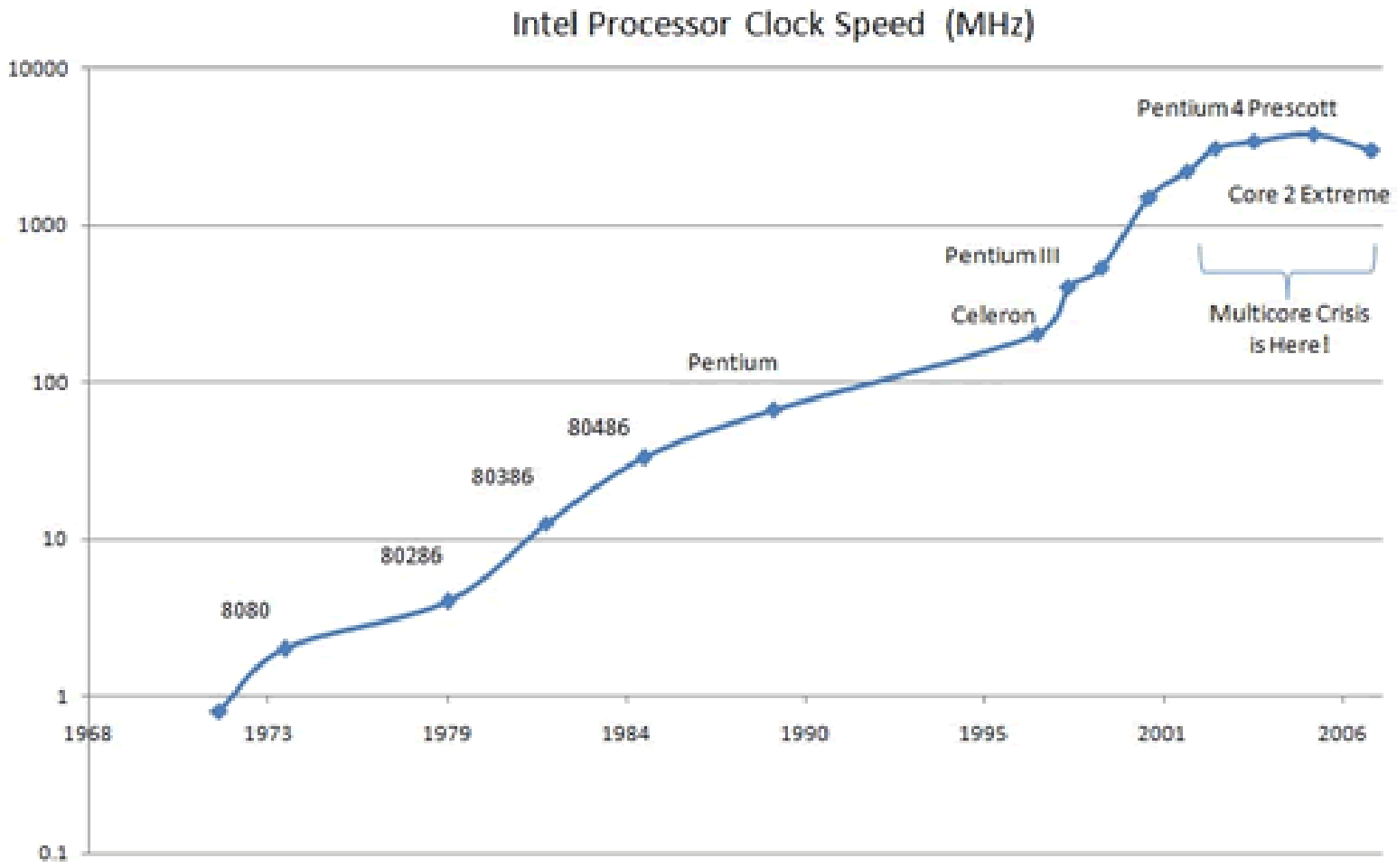
Slide Presentation
to accompany
Parallel Programming
Bruce P. Lester

Lesson 1

Overview of
Multi-Core Processors

Evolution of Processor Speed



Software Multi-Threading

- Multi-Threading is done explicitly in the software
- A ***Thread*** consists of a sequence of program instructions and some data values
- Threads are created automatically by the operating system (example: one thread for each application that is open)
- Multiple threads can also be created within one application program

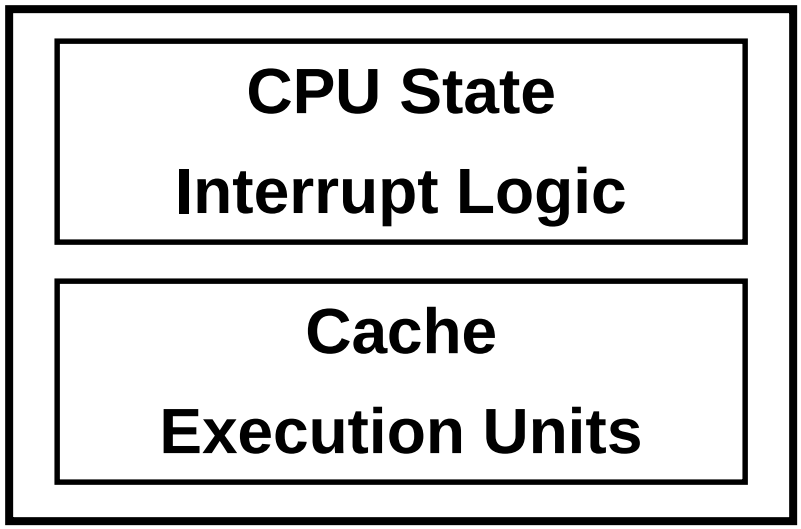
Purpose of Multi-Threading

- Threads created by operating system to improve utilization of the hardware
- Example: One thread is downloading a file from the Internet while another thread is running Microsoft Word
- Threads hide I/O latencies: while waiting for a disk I/O thread to finish, another thread can execute on the processor

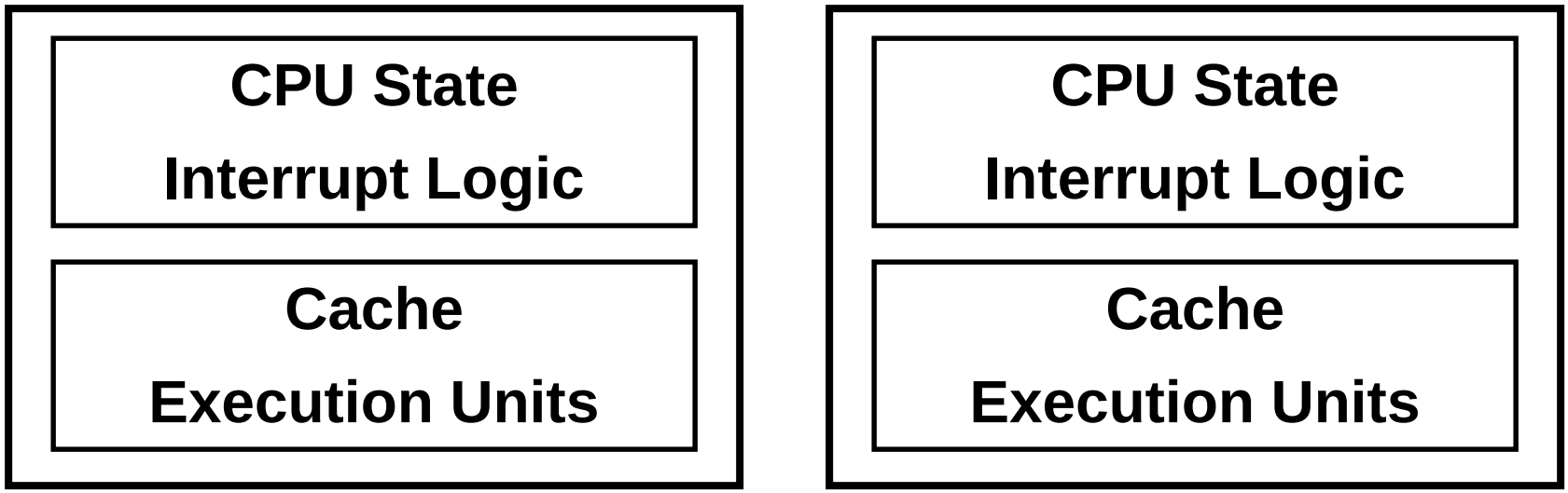
Multi-Threading on Multi-Core Processors

- Multi-Core means two or more execution cores capable of executing independent instruction streams in parallel
- Multiple threads can be executed in parallel by different cores
- Can produce significant performance improvements

