Open MP

Lecture Objectives:

1) Explain the difference between OpenMP and pthreads
2) Compile and link a simple OpenMP Program
3) Explain the usage of the #pragma omp directive.
4) Explain the appropriate mechanism to check whether the compiler supports openMP

Lab Discussion: What did we find?

10 minutes is never long enough.

Caching is good.

5 x 5,000,000

5001 x 5000

5,000,000 x 5

pthreads

- General Thread mechanism
  - Designed really before multicore processors were common
  - Tends to try and keep all threads of the same process on the same processor
    - Why? => Better mem access & Caching
  - Work great for systems where there is a lot of autonomy in the threads
    - Web server
    - UI
    - Etc.
- Not ideal for multicore
  - Why

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

Why 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.

#pragma

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?
    - Already supported by gcc (version 4.2 and up)
- OpenMP can be implemented incrementally, one function or even one loop at a time.
  - Very 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
- To run: 'a.out'
  - To change the number of threads:
    - setenv OMP_NUM_THREADS 4 (tcsh)
    - export OMP_NUM_THREADS=4(bash)

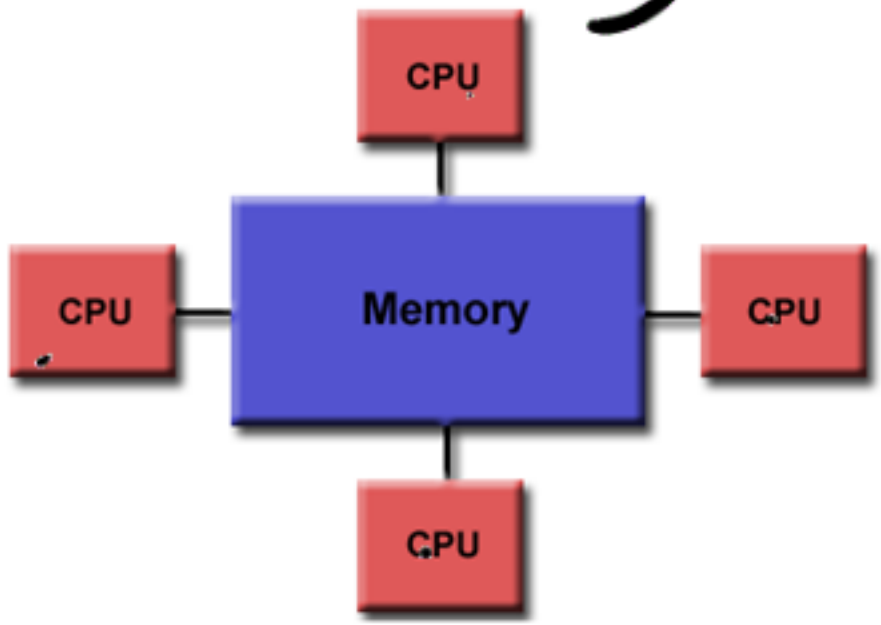
compile

Ties in the open mp library

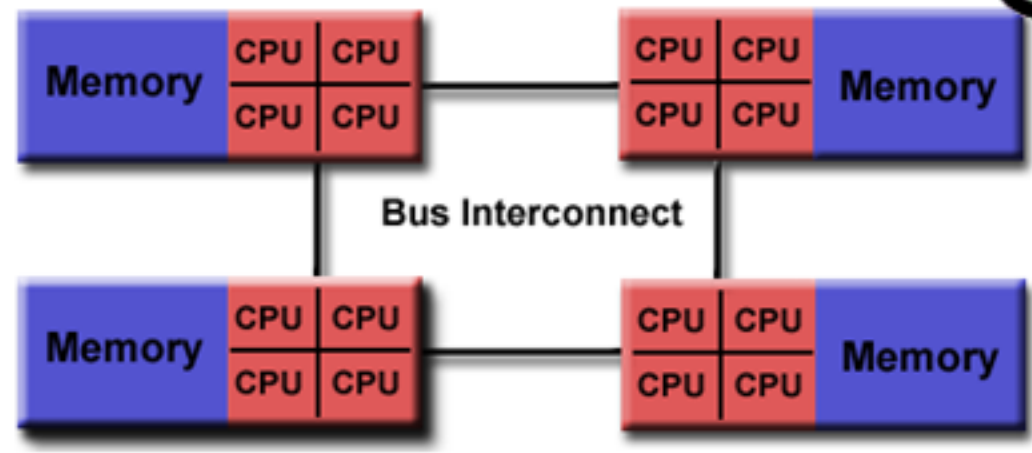
How many threads in parallel?

