High Performance Parallel Programming (CS61064)

Week – 3
Part 1

Pralay Mitra

Work-sharing Constructs

- **Situation 1**
  - **arr[i]=*f*(arr[i-1])**

Data Dependency

or

  - **arr[i]=arr[i-1]+2*x;**

- **Situation 2**
  - **x=y+z**
  - **d=b*c**
  - **s=u*t+(f*t*t)/2**
  - **f=ma**

Work-sharing Constructs

- **sections Construct**
  - **#pragma omp sections** *[clause[[,] clause] ...] new-line*
    {
    *[***#pragma omp section** *new-line*
    structured-block]
    *[***#pragma omp section** *new-line*
    structured-block]
    ...
    }

- Clause

- **private(***variable-list***)**
- **firstprivate(***variable-list***)**
- **lastprivate(***variable-list***)**
- **reduction(***operator***:** *variable-list***)**
- **nowait**

Work-sharing Constructs

- **single Construct**
  - **#pragma omp single** *[clause[[,] clause] ...] new-line*
    structured-block
  - **Clause**
    - **private(***variable-list***)**
    - **firstprivate(***variable-list***)**
    - **copyprivate(***variable-list***)**
    - **nowait**

The SINGLE construct allows code that is serial in nature to be executed inside a parallel region. The thread executing the code will be the first to reach the directive in the code. It doesn't have to be the master thread. All other threads proceed to the end of the structured block where there is an implicit synchronization.

Master construct

#pragma omp master
structured-block

Same as *single nowait* but only for master thread

Work-sharing Constructs

- **for Construct**
  - **#pragma omp for** *[clause[[,] clause] ...] new-line*
    for-loop
  - ***Clause***
    - **private(***variable-list***)**
    - **firstprivate(***variable-list***)**
    - **lastprivate(***variable-list***)**
    - **reduction(***operator*: *variable-list***)**
    - **ordered**
    - **schedule(***kind[, chunk_size]***)**
    - **nowait**

OpenMP work-sharing constructs

Functionality	Syntax in C/C++	Syntax in Fortran
Distribute iterations over the threads	#pragma omp for	!$omp do
Distribute independent work units	#pragma omp sections	!$omp sections
Only one thread executes the code block	#pragma omp single	!$omp single
Parallelize array-syntax		!$omp workshare

Example -5

```
#pragma omp parallel shared(n) private(i)
{
      #pragma omp for
       for(i=0; i<n; i++)
         printf("Thread %d executes loop iteration %d\n",
               omp_get_thread_num(),i);
}
```

Work sharing loops

```
Thread 7 executes loop iteration 9
Thread 3 executes loop iteration 5
Thread 6 executes loop iteration 8
Thread 2 executes loop iteration 4
Thread 4 executes loop iteration 6
Thread 5 executes loop iteration 7
Thread 1 executes loop iteration 2
Thread 1 executes loop iteration 3
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
```

Example -6

```
#pragma omp parallel shared(n,a,b) private(i)
{
      #pragma omp for
       for(i=0; i<n; i++)
         a[i]=i;

      #pragma omp for
       for(i=0; i<n; i++)
         b[i]=2*a[i];
}
```

Two work sharing loops in one parallel region

Work-sharing Constructs

– **for (***init-expr***;** *var logical-op b***;** *incr-expr***)**

- **init-expr/incr-expr: same as C**
- **var:** A signed integer variable. If this variable would otherwise be shared, it is implicitly made private for the duration of the for. This variable must not be modified within the body of the for statement. Unless the variable is specified lastprivate, its value after the loop is indeterminate.
- **logical-op:** >, <, >=, <=
- ***b*, and *incr:*** Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

Combined Parallel Work-sharing Constructs

- **parallel for Construct**
  - **#pragma omp parallel for** *[clause[[,] clause] ...] new-line*
    - *for-loop*

Loop construct: Scheduling

#pragma omp for schedule(kind[, chunk_size])
for-loops

Static
- iterations are divided into chunks of size chunk_size
- the chunks are assigned to the threads in a round-robin fashion
- must be reproducible within the same parallel region

Dynamic
- iterations are divided into chunks of size chunk_size
- the chunks are assigned to the threads as they request them
- the default chunk_size is 1

Guided
- iterations are divided into chunks of decreasing size
- the chunks are assigned to the threads as they request them
- chunk_size controls the minimum size of the chunks

Run-time
- controlled by environment variables

Example -7

#pragma omp parallel for schedule(kind [,chunk size])

```
#pragma omp parallel for num_threads(THREADS)
 for (i = 0; i < N; i++) {
   printf("Thread %d is doing iteration %d.\n", omp_get_thread_num( ), i);
 }
```

```
#pragma omp parallel for schedule(static) num_threads(THREADS)
 for (i = 0; i < N; i++) {
            sleep(i);   /* wait for i seconds */

   printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
 }
```

Best/Worst case runtime of this program?