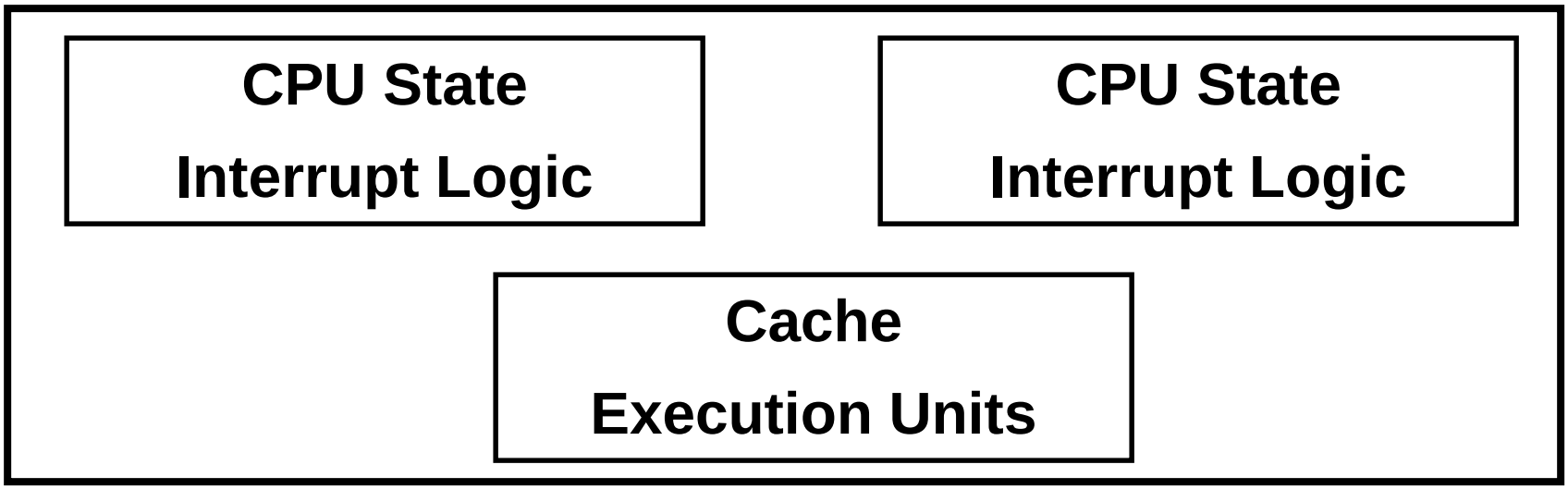
Computer Processor



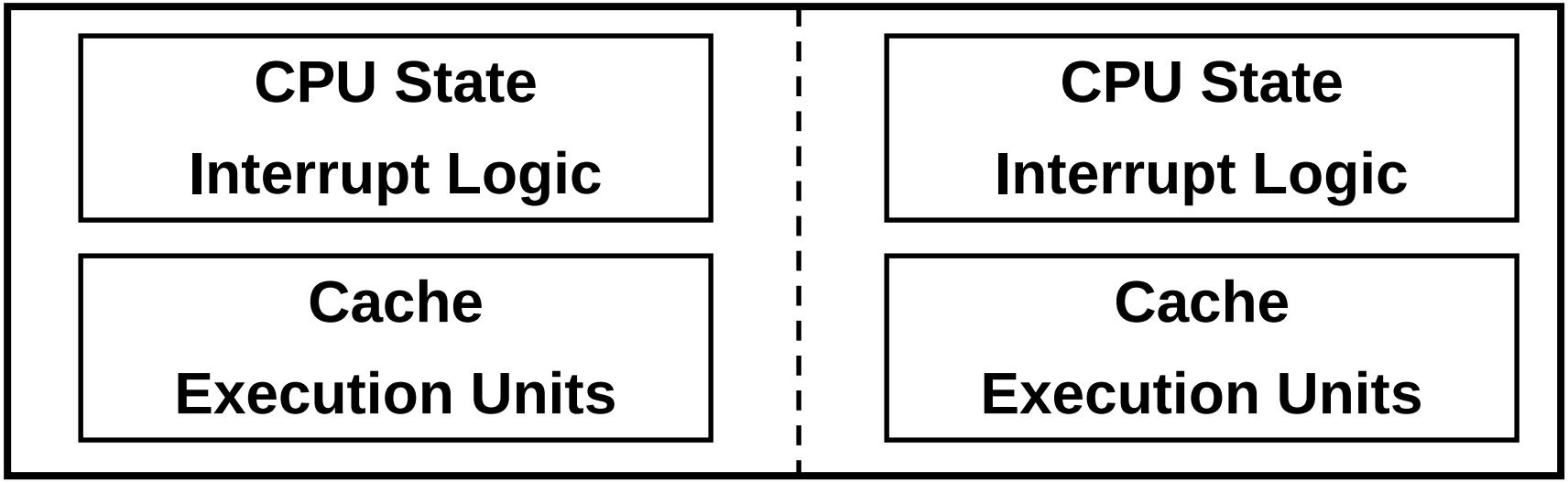
Multiprocessor Computer



Processor with Hyper-Threading

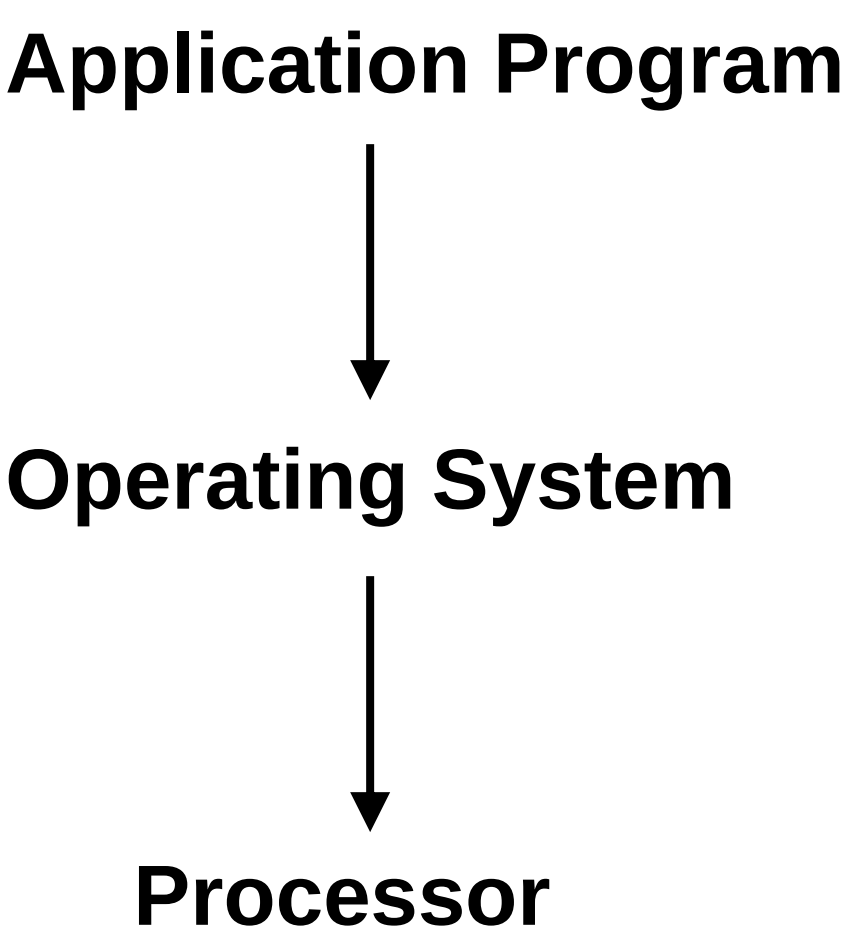


Multi-Core Processor



Hyper-Threading

- Processor provides support for multiple threads of execution
- Threads only **appear** to run in parallel
- Execution of instructions from different threads are interleaved
- Multi-core processors have **real** parallelism: threads are executed in parallel by different cores

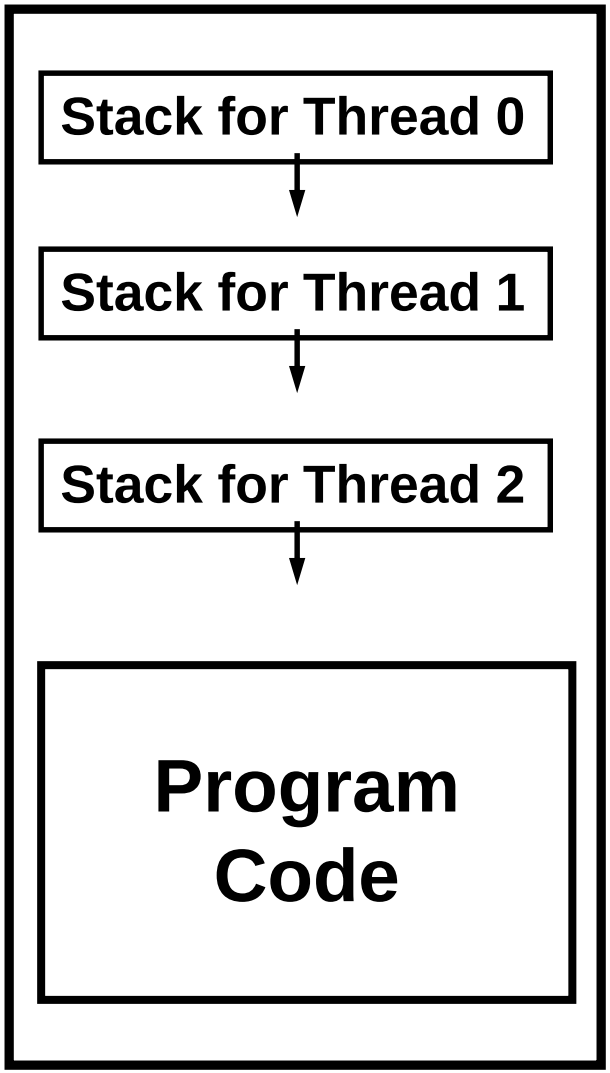


Creates multiple threads using API like OpenMP

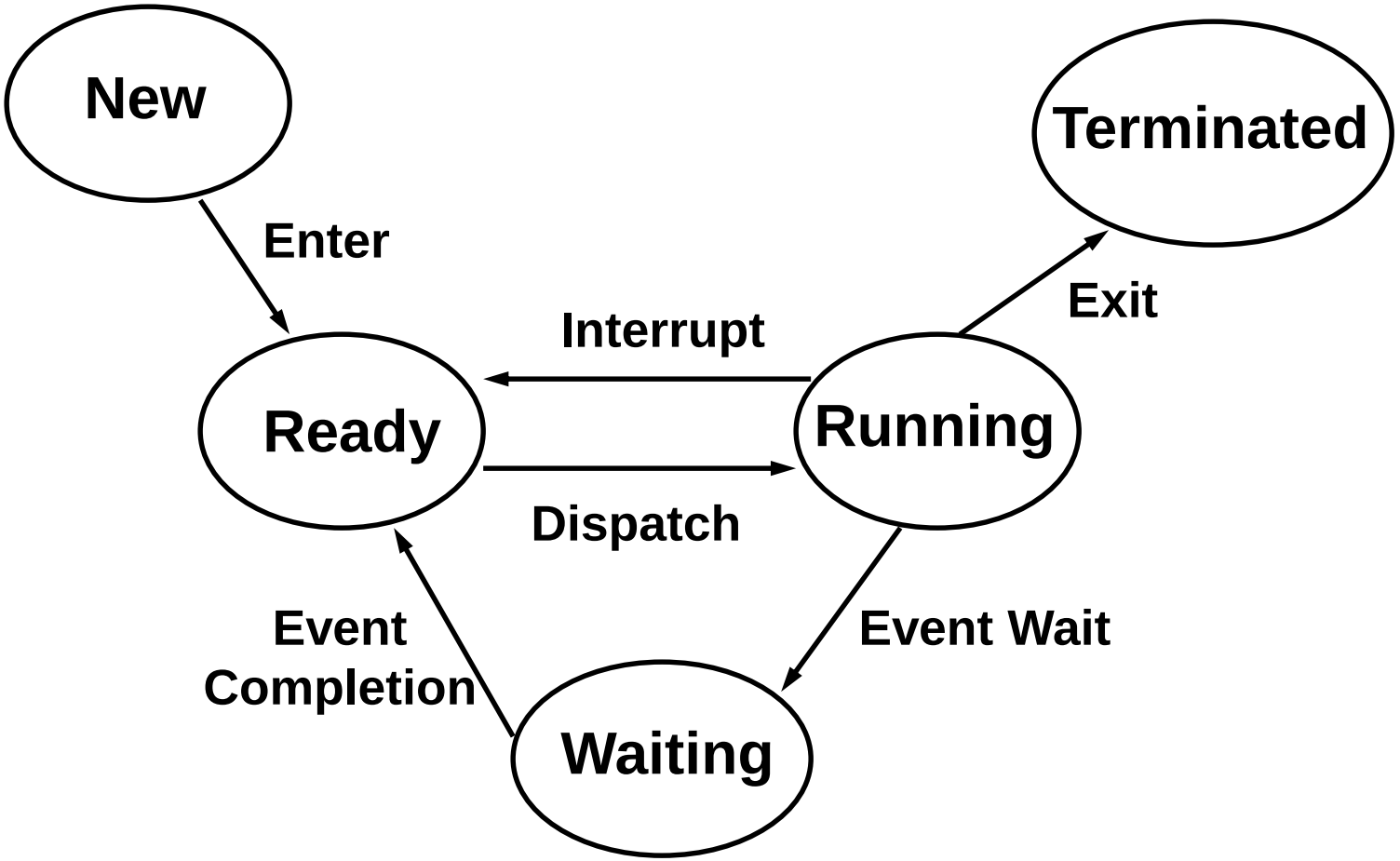
Schedules thread execution and manages memory of the threads

