上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

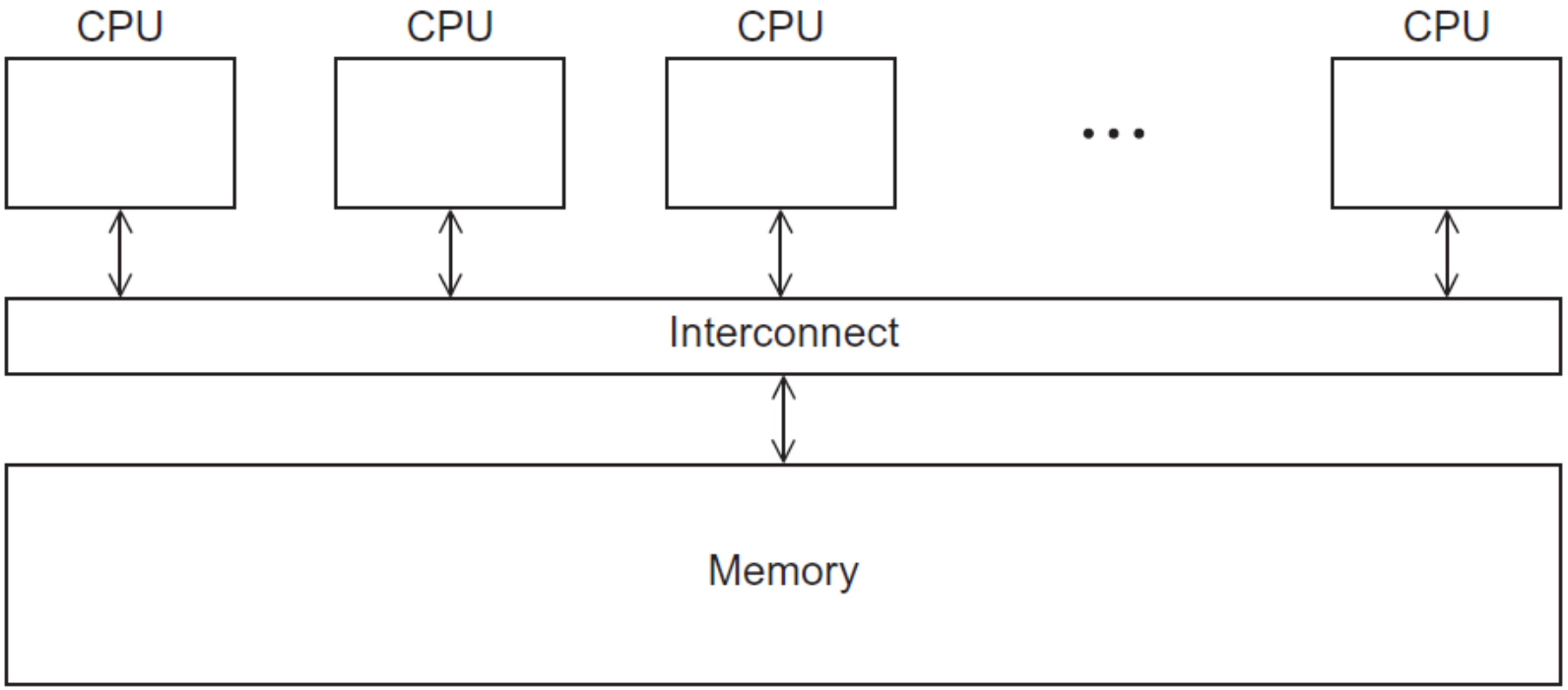
CS4302-01
Parallel and Distributed Computing

