memory, latency, and bandwidth).
- Avoids the extra communication overhead with MPI within node.
- OpenMP adds fine granularity (larger message sizes) and allows increased and/or dynamic load balancing.
- Some problems have two-level parallelism naturally.
- Some problems could only use restricted number of MPI tasks.
- Could have better scalability than both pure MPI and pure OpenMP.

A Pseudo Hybrid Code

```
Program hybrid
 call MPI_INIT ()
 call MPI_COMM_RANK (…)
 call MPI_COMM_SIZE (…)
  …  some computation and MPI communication
#PRAGMA OMP PARALLEL DO SHARED(n)
  do i=1,n
        … computation
  enddo
  …  some computation and MPI communication
 call MPI_FINALIZE ()
 end
```