- Designed for multi-processor/core, shared memory machines.

OpenMP

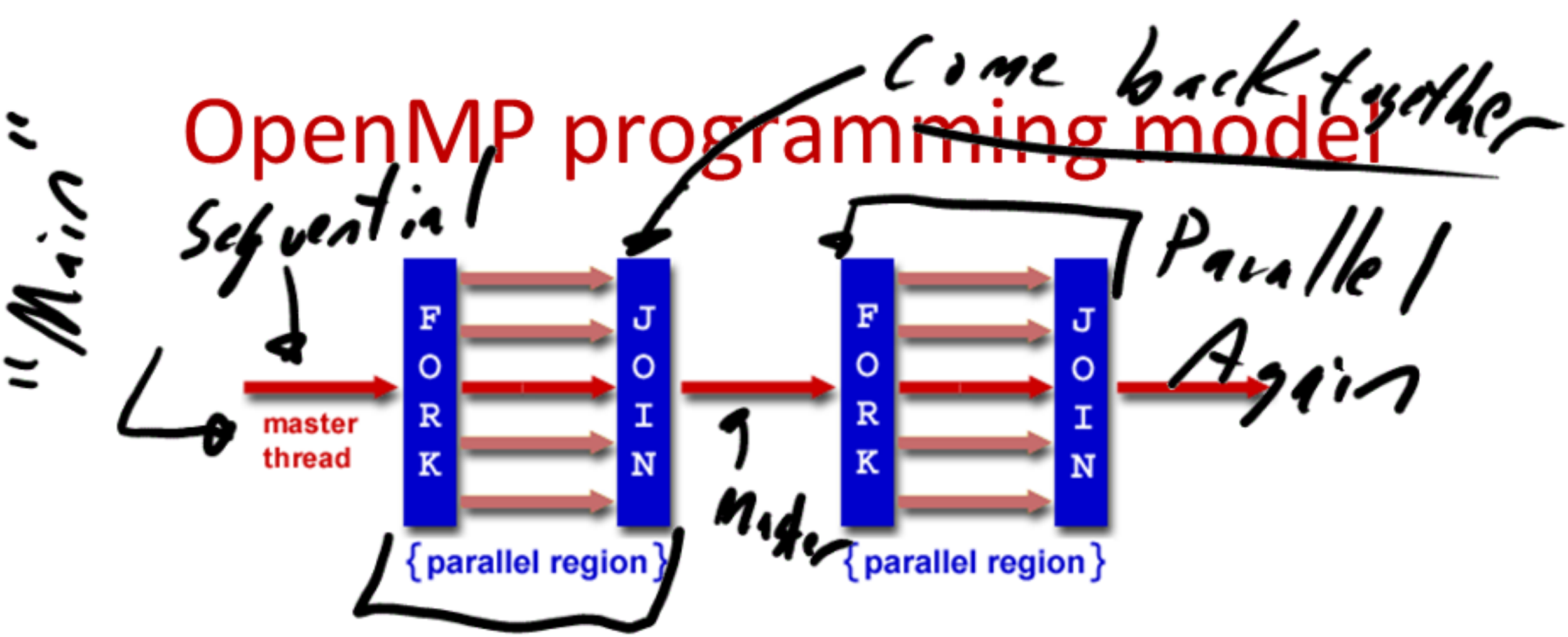


Uniform Memory Access



Non-Uniform Memory Access

OpenMP programming 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).

Definitions

- Team
  - A set of threads executing a program
- Master
  - The original thread which is running and spawns other threads
- Slave
  - A thread spawned by a master thread to solve a parallel segment of code
- Implicit barrier
  - A synchronization construct which ensures that the program does not continue until all slave threads have completed.