Lecture 3 OpenMP

Zhuoran Song

2023/9/22

A Shared Memory System



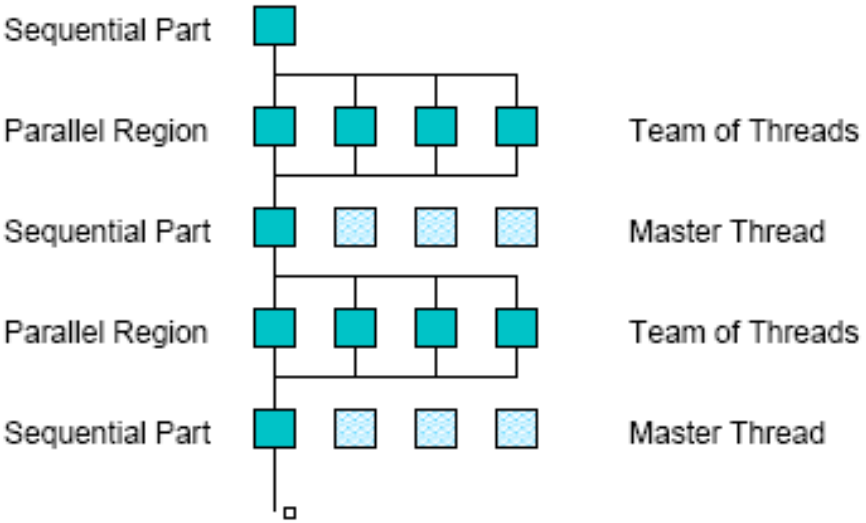
Shared Memory Programming

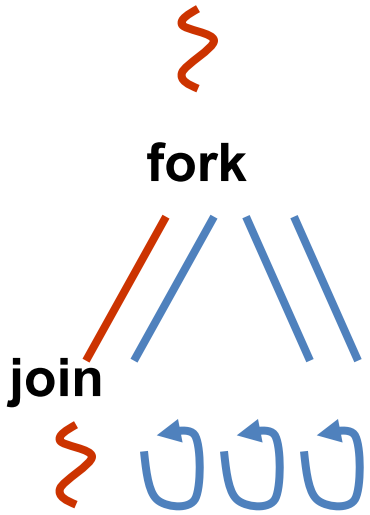
- **Shared Memory Programming**
  - *Start a single process and fork threads.*
  - *Threads carry out work.*
  - *Threads communicate through shared memory.*
  - *Threads coordinate through synchronization (also through shared memory).*

- **Distributed Memory Programming**
  - *Start multiple processes on multiple systems.*
  - *Processes carry out work.*
  - *Processes communicate through message-passing.*
  - *Processes coordinate either through message-passing or synchronization (generates messages).*

Execution Model

- **Fork-join model of parallel execution**
- **Begin execution as a single process (master thread)**
- **Start of a parallel construct:**
  - *Master thread creates team of threads (worker threads)*
- **Completion of a parallel construct:**
  - *Threads in the team synchronize -- implicit barrier*
- **Only master thread continues execution**
- **Implementation optimization:**
  - *Worker threads spin waiting on next fork*





OpenMP

- An API for shared-memory parallel programming, MP = multiprocessing
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU's, all of which have access to main memory.
- Higher-level support for scientific programming on shared memory architectures.
- Programmer identifies parallelism and data properties, and guides scheduling at a high level.
- System decomposes parallelism and manages schedule.

See http://www.openmp.org

OpenMP

- **Common model for shared-memory parallel programming**
  - *Portable across shared-memory architectures*
- **Scalable (on shared-memory platforms)**
- **Incremental parallelization**
  - *Parallelize individual computations in a program while leaving the rest of the program sequential*
- **Compiler based**
  - *Compiler generates thread program and synchronization*
- **Extensions to existing programming languages (Fortran, C and C++)**
  - *mainly by directives*
  - *a few library routines*

Programmer's View

- **OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax**
  - *Exact behavior depends on OpenMP implementation!*
  - *Requires compiler support (C/C++ or Fortran)*
- **OpenMP will:**
  - *Allow a programmer to separate a program into serial regions and parallel regions, rather than concurrently-executing threads.*
  - *Hide stack management*
  - *Provide synchronization constructs*
- **OpenMP will not:**
  - *Parallelize automatically*
  - *Guarantee speedup*
  - *Provide freedom from data races*

Pragmas

- **Pragmas are special preprocessor instructions.**
- **Typically added to a system to allow behaviors that aren’t part of the basic C specification.**
- **Compilers that don’t support the pragmas ignore them.**
- **The interpretation of OpenMP pragmas**
  - *They modify the statement immediately following the pragma*
  - *This could be a compound statement such as a loop*

#pragma omp …

“Hello World”

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

指定多少个线程

“Hello World”