Example -8

```
#pragma omp parallel for schedule(static) num_threads(THREADS)
 for (i = 0; i < N; i++) {
            sleep(i);   /* wait for i seconds */

   printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
 }
```

```
#pragma omp parallel for schedule(dynamic) num_threads(THREADS)
 for (i = 0; i < N; i++) {
            sleep(i);    /* wait for i seconds */

   printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
 }
```

Example -8

```
Static Schedule....
Thread 0 has completed iteration 0.
Thread 0 has completed iteration 1.
Thread 1 has completed iteration 2.
Thread 2 has completed iteration 4.
Thread 3 has completed iteration 5.
Thread 1 has completed iteration 3.
Thread 4 has completed iteration 6.
Thread 5 has completed iteration 7.
Thread 6 has completed iteration 8.
Thread 7 has completed iteration 9.


Dynamic Schedule....
Thread 1 has completed iteration 0.
Thread 7 has completed iteration 1.
Thread 0 has completed iteration 2.
Thread 3 has completed iteration 3.
Thread 6 has completed iteration 4.
Thread 2 has completed iteration 5.
Thread 5 has completed iteration 6.
Thread 4 has completed iteration 7.
Thread 1 has completed iteration 8.
Thread 7 has completed iteration 9.
```

Overhead?

Example -8

```
#pragma omp parallel for schedule(static) num_threads(THREADS)
  for (i = 0; i < N; i++) {

}
printf("Out of loop!\n");
```

```
#pragma omp parallel for schedule(dynamic) num_threads(THREADS)
  for (i = 0; i < N; i++) {

}
printf("Out of loop!\n");
```

Example -9

```
#pragma omp parallel for schedule(static,chunk) num_threads(THREADS)
  for (i = 0; i < N; i++) {

}
printf("Out of loop!\n");
```

```
#pragma omp parallel for schedule(dynamic,chunk) num_threads(THREADS)
  for (i = 0; i < N; i++) {

}
printf("Out of loop!\n");
```

Example -10

```
#pragma omp parallel for schedule(guided) num_threads(THREADS)
  for (i = 0; i < N; i++) {

}
printf("Out of loop!\n");
```

This scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced.

Loop Scheduling

#pragma omp parallel for schedule(kind [,chunk size])

Kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads. Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads.
auto	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
runtime	Uses the OMP_schedule environment variable to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

Question???

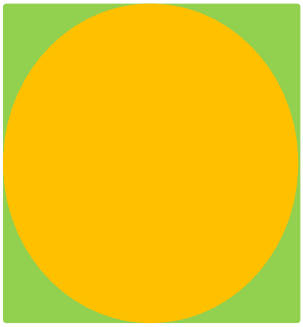
- Is the static kind optimum?

Home work

Compute the value of Pi (accuracy upto 10^{-12}) using MC simulation

$$\frac{area \cdot of \cdot square}{area \cdot of \cdot circle} = \frac{4r^2}{\pi r^2} = \frac{4}{\pi}$$

$$\pi = 4 * \frac{area \cdot of \cdot circle}{area \cdot of \cdot square}$$



Implement a serial program to solve the problem

Implement an OpenMP version of the above method and plot the improvement in time while using 1, 2, 4, 8 and 16 threads.

Also, please report your system specification (# of processors, etc.)