A first openMP Program

openMPDemo1.c

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #ifdef _OPENMP
4   #include <omp.h>
5   #endif
6
7   // This function will say hello twice (in honor of the Beetles).
8   void sayHelloHello()
9   {
10      #ifdef _OPENMP
11      int my_rank = omp_get_thread_num();      // Get the slave threads rank amongst all threads.
12      int thread_count = omp_get_num_threads(); // Determine how many slave threads there are.
13      #else
14      int my_rank = 0;        // If OpenMP is not supported, assume 1 thread.
15      int thread_count = 1;
16      #endif
17      // Say hello.
18      printf("Hello from parallel thread %d of %d.\n", my_rank, thread_count);
19      printf("Hello again from parallel thread %d of %d.\n", my_rank, thread_count);
20  }
21
```

A first openMP Program

```
22  // The following code will say hello from each OpenMP thread in the system.
23  int main(int argc, char*argv[])
24  {
25    int index;
26
27    int threadCount = 1;
28
29    if (argc ==2)
30      {
31        // Get the number of threads from the command line
32        threadCount = strtol(argv[1], NULL, 10);
33      }
34
35  #pragma omp parallel num_threads(threadCount)
36    sayHelloHello();
37  }
```

I would like threadCount threads for the next block.

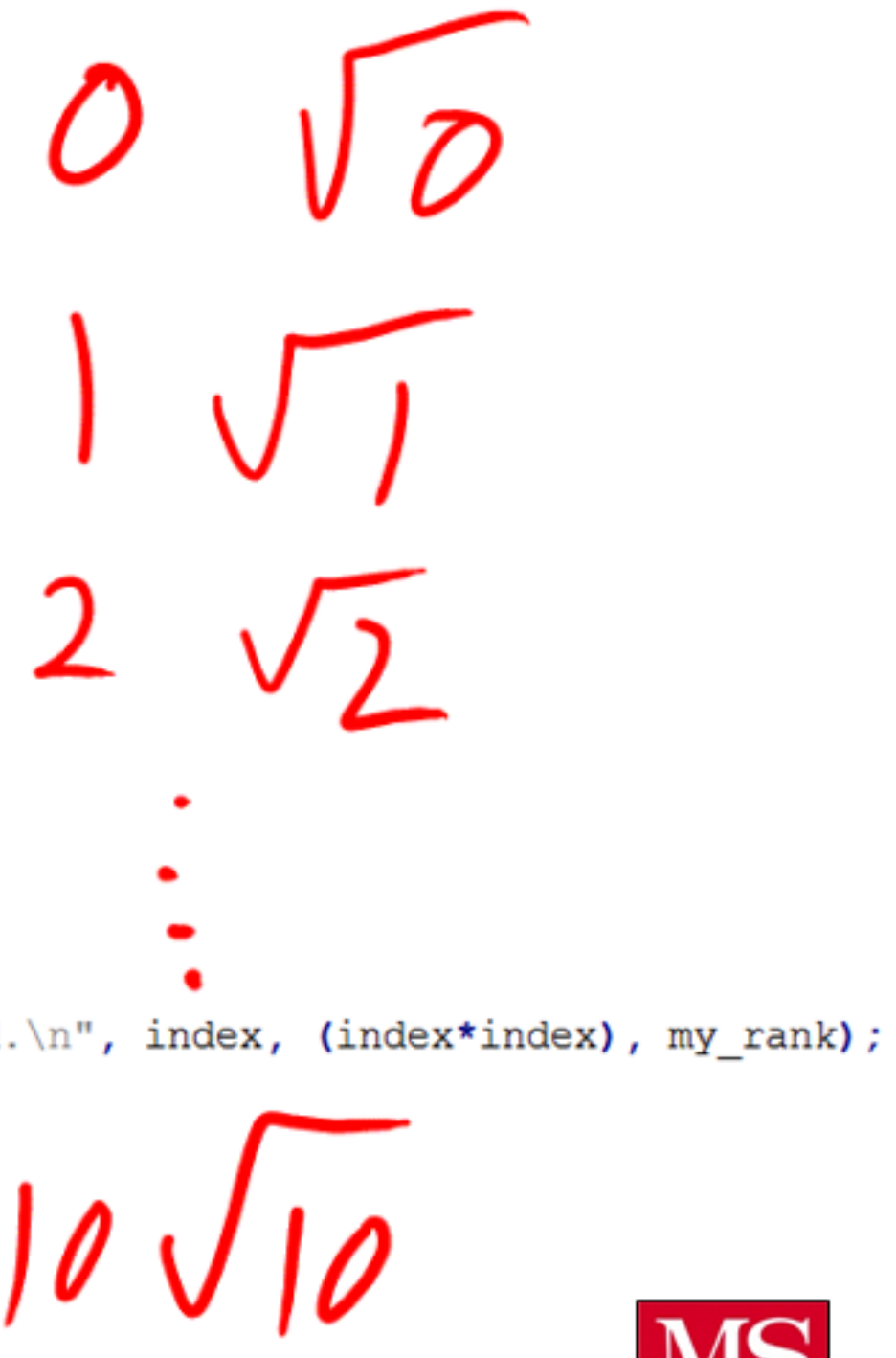
Parallel

#pragma ⇒ Instruction to the compiler. omp command follows.

- Parallel directive
  - The structured block which follows the directive is to execute in parallel
  - The structured block may be a single statement, a set of bracketed code, or any other block of code
  - The block will execute on each parallel thread

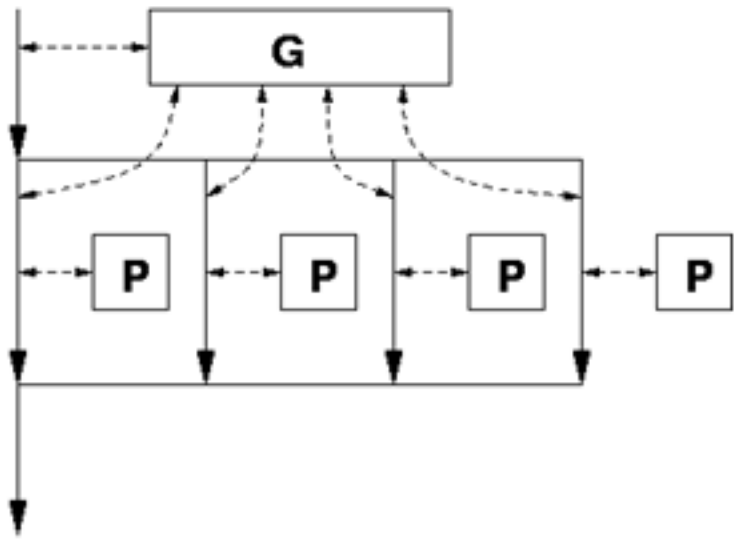
• Here's an example with a loop