Executes the instructions of the threads

Memory for Multiple Threads



Thread State Diagram



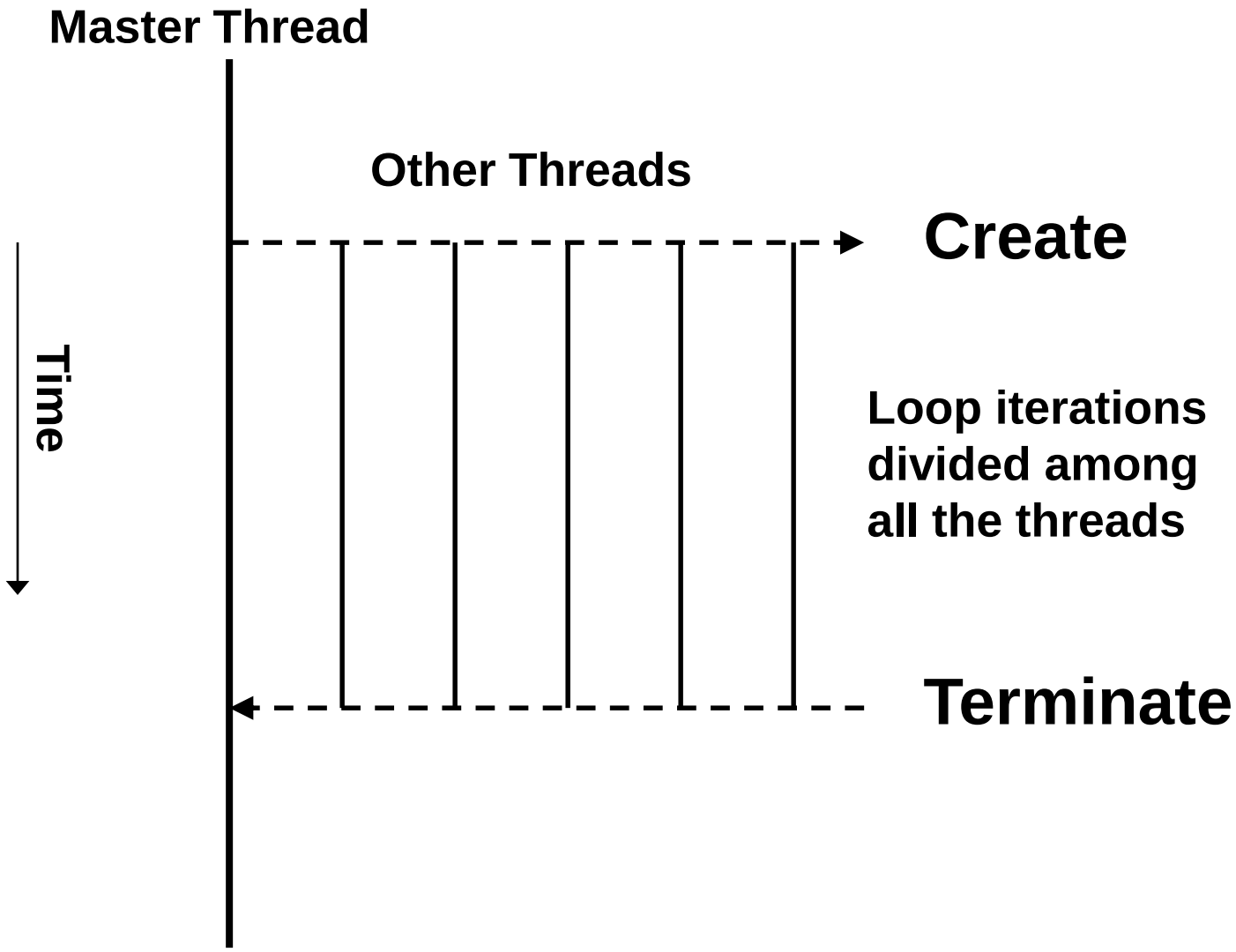
C Language

```
/* multiply two arrays */
for (i = 1; i <= 10000; i++)
  c[i] = a[i] * b[i];
```

C Language using OpenMP

```
/* create four threads */
omp_set_num_threads(4);
#pragma omp parallel for
for ( i = 1; i <= 10000; i++ )
  c[i] = a[i] * b[i];
```

OpenMP Parallel For



C Program

```
#include <stdio.h>
#define size 10000
int i, a[size], b[size], c[size];

int main( )  {
  /* initialization */
  for (i = 0; i < size; i++)
    a[i] = b[i] = i;
  /* multiply arrays */
  for (i = 0; i < size; i++)
      c[i] = a[i] * b[i];
}
```

C with OpenMP

```
#include <stdio.h>
#include <omp.h>
#define size 10000
int i, a[size], b[size], c[size];

int main( )  {
  for (i = 0; i < size; i++)
    a[i] = b[i] = i;
  omp_set_num_threads(2);
  #pragma omp parallel for
     for (i = 0; i < size; i++)
       c[i] = a[i] * b[i];
}
```

Function to Set Number of Threads:

```
void omp_set_num_threads( int t )
```

Rank Sort

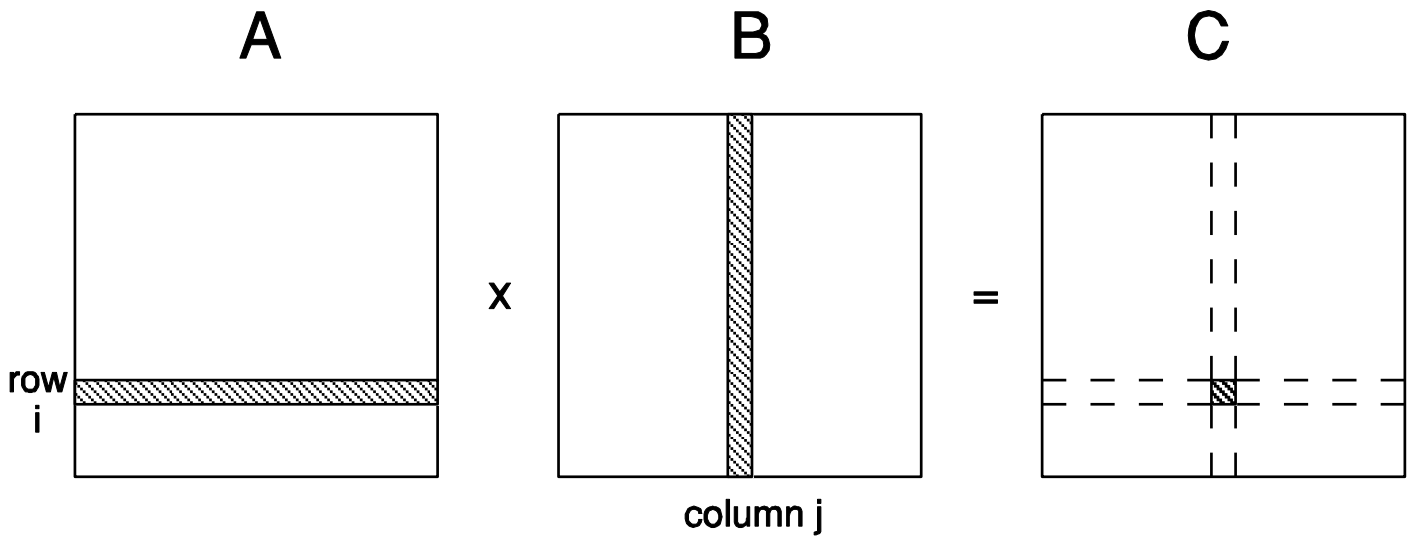
(Unsorted) LIST	RANK
15	4
10	3
39	7
8	2
22	6
4	0
19	5
6	1

```
#define n 10000
int values[n], final[n];   int i;

 void PutinPlace( int src ) {
   int testval, j, rank;

   testval = values[src];
   j = src;   /*j moves through the whole array*/
   rank = 0;
   do {
     j = j % n;
     if (testval >= values[j]) rank = rank + 1;
   } while (j != src);
   final[rank] = testval;  /*put into position*/
 }

main() {
for (i = 0; i < n; i++)
  cin >> values[i];  /*initialize values*/
for (i = 0; i < n; i++)
   PutinPlace(i);  /*put values[i] in place*/
}
```

Sequential Rank Sort

```
#include <omp.h>
#define n 10000
int values[n], final[n];
int i;

void PutinPlace( int src ) {
  int testval, j, rank;

   /* same as Sequential Version */

}

main() {
for (i = 0; i < n; i++)
  cin >> values[i];  /*initialize values*/
omp_set_num_threads(2);
#pragma omp parallel for
  for (i = 0; i < n; i++)
    PutinPlace(i);  /*put values[i] in place*/
}
```

Parallel Rank Sort using OpenMP



C[i,j] is computed as vector product of row i of A with column j of B.