High Performance Parallel Programming (CS61064)

Week – 3
Part 2

Pralay Mitra

Example 11: Matrix Multiplication

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define NRA 62                 /* number of rows in matrix A */
#define NCA 15                 /* number of columns in matrix A */
#define NCB 7                  /* number of columns in matrix B */

int main (int argc, char *argv[])
{
int     tid, nthreads, i, j, k, chunk;
double  a[NRA][NCA],
        b[NCA][NCB],
        c[NRA][NCB];           /* result matrix*/

chunk = 10;                 /* set loop iteration chunk size */
```

Example 11: Matrix Multiplication

```
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
  {
  tid = omp_get_thread_num();
  if (tid == 0)     {
    nthreads = omp_get_num_threads();
    printf("Starting matrix multiple example with %d threads\n",nthreads);
    printf("Initializing matrices...\n");
    }

#pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      c[i][j]= 0;
```

Example 11: Matrix Multiplication

```
printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)  {
    printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<NCB; j++)
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
  }   /*** End of parallel region ***/

printf("Result Matrix:\n");
for (i=0; i<NRA; i++)  {
  for (j=0; j<NCB; j++)
    printf("%6.2f   ", c[i][j]);
    printf("\n");
  }
}
```

Example -13

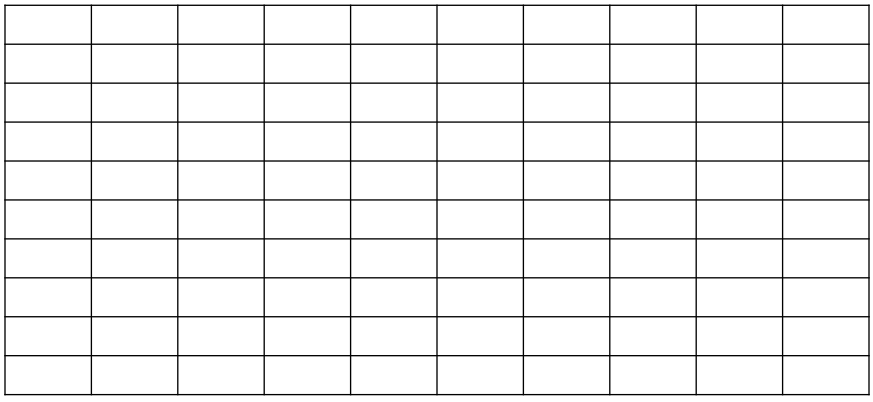
```
#pragma omp parallel for schedule(dynamic) private(tmp, i, j, k)
        for (i=0; i<Ndim; i++){
                for (j=0; j<Mdim; j++){
                        tmp = 0.0;
                        for(k=0;k<Pdim;k++){
                                tmp += *(A+(i*Ndim+k)) *  *(B+(k*Pdim+j));
                        }
                        *(C+(i*Ndim+j)) = tmp;
                }
        }
```

Loop construct: Nested Loops

WRONG

```
#pragma omp parallel
{
        #pragma omp for
                for(int ii = 0; ii < n; ii++) {
                        #pragma omp for
                                for(int jj = 0; jj < m; jj ++) {
                                        A[ii][jj] = ii*m + jj;
                                }
                }
}
```

Exploit the row-major concept



A[i][j]=*(A+(i-1)*COL+j)

Collapse clause

Available in OpenMP 3.0 and later

Clause for the PARALLEL DO directive

Cause an automatic "collapse" (merge) of the loops, and thus automatic parallelization of inner loops too

Most things taken care automatically, but user have to be careful

Loop construct: Nested Loops

```
#pragma omp parallel
{
        #pragma omp for collapse(2)
                for(int ii = 0; ii < n; ii++) {
                        for(int jj = 0; jj < m; jj ++) {
                                A[ii][jj] = ii*m + jj;
                        }
                }
}
```

The collapsed loops must be perfectly nested.

Collapse Clause

- **Purpose**
  - Specifying the COLLAPSE clause allows you to parallelize multiple loops in a nest without introducing nested parallelism.
- **Rules**
  - Only one collapse clause is allowed on a work sharing DO or PARALLEL DO directive
  - The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.
  - The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
  - If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
  - The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP directive between the loops which are collapsed.
  - The associated do-loops must be structured blocks. Their execution must not be terminated by an EXIT statement.
  - If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a CYCLE statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Example-14