```
6  // The following code will print out the numbers between 0 and 10 and the squares of
7  // those numbers before exiting.
8  // It will use openMP to do this in parallel.
9  int main(int argc, char*argv[])
10 {
11   int index;
12   int threadCount = 1;
13   if (argc ==2)
14     {
15       // Get the number of threads from the command line.
16       threadCount = strtol(argv[1], NULL, 10);
17     }
18 #pragma omp parallel num_threads(threadCount)
19   for (index = 0; index < 10; index++)
20   {
21     // Only call the API if the system is using OPENMP.
22     #ifdef _OPENMP
23     int my_rank = omp_get_thread_num();
24     #else
25     int my_rank = 0;
26     #endif
27     printf("Number: %d\tSqrt: %Lf Calculated by thread %d.\n", index, (index*index), my_rank);
28   }
29 }
```



Parallel versus parallel for

Data model



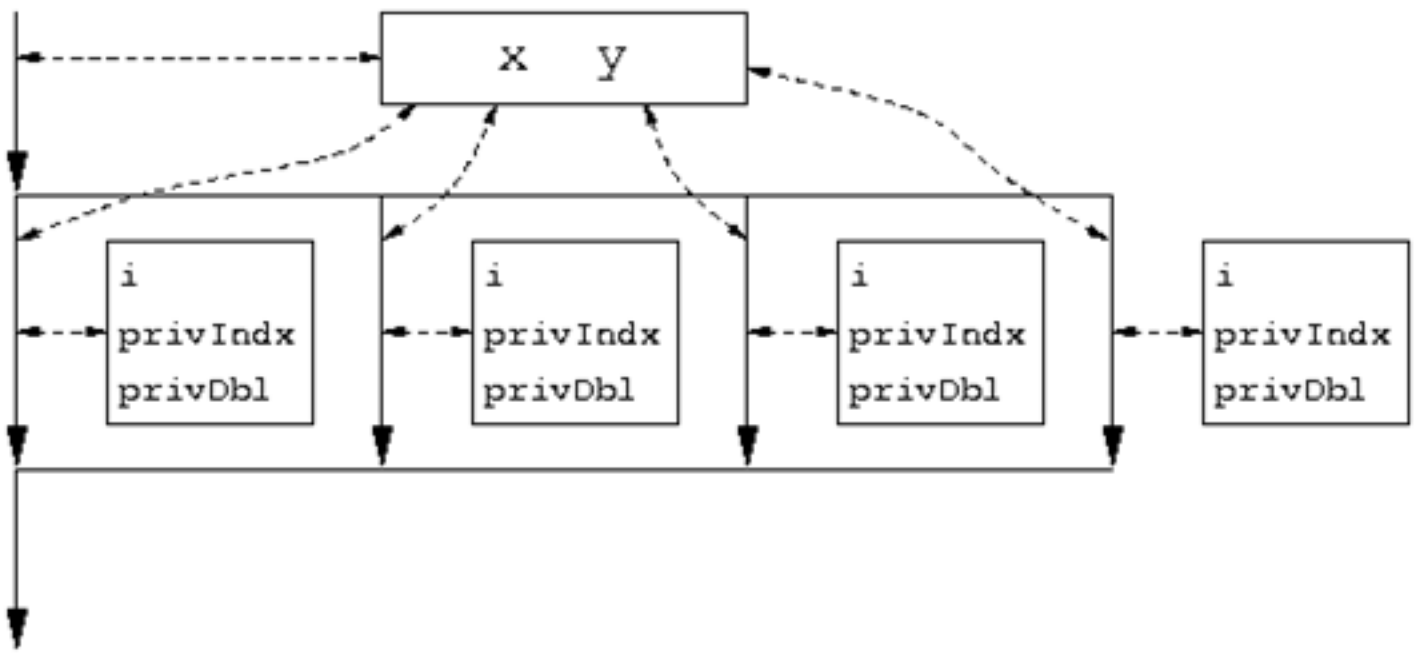
P = private data space
G = global data space

- 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.

MSOE