Matrix multiplication

Matrix Multiplication C = A * B:

```
for i = 0 to n-1 do
  for j = 0 to n-1 do
     compute C[i][j] as the vector product
       of row i of A with column j of B;
```

```
#define n 10000
float A[n], B[n], C[n];
int i, j;

main()  {
   ...

for (i = 0; i < n; ++) {
  for (j = 0; j < n; j++) {
    sum = 0;
    for  (k = 0; k < n; k++)
      sum = sum + A[i][k] * B[k][j];
    C[i][j] = sum;
  }
}
}
```

Sequential Matrix Multiplication

```
#include <omp.h>
#define  n  10000
 float  sum, A[n][n], B[n][n], C[n][n];
 int  i, j, k;

 main()  {
   ...
 omp_set_num_threads(2);
 #pragma omp parallel for private(sum,j,k)
   for (i = 0; i < n; i++) {
     for (j = 0; j < n; j++) {
       sum = 0;
       for  (k = 0; k < n; k++)
         sum = sum + A[i][k] * B[k][j];
       C[i][j] = sum;
     }
   }
 }
```

Matrix Multiplication (OpenMP Version)

Programming Exercise Logarithm of an Array

- Write a multi-core program using C and OpenMP to compute the *log* of every element of an array A[100000000].
- Run and test the program using Microsoft Visual Studio.
- Determine the execution time for total number of threads from 1 to 10.
- Explain the reason why additional threads sometimes do not reduce the execution time.

Program Execution Time

`clock_t clock(void)`

returns number of clock ticks since start of program (one clock tick is about a millisecond)

```
#include <time.h>
clock_t  t;

int main( ) {
  t = clock( );  /* time now */

  ...

  printf("Elapsed Time: %d\n", clock( )-t);
}
```

Performance Measure

Ordinary (sequential) program execution time: S
Multi-Threaded program execution time: T
Number of cores in the processor: P

“Speedup” is defined as: S/T

“Processor Utilization” is defined as: Speedup/P

The maximum speedup is P. Therefore, the maximum utilization is 100%.

Setting Command Window Properties

- Right-click on Command Window title bar and select “Properties”.
- Use “Colors” tab and set “Screen Text” all 0 and “Screen Background” all 255.
- Use “Layout” tab and set Screen Buffer Size height to 1000.
- Click OK and select “Save Properties for future”.

Printing Program Output

- After executing program, type Alt + “Print Screen”.
- Start Menu–> Programs–> Accessories–> Paint
- Select Edit Menu and Paste (or Control-v)
- Select File Menu and Page Setup. Set Scaling to “Fit to 1 page”.
- Select File Menu and Print.

Programming Exercise Merging Sorted Lists

- Write a multi-core program using C and OpenMP to merge two sorted lists X[n] and Y[n] into a sorted list Z[2*n]. (Use n = 40000000)
- Run and test the program using Microsoft Visual Studio.
- Determine the execution time for total number of threads from 1 to 10.
- Explain the reason why additional threads sometimes do not reduce the execution time.

Merging Sorted Lists

- Consider the usual sequential algorithm for merging two sorted lists: use a pointer for each sorted list X and Y that steps through the list. Compare the elements from the two lists that are pointed to, and copy the smaller element to the final list Z.
- Explain why this algorithm is difficult to parallelize.
- What happens if the *for* loop for this algorithm is parallelized?

Merging Sorted Lists
The Algorithm

- Move through the elements of list X one at a time.
- For each X[i], scan through list Y to find the location j where it would be placed if it were inserted in sorted order into list Y.
- Then copy X[i] to Z[i+j].
- Do the same thing for list Y, finding the location j in list X where Y[i] would be placed.
- Then copy Y[i] to Z[i+j].
- The above sequential algorithm is easily parallelized using OpenMP parallel for.

Lesson 2

Locks and Reduction

Slide Presentation
to accompany
Parallel Programming
Bruce P. Lester

for (index = start; index [< | <= | >= | >] end; [index++ | ++index | index- - | - -index | index += inc | index - = inc | index = index + inc | index = inc + index | index = index - inc])

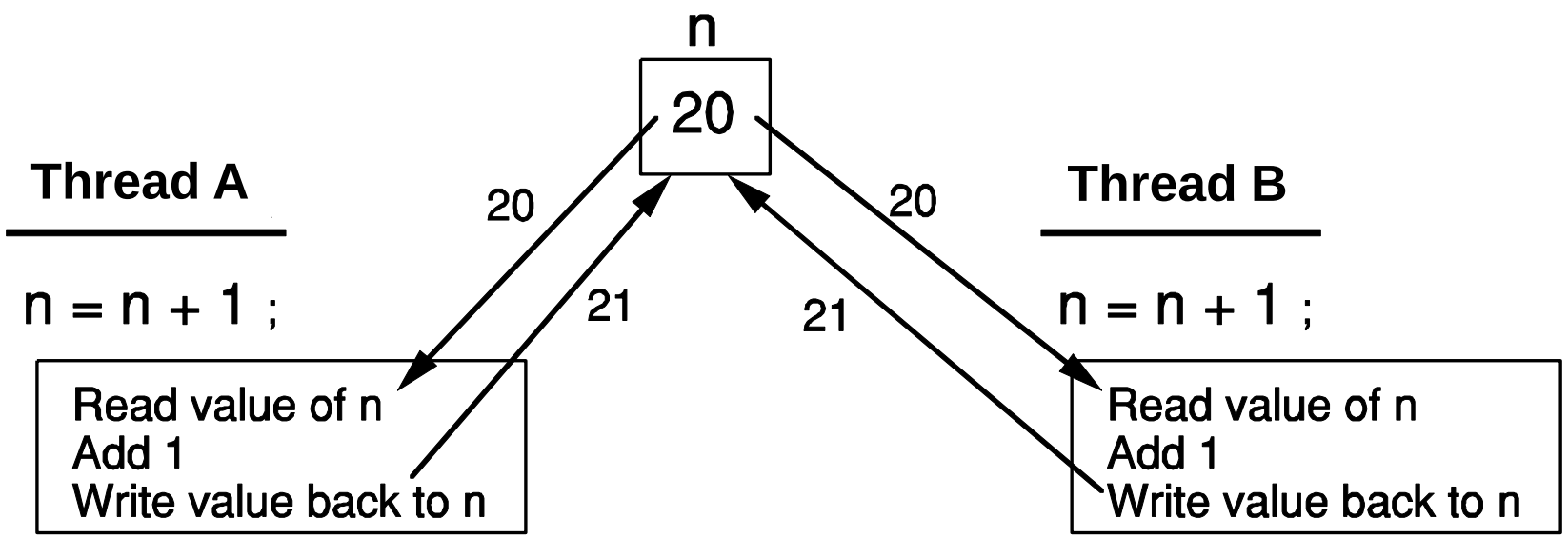
Allowed Form of Parallel For Loops

```
int main( )  {
  int i, a[size], b[size], c[size];

  #pragma omp parallel for
     for (i=0; i < size; i++)
        c[i] = a[i] * b[i];
}
```

Arrays a, b, c are *shared* by all threads
Variable i is *private*:
each thread has its own copy

Shared vs. Private Variables

- All variables declared in the C program are **shared**, unless explicitly listed in a “private” clause in the parallel pragma.
- When a **shared** variable is modified by any thread, all other threads see this modification during the next read because there is only one copy of the variable.
- When a **private** variable is modified by a thread B, only thread B will see this modification. All other threads have a different copy of the private variable, each with its own distinct value.



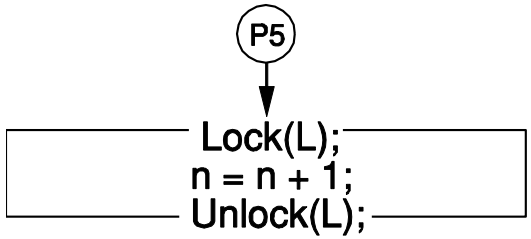
Error with shared data

```
/* PROGRAM Search (with data sharing error) */
int  A[200];
int  i, val, n;

main( )
{
   ...

  n = 0;
  cin >> val;
#pragma omp parallel for
  for (i = 0; i < 200; i++)
    if (A[i] == val) n = n + 1;
  cout << "Total occurrences: " << n << endl;
   ...
}
```

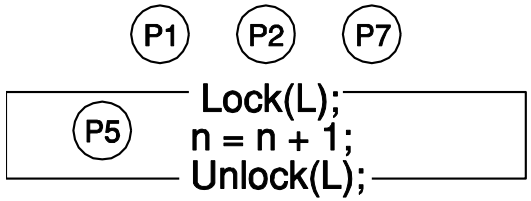
PROGRAM Search (with locking)

```
int  A[201];
int  i, val, n;
spinlock L;

main( )  {
   ...
   n = 0;
   cin >> val;
#pragma omp parallel for
   for (i = 0; i < 200; i++)
     if (A[i] == val)  {
       Lock(L);   /*Enter and lock the spinlock*/
       n = n + 1;
       Unlock(L); /*Unlock to allow another to enter*/
     }
   cout << "Total occurrences: " << n << endl;
   ...
}
```

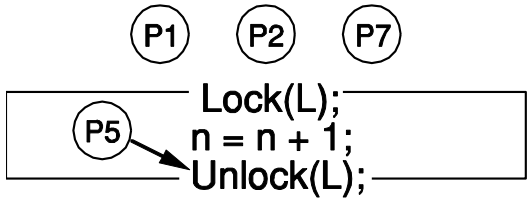
Effect of Spinlocks



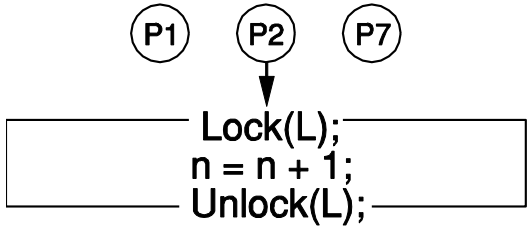
L initially unlocked
P5 enters and locks L