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./ omp_hello 4
```

compiling

running with 4 threads

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

possible outcomes

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4

Generate Parallel Threads

- **# pragma omp parallel num_threads (thread_count**
  - *Most basic parallel directive.*
  - *The number of threads that run the following structured block of code is determined by the run-time system.*
- **# pragma omp parallel num_threads (thread_count**
  - *Clause: text that modifies a directive.*
  - *The num_threads clause can be added to a parallel directive.*
  - *It allows the programmer to specify the number of threads that should execute the following block.*

Query Fuctions

int omp_get_num_threads(void); 目前有多少线程

Returns the number of threads currently in the team executing the parallel region from which it is called

int omp_get_thread_num(void); 目前的线程**id**

Returns the thread number, within the team, that lies between `0` and `omp_get_num_threads()-1`, inclusive. The master thread of the team is thread `0`

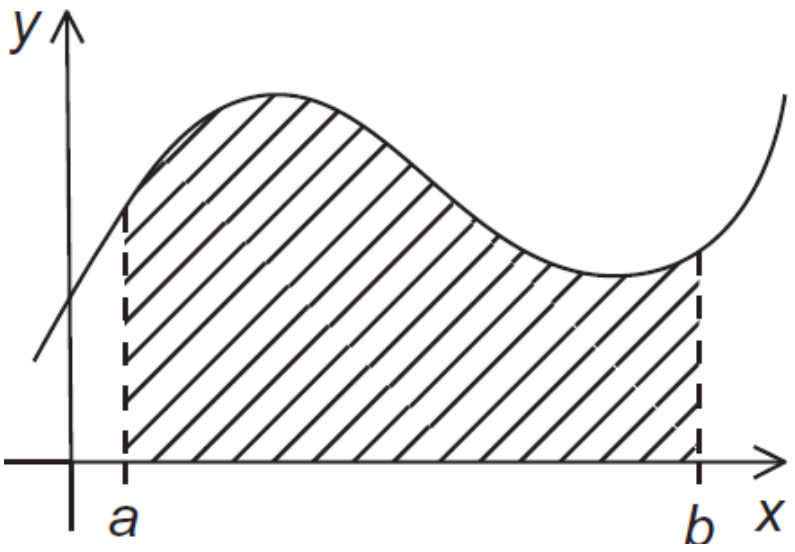
- There may be system-defined limitations on the number of threads that a program can start.

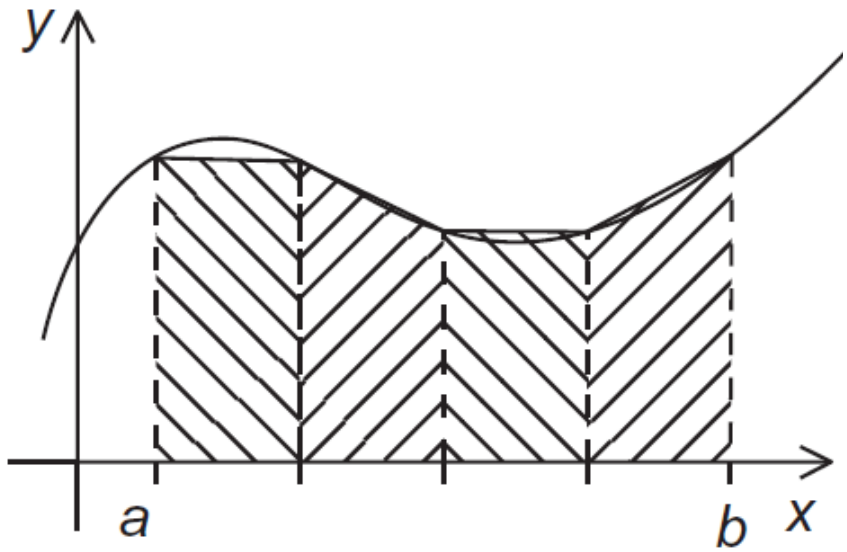
  setenv OMP_NUM_THREADS 16 [csh, tcsh] 线程上限

- The OpenMP standard doesn't guarantee that this will actually start thread_count threads.

- Use the above functions to get the actual thread number and ID.

The Trapezoidal Rule






