OpenMP Tutorial

Seung-Jai Min
(smin@purdue.edu)

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN

Parallel Programming Standards

- Thread Libraries
  - Win32 API / Posix threads
- Compiler Directives
  - OpenMP (Shared memory programming)

    OUR FOCUS
- Message Passing Libraries
  - MPI (Distributed memory programming)

Shared Memory Parallel Programming in the Multi-Core Era

- Desktop and Laptop
  - 2, 4, 8 cores and … ?
- A single node in distributed memory clusters
  - Steele cluster node: 2 → 8 → (16) cores
- Shared memory hardware Accelerators
    - Cell processors: 1 PPE and 8 SPEs
    - Nvidia Quadro GPUs: 128 processing units

OpenMP:
Some syntax details to get us started

- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:

    #pragma omp *construct [clause [clause]…]*
  - For Fortran, the directives take one of the forms:

    C$OMP *construct [clause [clause]…]*

    !$OMP *construct [clause [clause]…]*

    *$OMP *construct [clause [clause]…]*
- Include files

    #include “omp.h”

How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

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];
  }
}
```

How is OpenMP typically used? (Cont.)

- Single Program Multiple Data (SPMD)

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];
  }
}
```

How is OpenMP typically used? (Cont.)

- Single Program Multiple Data (SPMD)

Thread 0

```
void main()
{
  int i, k, N
  double A[N]
  lb = 0;
  ub = 250;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 1

```
void main()
{
  int i, k, N
  double A[N]
  lb = 250;
  ub = 500;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 2

```
void main()
{
  int i, k, N
  double A[N]
  lb = 500;
  ub = 750;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 3

```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  lb = 750;
  ub = 1000;
  for (i=lb;i<ub;i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

OpenMP Fork-and-Join model

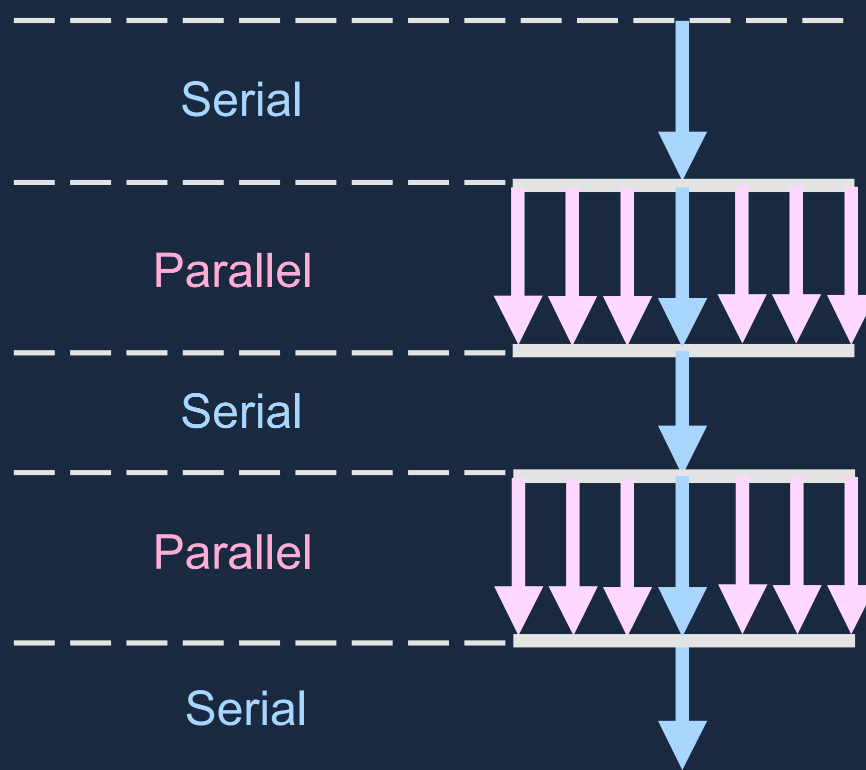
```
printf(“program begin\n”);
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf(“program done\n”);
```



OpenMP Constructs

1. Parallel Regions
   - #pragma omp parallel
2. Worksharing
   - #pragma omp for, #pragma omp sections
3. Data Environment
   - #pragma omp parallel shared/private (…)
4. Synchronization
   - #pragma omp barrier
5. Runtime functions/environment variables
   - int my_thread_id = omp_get_num_threads();
   - omp_set_num_threads(8);

OpenMP: Structured blocks

- Most OpenMP constructs apply to structured blocks.
  - Structured block: one point of entry at the top and one point of exit at the bottom.
  - The only “branches” allowed are STOP statements in Fortran and exit() in C/C++.