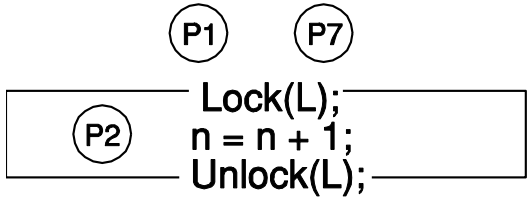
L is now locked
P1, P2, P7 kept out



P5 unlocks L as it leaves



L is now unlocked
P2 enters and locks L



L is now locked
P1, P7 kept out

Locks in OpenMP

```
void omp_set_lock(omp_lock_t *lock);
```

If the lock is currently in "unlocked" state, then the lock is put into the "locked" state and calling thread continues execution.
If lock is currently in "locked" state, calling thread is blocked until lock becomes "unlocked".

```
void omp_unset_lock(omp_lock_t *lock);
```

Changes the lock from "locked" state to the "unlocked" state, so new threads can enter.

```
void omp_init_lock(omp_lock_t *lock);
```

Must be done once for each lock.

```
#define  n  20        /*dimension of the image*/
#define  max  10  /*maximum pixel intensity*/
int  i, image[n][n], hist[max];

main( )  {
  ...  /*Initialize the image array*/
  for (i = 0; i < max; i++) hist[i] = 0;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)  {
      intensity = image[i][j];
      hist[intensity] = hist[intensity] + 1;
    }
  }
}
```

Sequential program for histogram

```
#include <omp.h>
#define n 20    /*dimension of the image*/
#define max 10  /*maximum pixel intensity*/
int i,j,intensity, image[n][n], hist[max];
omp_lock_t L[max];

main( )  {
  ...  /*Initialize the image array*/
  for (i = 0; i < max; i++) hist[i] = 0;
  #pragma omp parallel for private(j,intensity)
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++)  {
        intensity = image[i][j];
        omp_set_lock(&L[intensity]);
          hist[intensity] = hist[intensity] + 1;
        omp_unset_lock(&L[intensity]);
      }
    }
}
```

Parallel program for histogram (OpenMP Version)

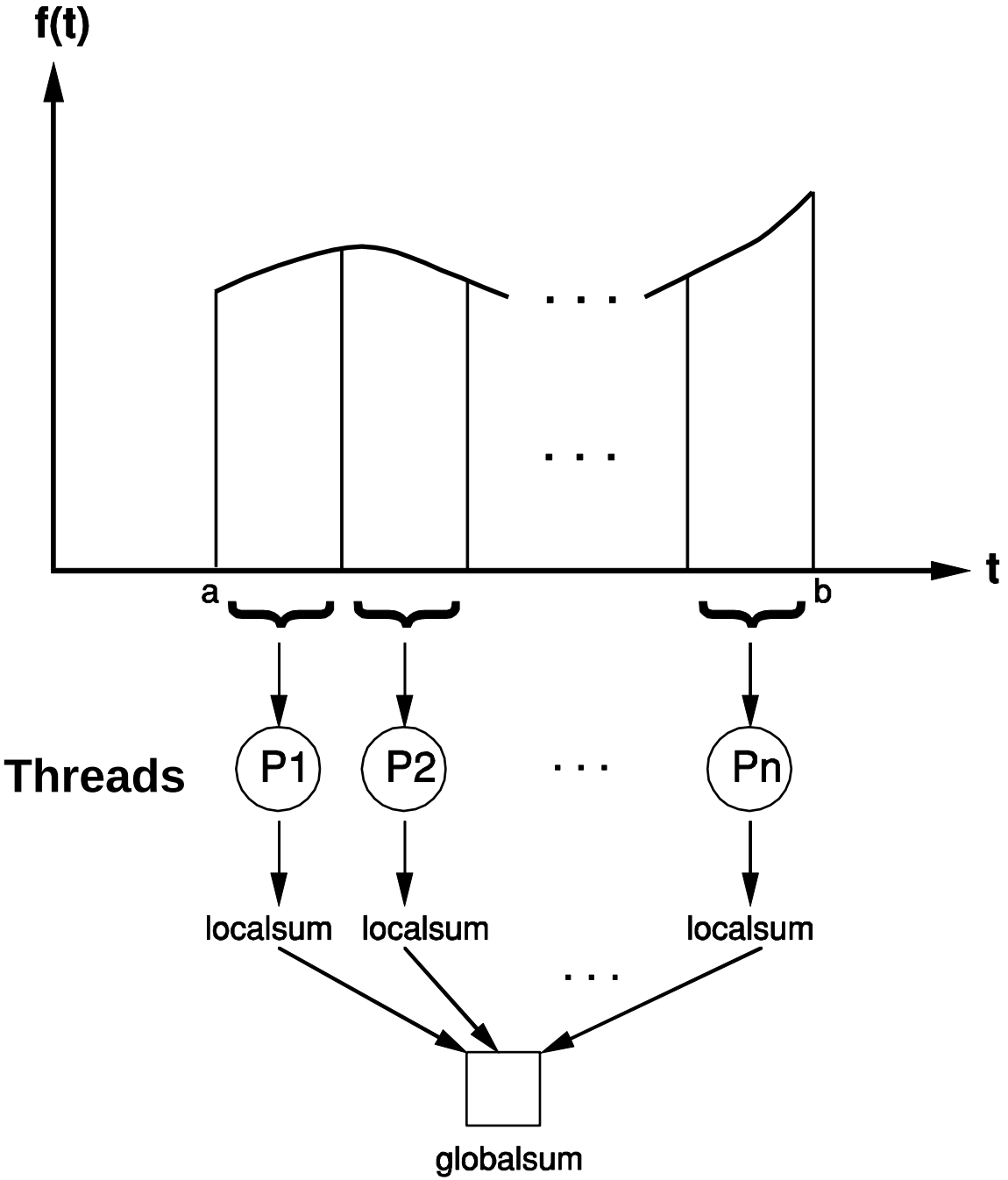
$$\int_a^b f(t)\,dt$$

$$w * [\ f(a)/2 + f(a+w) + f(a+2w) + \ldots + f(a+nw) + f(b)/2\ ]$$

Trapezoid Rule for Numerical Integration

Sequential Numerical Integration

```
sum = 0.0;
t = a;
for (i = 1;, i <= n; i++) {
  t = t + w;  /*Move to next point*/
  sum = sum + f(t); /*Add point to sum*/
}
sum = sum + (f(a) + f(b)) / 2;
answer = w * sum;
```



Parallel Numerical Integration

```
#define n ...  /*number of sample points*/
float a, b, sum, t, w;
int i, n;

main( ) {
  w = (b - a)/n;
  sum = 0.0;
  #pragma omp parallel for private(t) reduction(+:sum)
    for (i = 1; i <= n; i++) {
      t = a + i * w;  /*position of point*/
      sum = sum + f(t); /*Add point to sum*/
    }
  sum = sum + (f(a) + f(b)) / 2;
  answer = w * sum;
}
```

Parallel Numerical Integration

Reductions

`reduction( <op> : <variable> )`

`<variable>` is any shared variable

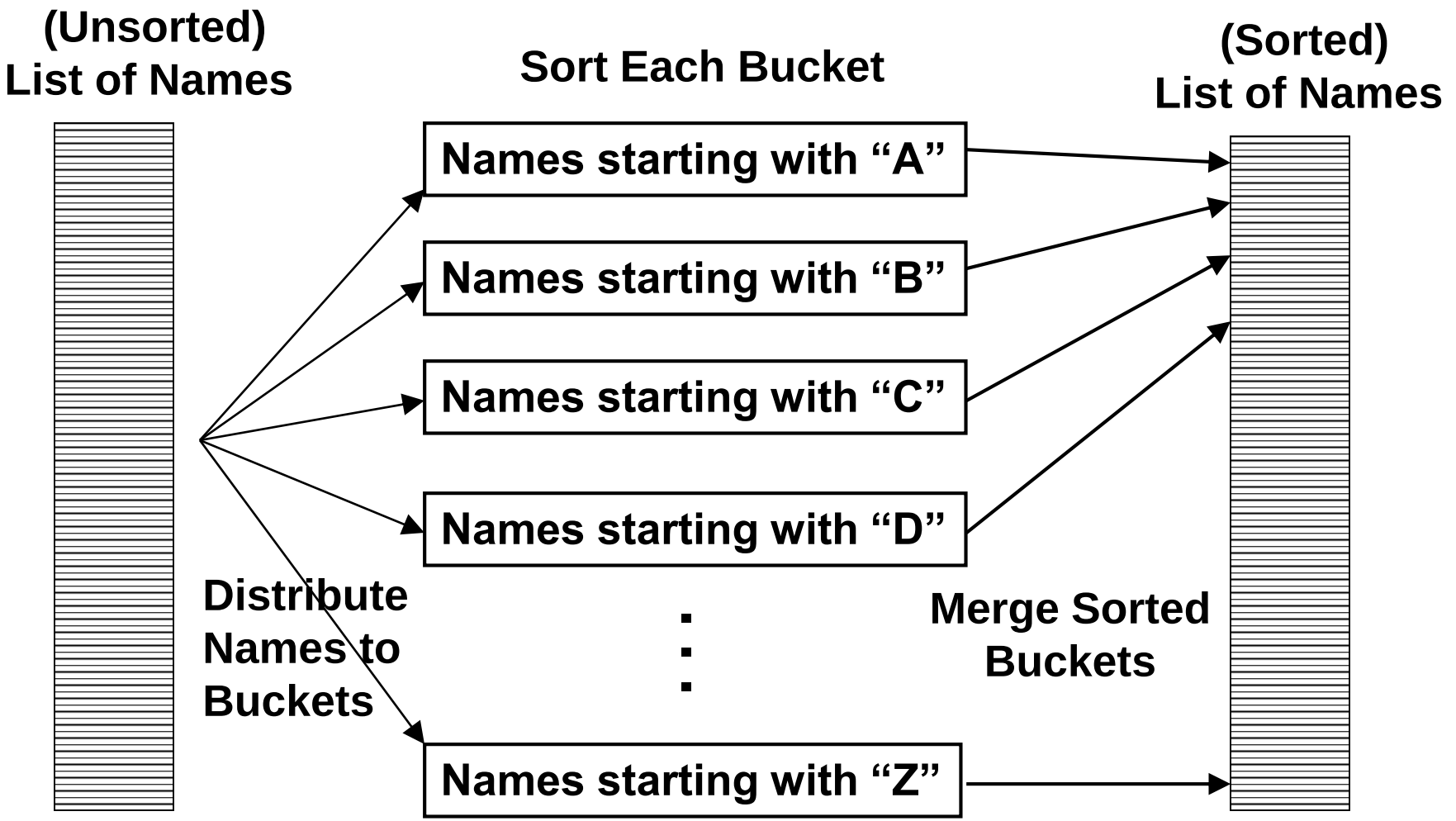
`<op>` is one of following:

