Introduction to OpenMP

Dr. Tassadaq Hussain

RIPHAH
INTERNATIONAL
UNIVERSITY

Microsoft Research
Centre

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

Motivation

- Pthread is too tedious: explicit thread management is often unnecessary
  - Consider the matrix multiply example
    - We have a sequential code, we 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!!!

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 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 in the website and other places.

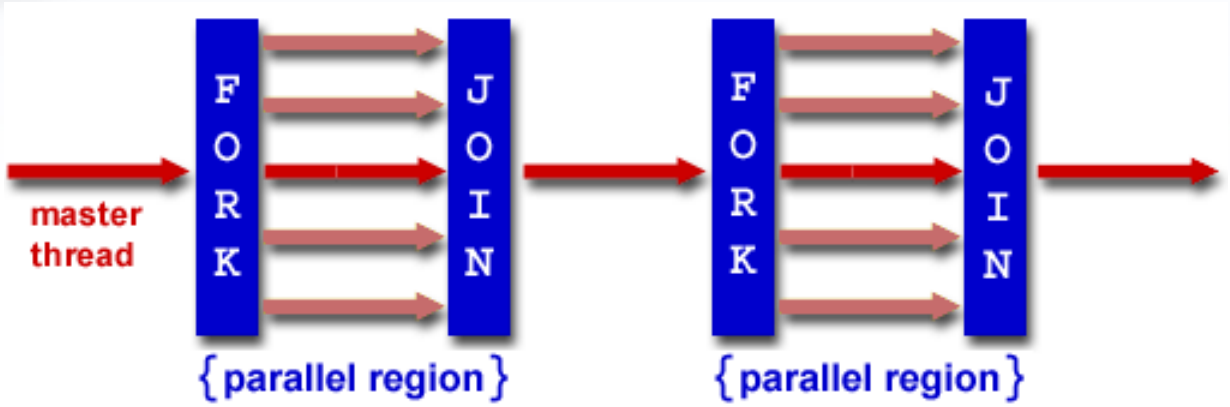
Why OpenMP?

- OpenMP is portable: supported by HP, IBM, Intel, SGI, SUN, and others
  - It is the de facto standard for writing shared memory programs.
  - To become an ANSI standard?
- 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?

- GCC 4.2 and above supports OpenMP 3.0
  - gcc –fopenmp a.c
  - Try example1.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

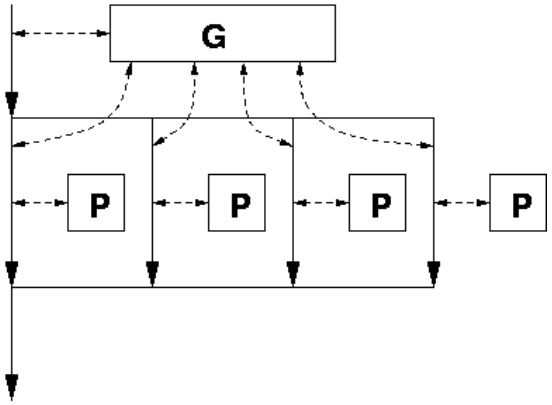


- OpenMP uses the fork-join model of parallel execution.
  - All OpenMP programs begin with a single master thread.
  - The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK).
  - When the team threads complete the parallel region,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