OpenMP: Structured blocks

A Structured Block

```
#pragma omp parallel
{
more:  do_big_job(id);
          if(++count>1) goto more;
}
   printf(" All done \n");
```

Not A Structured Block

```
 if(count==1) goto more;
#pragma omp parallel
{
more:  do_big_job(id);
          if(++count>1) goto done;
}
done:     if(!really_done()) goto more;
```

Structured Block Boundaries

- In C/C++: a block is a single statement or a group of statements between brackets {}

```
#pragma omp parallel
{
  id = omp_thread_num();
  A[id] = big_compute(id);
}
```

```
#pragma omp for
for (I=0;I<N;I++) {
   res[I] = big_calc(I);
   A[I] = B[I] + res[I];
}
```

Structured Block Boundaries

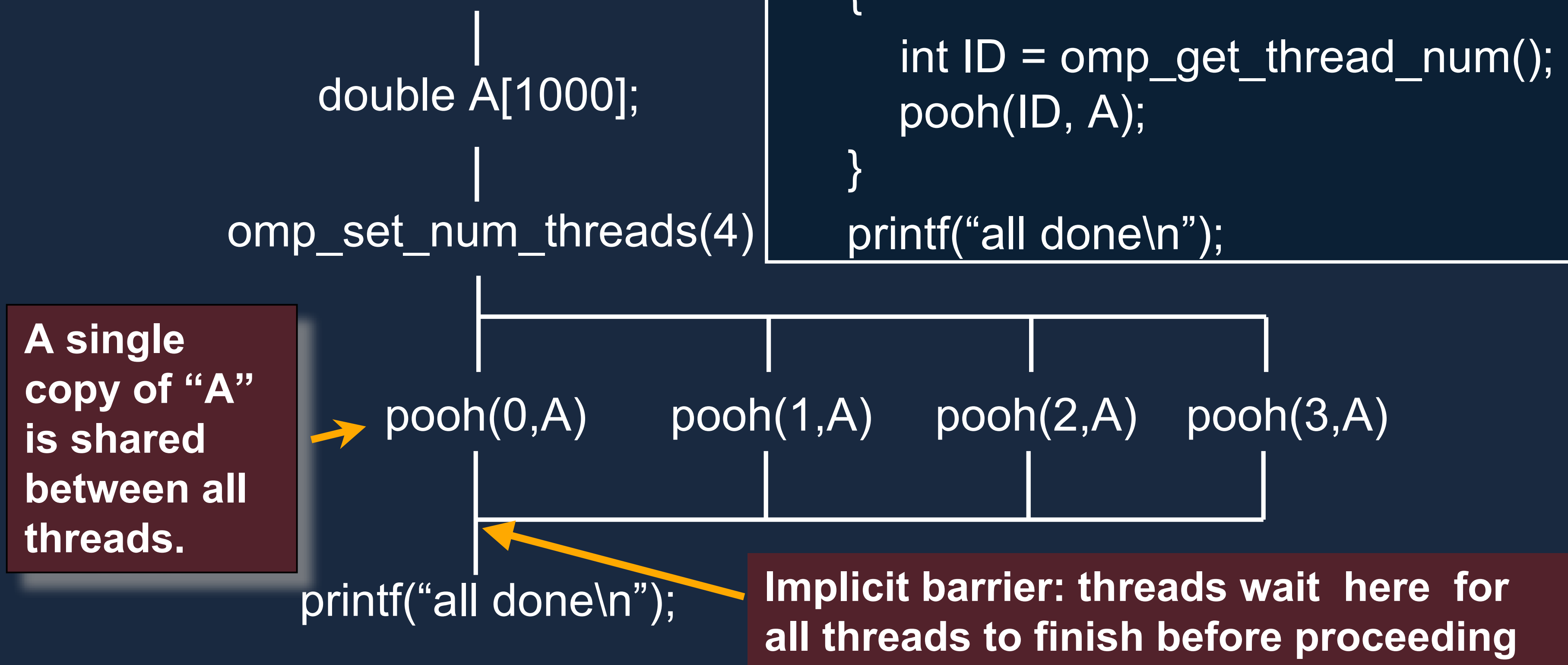
- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```
C$OMP PARALLEL
10    W(id) = garbage(id)
      res(id) = W(id)**2
      if(res(id) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
      do I=1,N
        res(I)=bigComp(I)
      end do
C$OMP END PARALLEL DO
```

OpenMP Parallel Regions

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



The OpenMP API
Combined parallel work-share

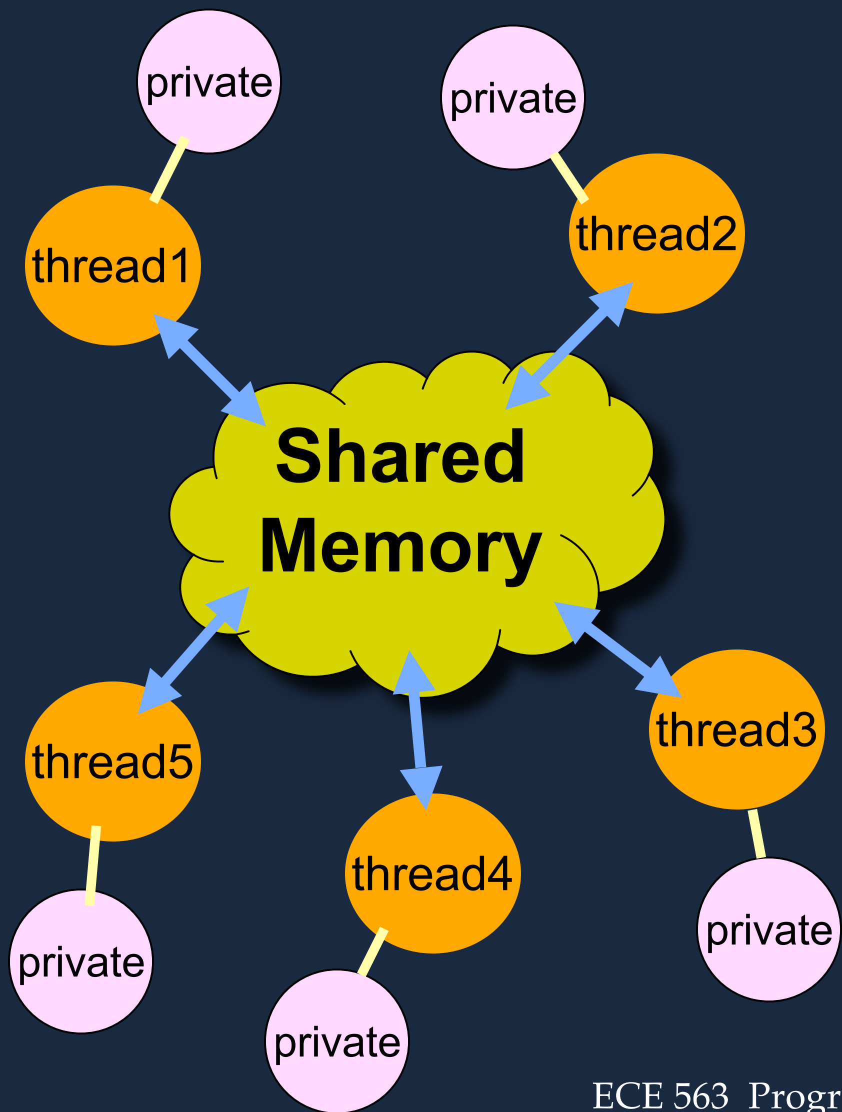
- OpenMP shortcut: Put the "parallel" and the work-share on the same line