- **`+`** sum
- **`*`** product
- **`&&`** logical and
- **`||`** logical or

(also, bit operations allowed)

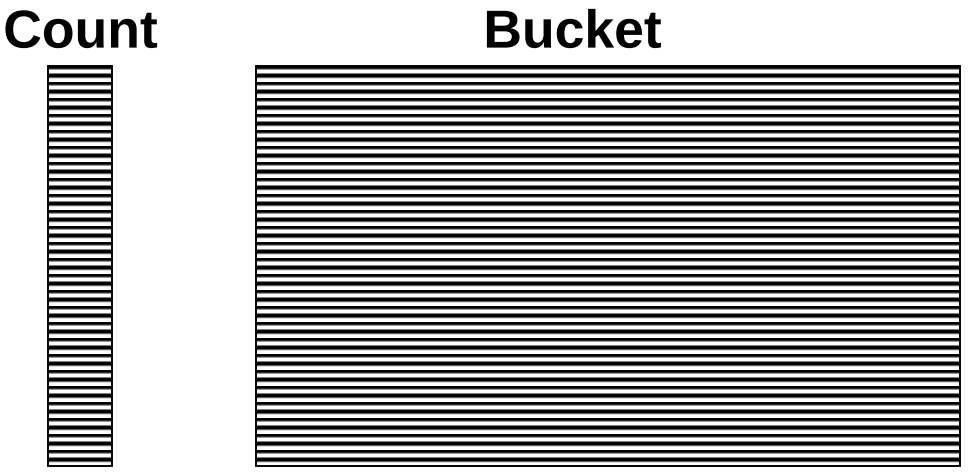
Programming Project: Bucket Sort

Programming Project: Bucket Sort

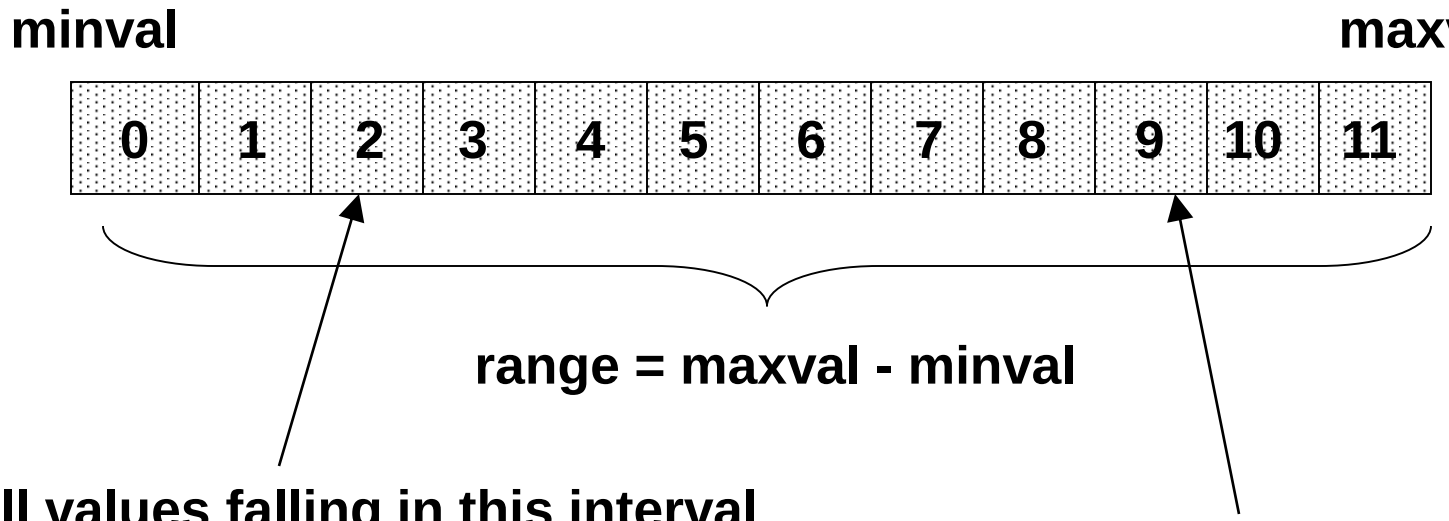


Bucket Sort
Data Structures

```
#define n 10000000 /*length of list*/
#define bsize 1000000  /*size of buckets*/
#define m 100  /*number of buckets*/
int list[n];  /*unsorted list of integers*/
int final[n]; /*sorted list of integers*/
int bucket[m][bsize]; /*buckets*/
int count[m]; /*number of items stored in bucket*/
```



Selecting the Bucket




```
range = maxval - minval + 1;
/* determine destination bucket for list[i] */
bnum = (int)((float)m * ((float)(list[i]-minval)/range));
```

Bucket Sort Phases

- **Distribution Phase**: Put each list element into the proper bucket.
- **Sort Phase**: Sort the contents of each bucket individually.
- **Merge Phase**: Merge the sorted buckets into final sorted list.

Helpful Code

```
/*initialize the list*/
for ( i = 0; i < n; i++ ) list[i] = rand( );

/*function to compare two integers*/
/* returns -1 if p < q, 0 if p = q, 1 if p > q */
int lt(const void *p, const void *q) {
      return (*(int *)p - *(int *)q);
}

/*sort bucket b using quick sort function*/
qsort(bucket[b], count[b], sizeof(int), lt );
```

Your Assignment

- Write a C program to do the Bucket Sort.
- Test program using list of 10,000,000 numbers. Record the execution time.
- Use OpenMP to create a Multi-Threaded version of your program.
- Test the Multi-Threaded version and compare the execution time to the original program.
- Determine the execution time (and speedup) of each of three Phases of the program separately.
- Write two parallel versions of the Distribution Phase (see next slide) and explain the reasons for the differences.

Distribution Phase

- Write two versions of the multi-threaded Distribution Phase.

  Version 1: Use a single spinlock.

  Version 2: Use fine-grain locking (multiple locks).
- Record the execution time for each Version and compare to the sequential time. Explain in detail the reasons for the differences.

Turn in the Following

- Program listing of the parallel version
- Performance statistics: sequential time, parallel time, speedup, processor utilization
- Execution time for sequential and two parallel versions of Distribution Phase and explanation of reasons for the differences.

Scalable Version for Performance Testing

- Part 1 is the sequential Bucket Sort with output of execution time.
- Part 2 is parallel version with output of execution time.
- Set number of threads to number of cores using *omp_get_num_procs*( ).
- At the end, output speedup (sequential time divided by parallel time).

Slide Presentation on Multi-Core Programming

to accompany

Parallel Programming
CS 471

by Bruce P. Lester

Computer Science Department
Maharishi University of Management
Fairfield, Iowa 52556

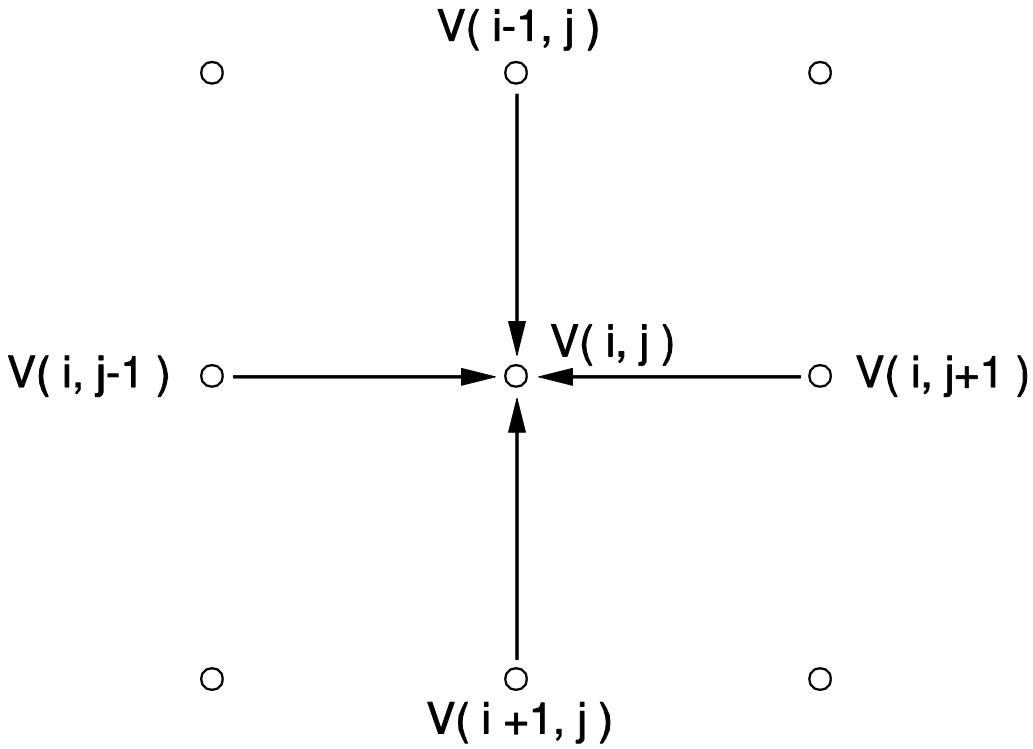
Lesson 3

Performance Improvements

Slide Presentation
to accompany
Parallel Programming
Bruce P. Lester

Laplace's Equation

$$\frac{\partial^2 v}{\partial^2 x} + \frac{\partial^2 v}{\partial^2 y} = 0$$



$$V(i, j) = \frac{V(i-1, j) + V(i+1, j) + V(i, j-1) + V(i, j+1)}{4}$$

Computing voltage at each point

Sequential Jacobi Relaxation Program

```
float  A[n][n], B[n][n], change;

main( ) {
  ...   /* array initialization */
  do {
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
       B[i][j] = (A[i-1][j] + A[i+1][j] +
                  A[i][j-1] + A[i][j+1]) / 4.0;
   done = 1;
   for (i = 0; i < n; i++) {
     for (j = 0; j < n; j++) {
       change = fabs(B[i][j] - A[i][j]);
       if (change > tolerance) done = 0;
       A[i][j] = B[i][j];
     }
   }
  } while (!done); /* convergence test */
}
```