```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

OpenMP Implementation

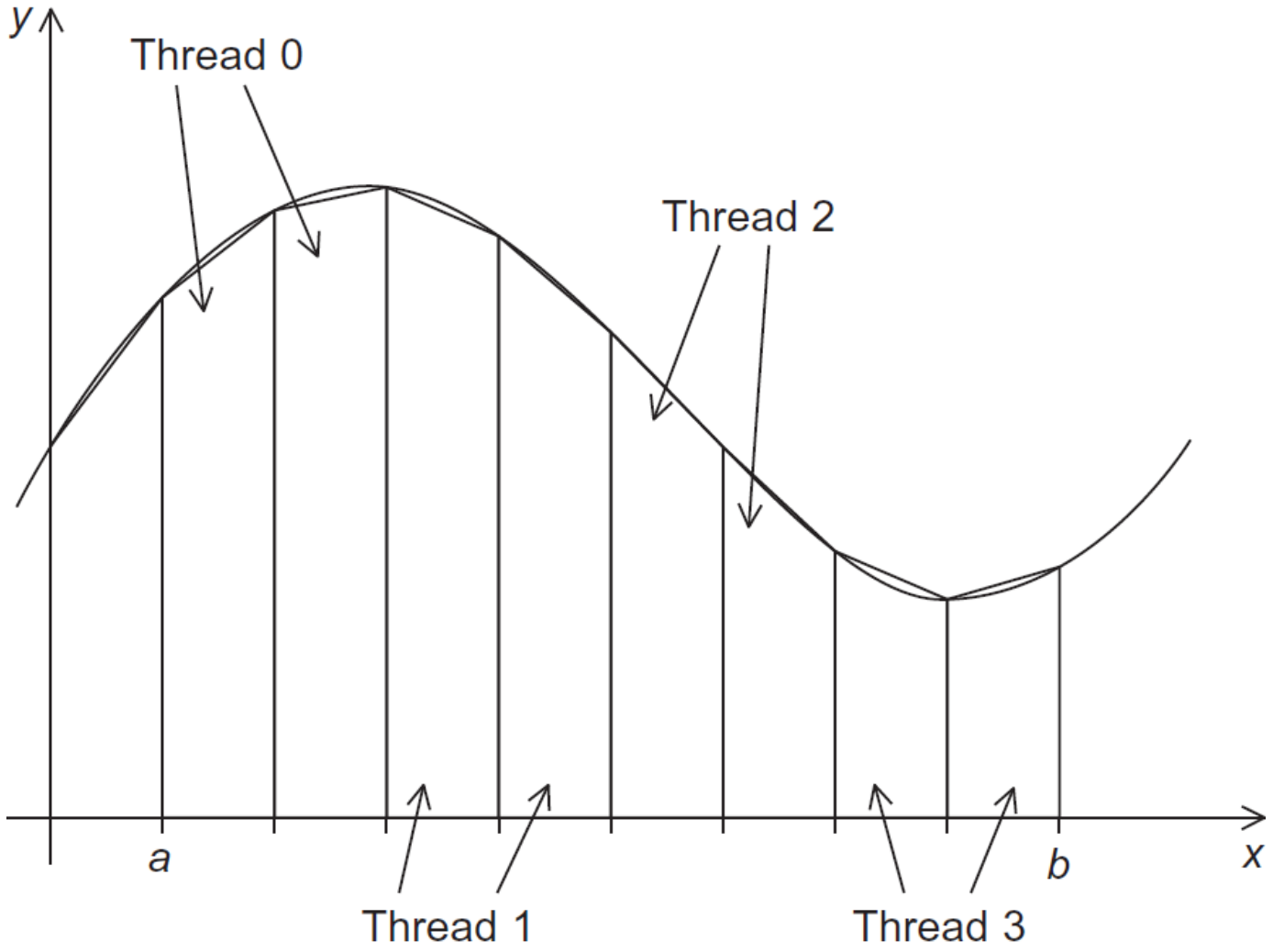
- **We identified two types of tasks:**

  a) computation of the areas of individual trapezoids, and

  b) adding the areas of trapezoids.

- **There is no communication among the tasks in the first collection, but each task in the first collection communicates with task b.**
- **So we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).**

OpenMP Implementation



OpenMP Implementation

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```

OpenMP Implementation

```
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

Mutual Exclusion

Time	Thread 0	Thread 1
0	`global_result = 0` to register	finish `my_result`
1	`my_result = 1` to register	`global_result = 0` to register
2	add `my_result` to `global_result`	`my_result = 2` to register
3	store `global_result = 1`	add `my_result` to `global_result`
4		store `global_result = 2`

Unpredictable results when two (or more) threads attempt to simultaneously execute:

global_result += my_result ;

Reduction

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

+	bitwise &	logical &
-	bitwise |	logical |
*	bitwise ^	max/min

Reduction

A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```

+, *, -, &, |, ^, &&, ||

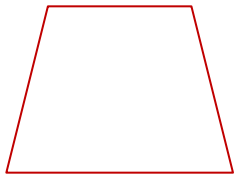
```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count) \
       reduction(+: global_result)
    global_result += Local_trap(double a, double b, int n);
```

“Parallel For”

- Forks a team of threads to execute the following structured block.
- However, the structured block following the parallel for directive must be a for loop.
- Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.

“Parallel For”

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```



```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```

Caveats

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

note 2 threads

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

Caveats

- OpenMP compilers don’ t check for dependences among iterations in a loop that’ s being parallelized with a parallel for directive.