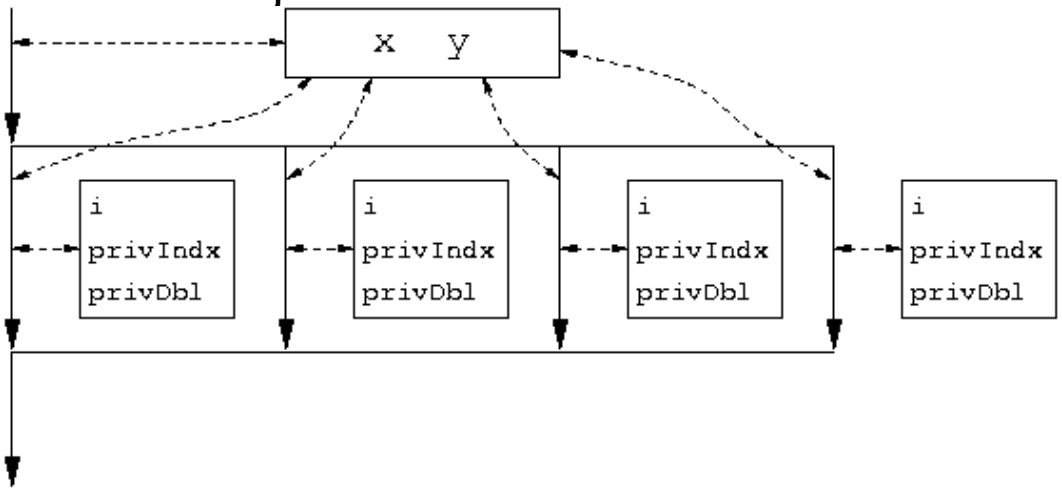
Data model



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)
    - several variations, depending on the initial values and whether the results are copied outside the region.

```
#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 );
  }
}
```



execution context for "arrayUpdate_II"

Parallel for loop index is Private by default.

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 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.

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)

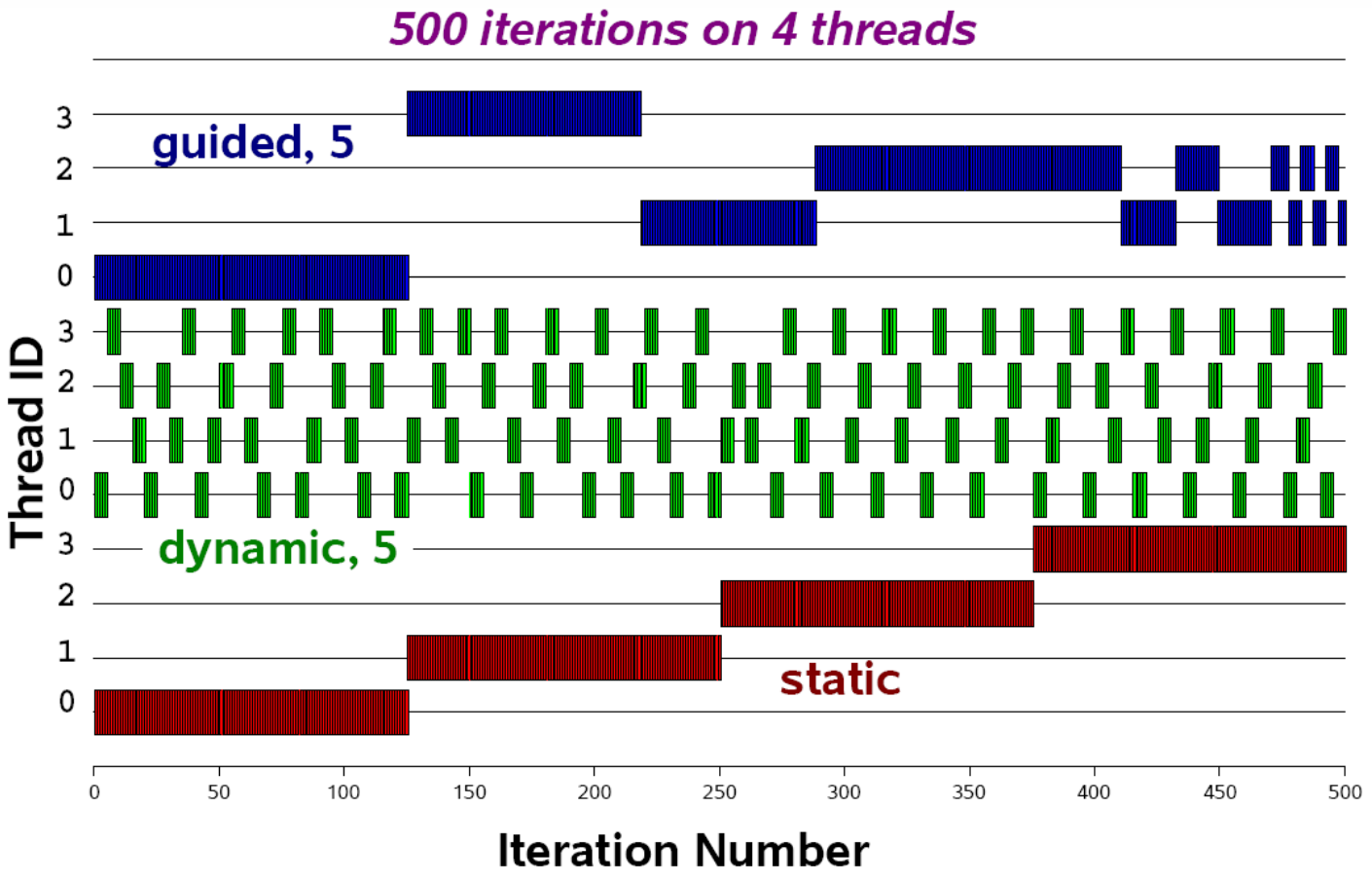
The omp for directive: example

```
#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

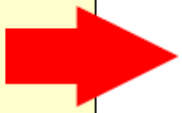
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

      #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
{ ... }
```