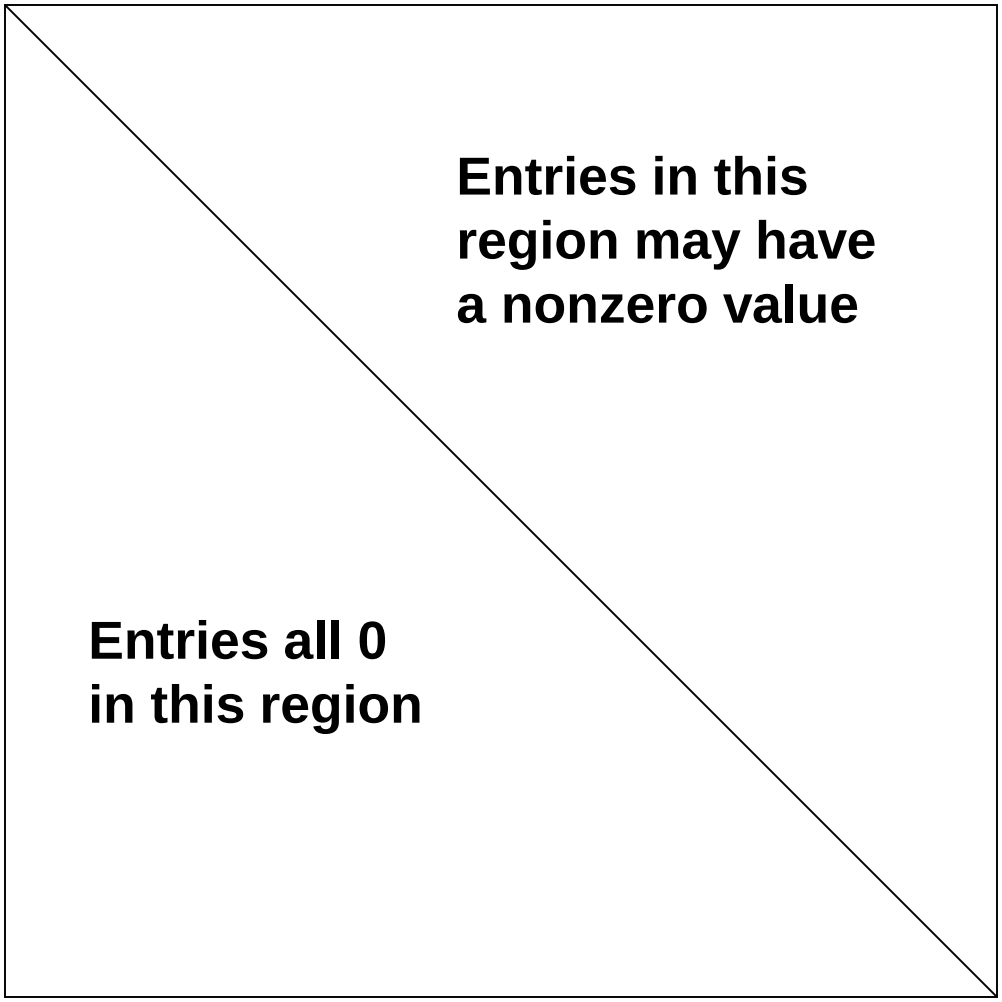
Parallel Jacobi Relaxation Program

```
main( ) {
  ... /* array initialization */
  do {
   #pragma omp parallel for private(j)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        B[i][j] = (A[i-1][j] + A[i+1][j] +
                   A[i][j-1] + A[i][j+1]) / 4.0;
    done = 1;
   #pragma omp parallel for private(j,change) reduction(&&:done)
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
        change = fabs(B[i][j] - A[i][j]);
        if (change > tolerance) done = 0;
        A[i][j] = B[i][j];
      }
    }
  } while (!done);
}
```

Time to Create Threads

- For a 1 Gigahertz processor, each clock cycle is 1 nanosecond (10^{-9} seconds).
- Cache Memory access time is about 10 nanoseconds.
- Main Memory access time is about 50 nanoseconds.
- Time for OpenMP “parallel” pragma is 1500 nanoseconds for each thread.
- Granularity of threads must be large enough to compensate for this overhead.

Upper Triangular Matrix



Initialize Upper Triangular Matrix

```
for ( i = 0; i < n; i++ )
  for ( j = i; j < n; j++ )
    a[i][j] = compute_value(i,j);
```

Earlier iterations of outer i loop have much longer execution time than later iterations. If make loop parallel, some threads will much computing to do, and others will have little to do. This is called a Load Imbalance.

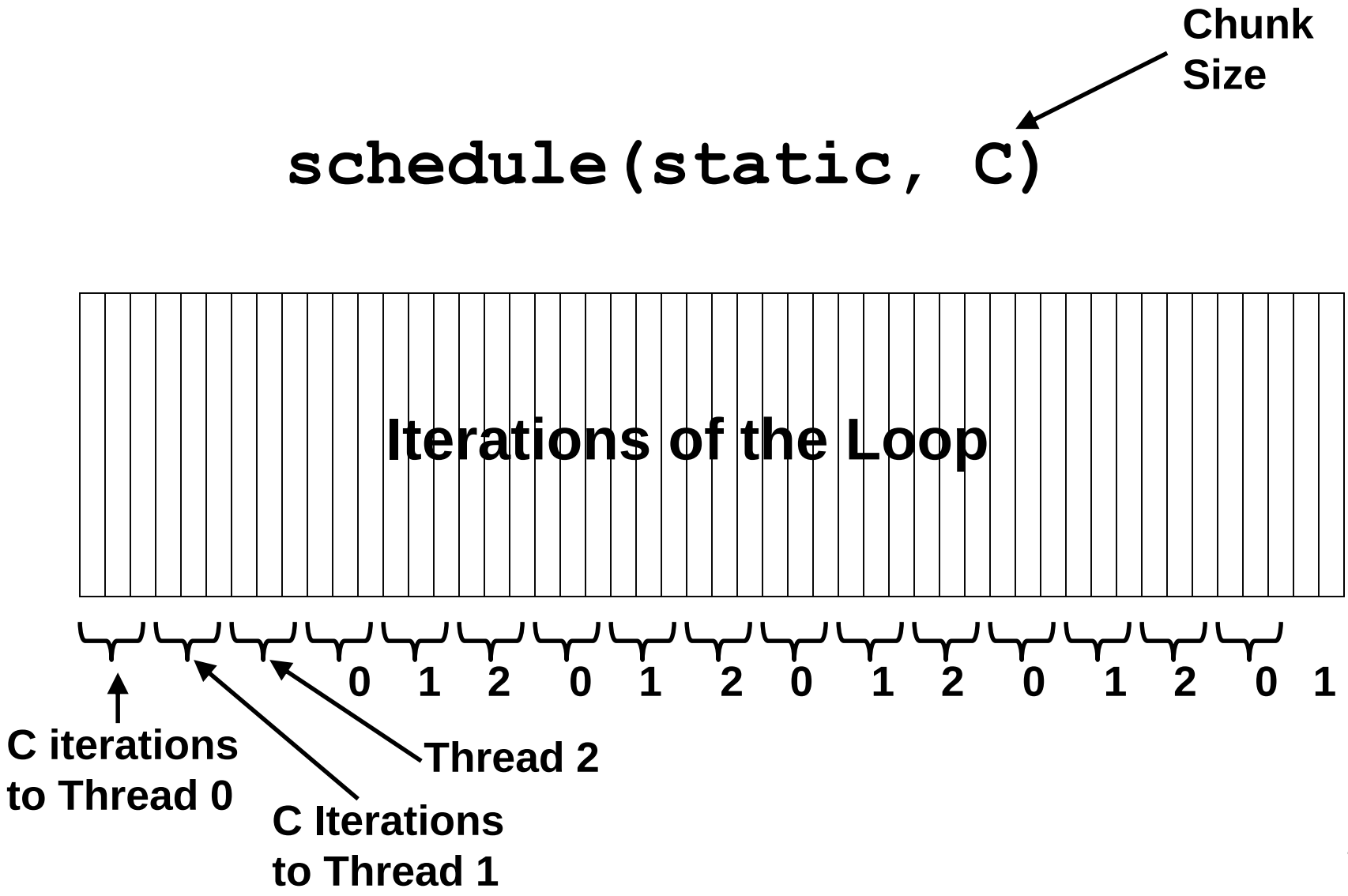
Scheduling Loops

- **Static Schedule**: All iterations are allocated to threads before they execute any iterations
- **Dynamic Schedule**: Some of the iterations allocated to threads at start of execution. Threads that complete their iterations are eligible to get additional work.

Performance Tradeoff

- Static Scheduling has low overhead, but may have high load imbalance.
- Dynamic Scheduling has higher overhead, but can reduce load imbalance.