```
int i;
double  res[MAX];
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i< MAX; i++) {
    res[i] = huge();
  }
}
```

```
int i;
double  res[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
  res[i] = huge();
}
```

the same OpenMP

Shared Memory Model



- Data can be shared or private
- Shared data is accessible by all threads
- Private data can be accessed only by the thread that owns it
- Data transfer is transparent to the programmer

Data Environment:
Default storage attributes

- Shared Memory programming model
  - Variables are shared by default
- Distributed Memory Programming Model
  - All variables are private

Data Environment:
Default storage attributes

- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

Data Environment

```
int foo(int x)
{
  /* PRIVATE */
  int count=0;
  return x*count;
}
```

```
int A[100]; /* (Global) SHARED */

int main()
{
  int ii, jj;     /* PRIVATE */
  int B[100];  /* SHARED */
#pragma omp parallel private(jj)
  {
     int kk = 1;      /* PRIVATE */
#pragma omp for
     for (ii=0; ii<N; ii++)
        for (jj=0; jj<N; jj++)
           A[ii][jj] = foo(B[ii][jj]);
  }
}
```

Work Sharing Construct

Loop Construct

```
#pragma omp for [clause[[,] clause …] new-line
    for-loops
```

Where clause is one of the following:

- private / firstprivate / lastprivate(list)
- reduction(operator: list)
- schedule(kind[, chunk_size])
- collapse(n)
- ordered
- nowait

Schedule

```
for (i=0; i<1100; i++)
   A[i] = … ;
```

#pragma omp parallel for schedule (static, 250) or (static)

250	250	250	250	100
p0	p1	p2	p3	p0

or

275	275	275	275
p0	p1	p2	p3

#pragma omp parallel for schedule (dynamic, 200)

200	200	200	200	200	100
p3	p0	p2	p3	p1	p0

#pragma omp parallel for schedule (guided, 100)

137	120	105	100	100	100	100	100	100	100	38
p0	p3	p0	p1	p2	p3	p0	p1	p2	p3	p0

#pragma omp parallel for schedule (auto)

Critical Construct