```
for (i=0; i<M; i++)
        for(j=0; j<N; j++)
                for (k=0; k<P; k++)
                        c[i][k] += a[i][j] * b[j][k];
```

Only the outer loop (i) is parallel.
Loops on j,k are serial.
Possible solutions:
(i) change it to 1D array
(ii) nested parallelism

Work Out

- Can you rewrite the matrix multiplication using collapse?
  - Is it possible to collapse two nested loops?
  - Is it possible to collapse three nested loops?

Allowed Combinations

Clause	PARALLEL	DO/for	SECTIONS	SINGLE	WORKSHARE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE
IF	OK					OK	OK	OK
PRIVATE	OK	OK	OK	OK		OK	OK	OK
SHARED	OK	OK				OK	OK	OK
DEFAULT	OK					OK	OK	OK
FIRSTPRIVATE	OK	OK	OK	OK		OK	OK	OK
LASTPRIVATE		OK	OK			OK	OK	
REDUCTION	OK	OK	OK			OK	OK	OK
COPYIN	OK					OK	OK	OK
SCHEDULE		OK				OK		
ORDERED		OK				OK		
NOWAIT		OK	OK	OK	OK			

Private and Firstprivate

```
int i, j;
i = 1;
j = 2;
#pragma omp parallel private(i) firstprivate(j)
{
  i = 3;
  j = j + 2;
}
printf("%d %d\n", i, j);
```

Lastprivate

```
#pragma omp parallel
{
    #pragma omp for lastprivate(i)
        for (i=0; i<n-1; i++)
            a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

Remember

- Loop indexes are automatically PRIVATE
- Everything “local or temporary” should be PRIVATE (or FIRSTPRIVATE or LASTPRIVATE) if its value is used outside the loop, before or after respectively)
- Everything “persistent” and/or used for different values of the loop index should be SHARED a SHARED variable that is not an array accessed with the loop indexes, should be written only in a CRITICAL region (serialize, so it’s slow).
- If you are using CRITICAL, see if REDUCTION is an option (may be changing the math a little bit).

Synchronization

- CRITICAL: Mutual Exclusion
- ATOMIC: Atomic Update
- BARRIER: Barrier Synchronization

Example -17

```
#include <stdio.h>
#include <omp.h>

int main()
{
      int x=0, size=12;

      x=x+1;
      printf("%d\n",x);
      return 0;
}
```

```
$ gcc -Wall example2.c
$ ./a.out
1
$
```

Example -18

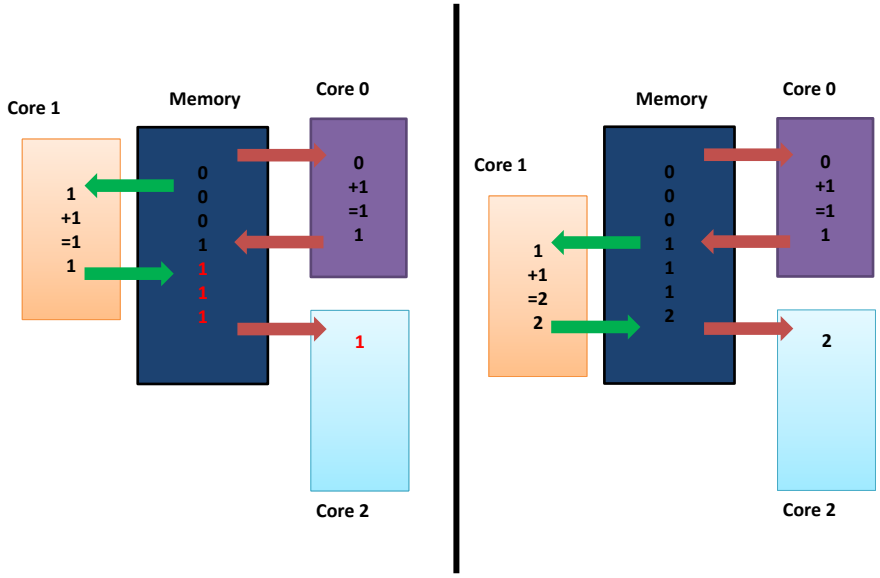
```
#include <stdio.h>
#include <omp.h>

int main()
{
      int x=0, size=12;

     omp_set_num_threads(size);
     #pragma omp parallel shared(x)
     {
           x=x+1;
      }
      printf("%d\n",x);
      return 0;
}
```

```
$ gcc -Wall -fopenmp example2_openMP.c
$ ./a.out
10
$ ./a.out
12
$ ./a.out
12
$ ./a.out
12
$ ./a.out
11
$
```