```
#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"

- When can we mark a loop a parallel loop?
  - How should we declare variables shared or private?

```
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 loop: executing each iteration concurrently is the same as executing each iteration sequentially.

- no loop carry dependencies: an iteration does not produce any data that will be consumed by another iteration.
  - y[i] is different for each iteration. privDbl is not (must make it private to be correct).

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 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 | 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.
  - See example3.c and example3a.c

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)

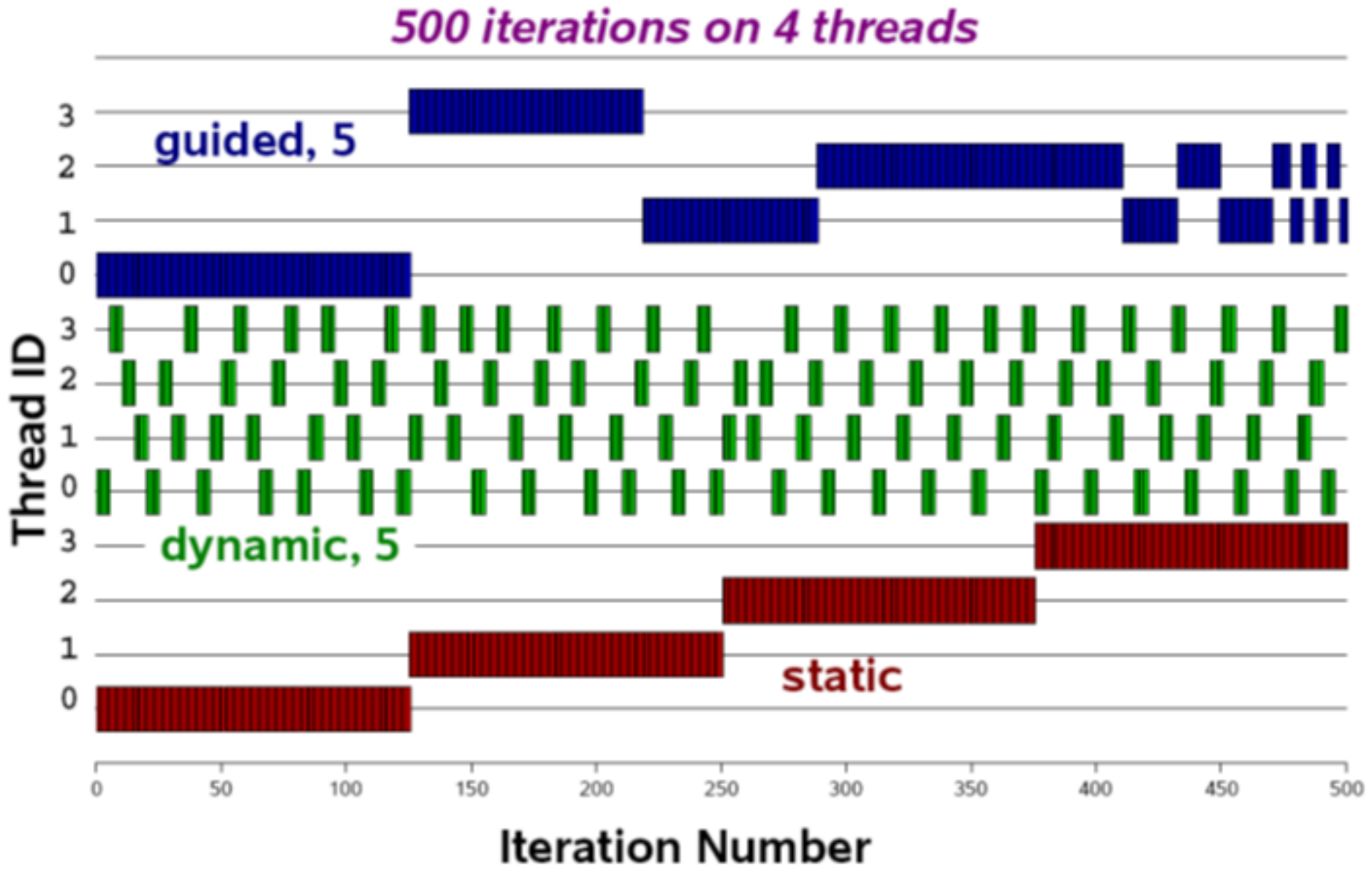
The omp for directive: example

```
#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 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
- Routines related to locks
- ……

Lock related routines

- Will only discuss simple lock: may not be locked if already in a locked state.
- Simple lock interface:
  - Type: omp_lock_t
  - Operations:
    - omp_init_lock(omp_lock_t *a)
    - omp_destroy_lock(omp_lock_t *a)
    - omp_set_lock(omp_lock_t *a)
    - omp_unset_lock(omp_lock_t *a)
    - omp_test_lock(omp_lock_t *a)

Openmp lock routines

- omp_init_lock initializes the lock. After the call, the lock is unset.
- omp_destroy_lock destroys the lock. The lock must be unset before this call.
- omp_set_lock attempts to set the lock. If the lock is already set by another thread, it will wait until the lock is no longer set, and then sets it.
- omp_unset_lock unsets the lock. It should only be called by the same thread that set the lock; the consequences of doing otherwise are undefined.
- omp_test_lock attempts to set the lock. If the lock is already set by another thread, it returns 0; if it managed to set the lock, it returns 1.

Openmp lock routines

- Can the lock mechanism used for loop carried dependence?
- See loopcarry_omp.c and loopcarry_omp_final.c

Realizing a customized reduction

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

#pragma omp parallel default (none) shared(n, x, localsum, nthreads)
      private(I)
{
 nthreads = omp_get_num_threads();
#pragma omp for
 for (I=0; I<n; I++) {
   localsum[omp_get_thread_num()] += x(I);
}

For (I=0; I<nthreads; I++) sum += localsum[I];
```

- Summary:
  - OpenMP provides a compact, yet powerful programming model for shared memory programming
  - 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.
    - See for example pi.c and piomp.c program.

- Challenges in developing correct openMP programs
  - Dealing with loop carried dependence
  - Removing unnecessary dependencies
  - Managing shared and private variables