```
sum = 0;
#pragma omp parallel private (lsum)
{
   lsum = 0;
   #pragma omp for
   for (i=0; i<N; i++) {
      lsum = lsum + A[i];
   }
   #pragma omp critical
   { sum += lsum; }
}
```

Threads wait their turn; only one thread at a time executes the critical section

Reduction Clause

Shared variable

```
sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++)
{
   sum = sum + A[i];
}
```

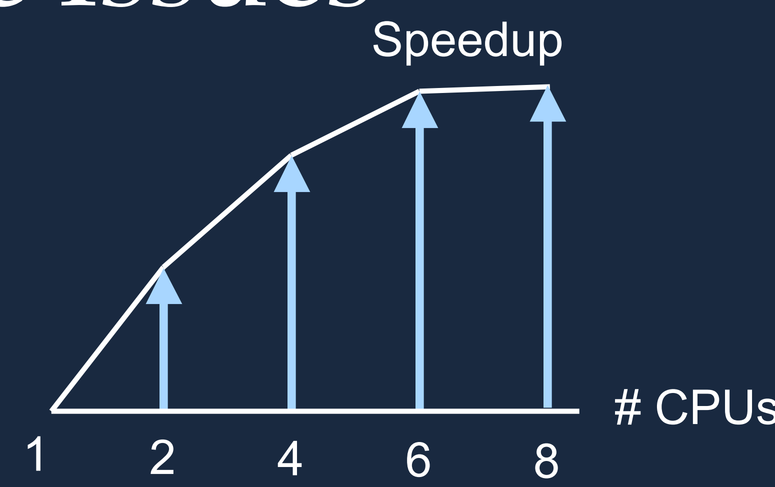
Performance Evaluation

- How do we measure performance? (or how do we remove noise?)

```
#define N 24000
For (k=0; k<10; k++)
{
#pragma omp parallel for private(i, j)
for (i=1; i<N-1; i++)
   for (j=1; j<N-1; j++)
      a[i][j] = (b[i][j-1]+b[i][j+1])/2.0;
}
```

Performance Issues

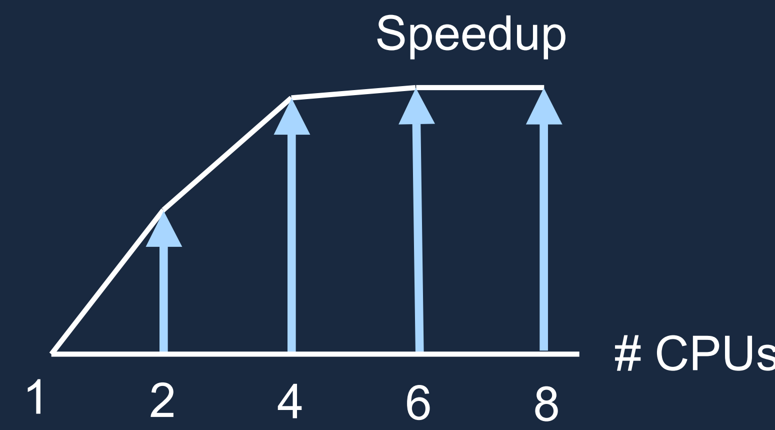
- What if you see a speedup saturation?




```
#define N 12000
#pragma omp parallel for private(j)
for (i=1; i<N-1; i++)
   for (j=1; j<N-1; j++)
      a[i][j] = (b[i][j-1]+b[i][j]+b[i][j+1]
                 b[i-1][j]+b[i+1][j])/5.0;
```

Performance Issues

- What if you see a speedup saturation?




```
#define N 12000
#pragma omp parallel for private(j)
for (i=1; i<N-1; i++)
   for (j=1; j<N-1; j++)
      a[i][j] = b[i][j];
```

Loop Scheduling

- Any guideline for a chunk size?

```
#define N <big-number>

chunk = ???;
#pragma omp parallel for schedule (static, chunk)
for (i=1; i<N-1; i++)
  a[i][j] = ( b[i-2] + b[i-1] + b[i]
              b[i+1] + b[i+2] )/5.0;
```

Performance Issues

- Load imbalance: triangular access pattern

```
#define N 12000
#pragma omp parallel for private(j)
for (i=1; i<N-1; i++)
   for (j=i; j<N-1; j++)
      a[i][j] = (b[i][j-1]+b[i][j]+b[i][j+1]
                b[i-1][j]+b[i+1][j])/5.0;
```

Summary

- OpenMP has advantages
  - Incremental parallelization
  - Compared to MPI
    - No data partitioning
    - No communication scheduling

Resources

http://www.openmp.org

http://openmp.org/wp/resources