Critical construct



Critical construct

```
#pragma omp critical [name]
        structured-block

Example
        #pragma omp parallel
        {
                #pragma omp critical(long_critical_name)
                        doSomeCriticalWork_1();
                #pragma omp critical
                        doSomeCriticalWork_2();
                #pragma omp critical
                        doSomeCriticalWork_3();
        }
```

Example -19

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int x=0, size=12;
    omp_set_num_threads(size);
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        {
          x=x+1;
        }
    }
    printf("%d\n",x);
    return 0;
}
```

```
$ gcc -Wall -fopenmp example2_openMP2.c
$ ./a.out
12
$ ./a.out
12
$ ./a.out
12
$ ./a.out
12
$ ./a.out
12
$
```

Atomic construct

```
#pragma omp atomic  [read | write | update | capture]
        expression-stmt

#pragma omp atomic capture
        structured-block
```

Atomic construct

```
#pragma omp atomic \
        [read | write | update | capture]
        expression-stmt

#pragma omp atomic capture
        structured-block
```

Atomic

```
#pragma omp parallel for shared(x, y, index, n)
  for (i=0; i<n; i++)
  {
    #pragma omp atomic
      x[index[i]] += work1(i);
    y[i] += work2(i);
  }
```

Example - atomic

```
float work1(int i) {
        return 1.0 * i;
}
float work2(int i) {
        return 2.0 * i;
}
void atomic_example(float *x, float *y, int *index, int n){
        int i;
        #pragma omp parallel for shared(x, y, index, n)
                for (i=0; i<n; i++) {
                        #pragma omp atomic update
                        x[index[i]] += work1(i);
                        y[i] += work2(i);
                }
}
```

```
int main()
{
        float x[1000];
        float y[10000];
        int index[10000];
        int i;
        for (i = 0; i < 10000; i++) {
                index[i] = i % 1000;
                y[i]=0.0;
        }
        for (i = 0; i < 1000; i++)
                x[i] = 0.0;
        atomic_example(x, y, index, 10000);
        return 0;
}
```

Barrier construct

#pragma omp barrier //Threads wait until all threads reach this point

Example (waiting for the master to come)

```
int counter = 0;
#pragma omp parallel
{
        #pragma omp master
                counter = 1;
        #pragma omp barrier
                printf("%d\n", counter);
}
```

Be careful not to cause deadlock:
No barrier inside of *critical, master, sections, single*!

Example -20

```
#include <stdio.h>
#include <omp.h>

int main()
{
      int i,x=0, a[10],b[10];

      for(i=0;i<10;i++) {
            a[i]=i;
            b[i]=10-i;
      }

      for(i=0;i<10;i++) {
            x = x + a[i]*b[i];
      }

      printf("%d\n",x);
      return 0;
}
```

```
$ ./a.out
165
$
```

Example -21

```
#include <stdio.h>
#include <omp.h>
int main()
{
      int i,x=0, a[10],b[10];
      for(i=0;i<10;i++) {
            a[i]=i;
            b[i]=10-i;
      }
      #pragma omp parallel
      {
            for(i=0;i<10;i++) {
                  x = x + a[i]*b[i];
            }
      }
      printf("%d\n",x);
      return 0;
}
```

```
$ gcc -Wall -fopenmp example3_openMP.c
$ ./a.out
620
$ ./a.out
834
$ ./a.out
350
$ ./a.out
213
$ ./a.out
138
$ ./a.out
186
$ ./a.out
221
$ ./a.out
106
$ ./a.out
403
$
```

Correct it.

Example -22

```
#include <stdio.h>
#include <omp.h>
int main()
{
      int i,x=0, a[10],b[10];
      for(i=0;i<10;i++) {
            a[i]=i;   b[i]=10-i;
      }
      #pragma omp parallel
      {
            #pragma omp for reduction (+:x)
                  for(i=0;i<10;i++) {
                        x = x + a[i]*b[i];
                  }
      }
      printf("%d\n",x);
      return 0;
}
```

```
$ ./a.out
165
$
```

Reduction Clause

- sum is the reduction variable
- cannot be declared shared
  - threads would overwrite the value of sum
- cannot be declared private
  - private variables don't persist outside of parallel region
- specified reduction operation performed on individual values from each thread

Reduction Operand

Operator	Initial value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Home Work

Consider a list of integers and output their mean and standard deviation.

Test the performance on using reduction clause over your previous implementation.