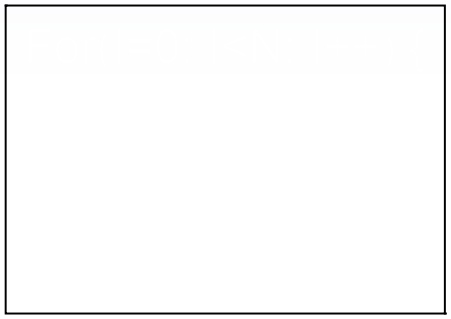
Single PARALLEL sections

Synchronization: barrier

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

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

```
  r(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



OpenMP environment variables

- OMP_NUM_THREADS
- OMP_SCHEDULE

OpenMP runtime environment

- omp_get_num_threads
- omp_get_thread_num
- omp_in_parallel
- Routines related to locks
- ……

OpenMP example

- ee pi.c

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];
```

Travelling Salesman Problem(TSP)

- The map is represented as a graph with nodes representing cities and edges representing the distances between cities.
- A special node (cities) is the starting point of the tour.
- Travelling salesman problem is to find the circle (starting point) that covers all nodes with the smallest distance.
- This is a well known NP-complete problem.

Sequential TSP

```
Init_q(); init_best();
While ((p = dequeue()) != NULL) {
    for each expansion by one city {
        q = addcity (p);
        if (complete(q)) {update_best(q);}
        else enqueue(q);
    }
}
```

OpenMP TSP

```
Do_work() {
  While ((p = dequeue()) != NULL) {
    for each expansion by one city {
        q = addcity (p);
        if (complete(q)) {update_best(q);}
        else enqueue(q);
    }
  }
}

main() {
   init_q(); init_best();
   #pragma omp parallel for
   for (i=0; I < NPROCS; i++)
      do_work();
}
```

Sequential SOR

```
for some number of timesteps/iterations {
  for (i=0; i<n; i++ )
        for( j=1, j<n, j++ )
              temp[i][j] = 0.25 *
                    ( grid[i-1][j] + grid[i+1][j]
                    grid[i][j-1] + grid[i][j+1] );
  for( i=0; i<n; i++ )
        for( j=1; j<n; j++ )
              grid[i][j] = temp[i][j];
}
```

- OpenMP version?

- 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

OpenMP discussion

- 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.
- OpenMP is easy to use!!

OpenMP discussion

- Expressiveness
  - Data parallelism:
    - MM and SOR
    - Fits nicely in the paradigm
  - Task parallelism:
    - TSP
    - Somewhat awkward. Use OpenMP constructs to create threads. OpenMP is not much different from pthread.

OpenMP discussion

- Exposing architecture features (performance):
  - Not much, similar to the pthread approach
    - Assumption: dividing job into threads = improved performance.
    - How valid is this assumption in reality?
      - Overheads, contentions, synchronizations, etc
  - This is one weak point for OpenMP: the performance of an OpenMP program is somewhat hard to understand.

OpenMP final thoughts

- Main issues with OpenMP: performance
  - Is there any obvious way to solve this?
    - Exposing more architecture features?
  - Is the performance issue more related to the fundamantal way that we write parallel program?
    - OpenMP programs begin with sequential programs.
    - May need to find a new way to write efficient parallel programs in order to really solve the problem.