- A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

Estimating PI

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

Estimating PI

loop dependency

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum)
        for (k = 0; k < n; k++) {
            sum += factor/(2*k+1);
            factor = -factor;
        }
        pi_approx = 4.0*sum;
```

Scope of Variables

- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.
- The default scope for variables declared before a parallel block is shared.

Estimating PI

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
            if (k % 2 == 0)
               factor = 1.0;
            else
               factor = -1.0;
            sum += factor/(2*k+1);
        }
```

Insures factor has private scope.

Scope of Variables

- Lets the programmer specify the scope of each variable in a block. `default(none)`

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

```
    double sum = 0.0;
#   pragma omp parallel for num_threads(thread_count) \
        default(none) reduction(+:sum) private(k, factor) \
        shared(n)
    for (k = 0; k < n; k++) {
        if (k % 2 == 0)
            factor = 1.0;
        else
            factor = -1.0;
        sum += factor/(2*k+1);
    }
```

Loop Carried Dependence

```
for (i=0; i<100; i++) {
    A[i+1]=A[i]+C[i];
    B[i+1]=B[i]+A[i+1];
}
```

- **Two loop carried dependence.**
- **One intra-loop dependence.**

Loop Carried Dependence

```
for (i=0; i<100; i++) {
    A[i]=A[i]+B[i];
    B[i+1]=C[i]+D[i];
}
```

Eliminating loop dependence:

```
A[0]=A[0]+B[0];
for (i=0; i<99; i++) {
    B[i+1]=C[i]+D[i];
    A[i+1]=A[i+1]+B[i+1];
}
B[100]=C[99]+D[99];
```

Loop Scheduling

- Default schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- Cyclic schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

Loop Scheduling

schedule (type , chunksize)

- Type can be:
  - *static*: *the iterations can be assigned to the threads before the loop is executed.*
  - *dynamic* or *guided*: *the iterations are assigned to the threads while the loop is executing.*
  - *auto*: *the compiler and/or the run-time system determine the schedule.*
  - *runtime*: *the schedule is determined at run-time.*
- The chunksize is a positive integer.

Static Scheduling

`schedule(static,1)`	Thread 0 : 0,3,6,9 Thread 1 : 1,4,7,10 Thread 2 : 2,5,8,11
`schedule(static,2)`	Thread 0 : 0,1,6,7 Thread 1 : 2,3,8,9 Thread 2 : 4,5,10,11
`schedule(static,4)`	Thread 0 : 0,1,2,3 Thread 1 : 4,5,6,7 Thread 2 : 8,9,10,11

Dynamic Scheduling

- The iterations are also broken up into chunks of chunksize consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- This continues until all the iterations are completed.
- The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.

Guided Scheduling

- Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
- However, in a guided schedule, as chunks are completed the size of the new chunks decreases exponentially.
- If no chunksize is specified, the size of the chunks decreases down to 1.
- If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.

Guided Scheduling

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

Runtime Scheduling

- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.

- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.

- For example

  setenv OMP_SCHEDULE GUIDED,4 [csh, tcsh]

  export OMP_SCHEDULE=GUIDED,4 [sh, ksh, bash]

Impact of Scheduling

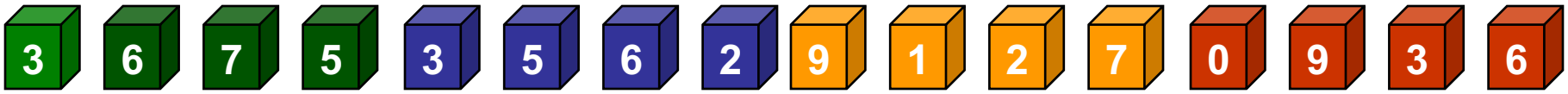
- Load balance
  - *Same work in each iteration?*
  - *Processors working at same speed?*
- Scheduling overhead
  - *Static decisions are cheap because they require no run-time coordination*
  - *Dynamic decisions have overhead that is impacted by complexity and frequency of decisions*
- Data locality
  - *Particularly within cache lines for small chunk sizes*
  - *Also impacts data reuse on same processor*

Data Locality

- Consider a 1-Dimensional array to solve the global sum problem, 16 elements, 4 threads

CYCLIC (chunk = 1):

BLOCK (chunk = 4):



Data Locality

Consider how data is accessed

- **Temporal locality**
  - *Same or nearby data used multiple times*
  - *Intrinsic in computation*
- **Spacial locality**
  - *Data nearby to be used and is present in "fast memory"*
  - *Same data transfer (same cache line, same DRAM transaction)*
- **What can we do to get locality**
  - *Appropriate data placement and layout*
  - *Code reordering transformations*