Static Scheduling



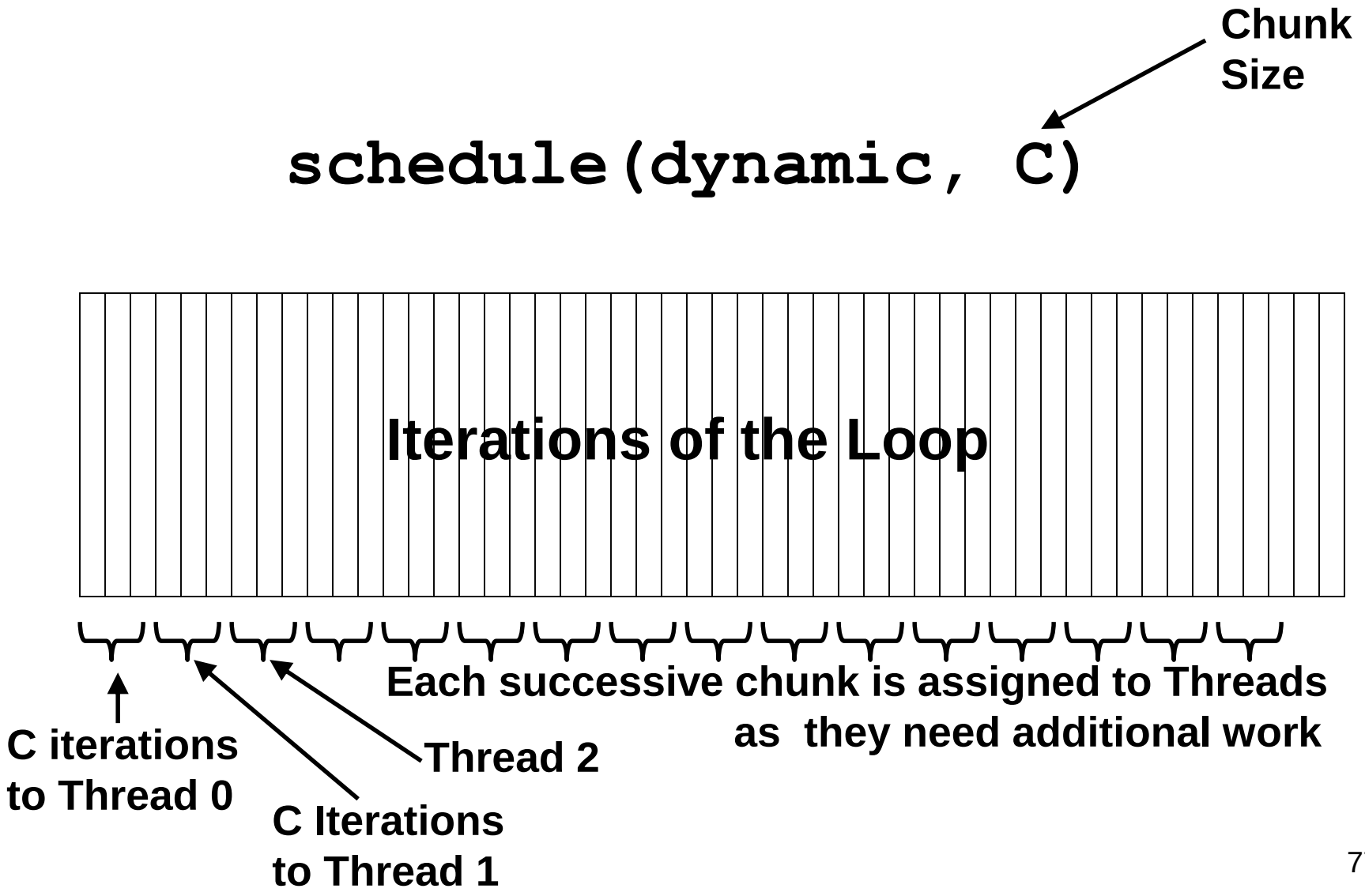
Initialize Upper Triangular Matrix Using Static Scheduling

```
#pragma omp parallel private(j) schedule(static,1)
   for ( i = 0; i < n; i++ )
     for ( j = i; j < n; j++ )
       a[i][j] = compute_value(i,j);
```

i = 0: Thread 0
i = 1: Thread 1
i = 2: Thread 2
i = 3: Thread 0
i = 4: Thread 1
i = 5: Thread 2
i = 6: Thread 0
...

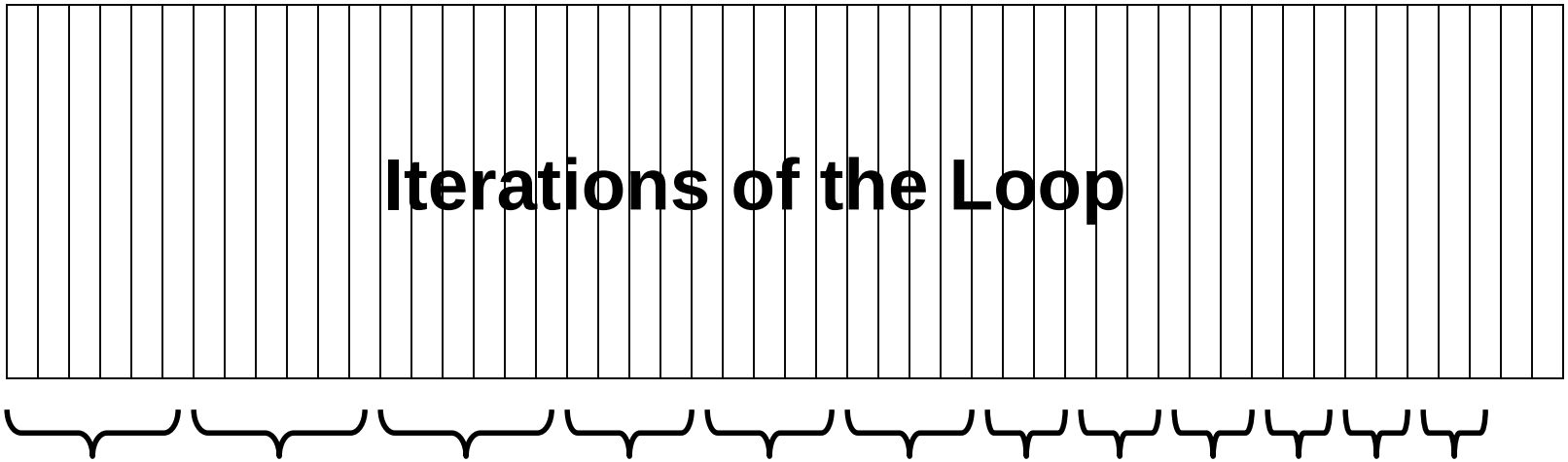
Interleaved assignment of successive loop iterations balances the load among the threads.

Dynamic Scheduling



Guided Scheduling

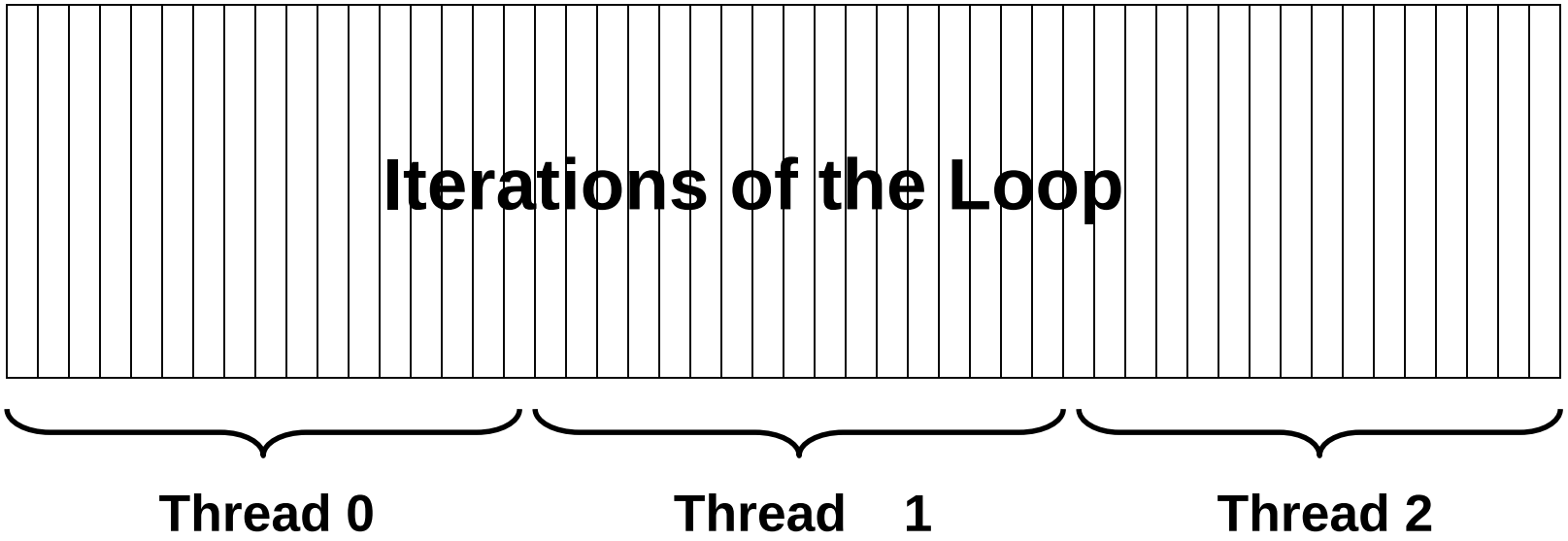
`schedule(guided)`



Large chunks initially assigned to Threads

Additional chunks of progressively smaller size assigned dynamically to Threads as needed

Default Method
(if schedule is not specified)



Equal number of iterations
statically assigned to each Thread

Which Schedule is Best?

- Optimal scheduling method and chunk size depends not only on the characteristics of the program, but also the properties of the operating system, processor, and memory system.
- For a particular program, can run some performance benchmarks to help select the best scheduling method.
- If the loop iterations have equal duration, then the default method is best.

Data Dependence

- Statement *S2* is said to be ***data dependent*** on statement *S1* if:

  1) There is a possible execution path such that statement *S1* and *S2* both reference the same memory location *L*.

  2) The execution of *S1* that references L occurs before the execution of *S2* that references *L*.

Data Dependence

- In order for *S2* to depend upon *S1*, it is necessary for some execution of *S1* to write to a memory location *L* that is later read by an execution of *S2*. This is also called *flow dependence*.
- Other dependencies exist when two statements write the same memory location *L*, called an *output dependence.*
- Read occurs before a write, called an *anti-dependence.*

Loop-carried Dependence

- **Loop-carried dependence**
  S1 can reference the memory location *L* on one iteration of a loop; on a subsequent iteration *S2* can reference the same memory location *L*.
- **Loop-independent dependence**
  S1 and *S2* can reference the same memory location *L* on the same loop iteration, but with *S1* preceding *S2* during execution of the loop iteration.

Loop-carried Dependence

	iteration k	iteration $k+d$
	Loop-carried flow dependence	
statement S_1	write L	
statement S_2		read L
	Loop-carried anti-dependence	
statement S_1	read L	
statement S_2		write L
	Loop-carried output dependence	
statement S_1	write L	
statement S_2		write L

Loop-carried Dependence

```
// Do NOT do this. It will fail due to loop-carried
// dependencies.

x[0] = 0;
y[0] = 1;

#pragma omp parallel for private(k)

      for ( k = 1; k < 100; k++ ) {
         x[k] = y[k-1] + 1;  // S1
         y[k] = x[k-1] + 2;  // S2
      }
```

Loop-carried Dependence

- Because OpenMP directives are commands to the compiler, the compiler will thread this loop (see previous slide). However, the threaded code will fail because of loop-carried dependence.
- The only way to fix this kind of problem is to rewrite the loop, or to pick a different algorithm that does not contain the loop-carried dependence.

Homework Exercises

OMP1. Under what conditions will dynamic loop scheduling perform better than static scheduling?

OMP2. Give an example of a simple parallel for-loop that would probably execute faster using schedule (static,1) than the default scheduling method. Your example should not have nested loops.

OMP3. What performance advantages does guided scheduling offer over dynamic scheduling?

Homework Exercises

OMP4. The following program implements Matrix-Vector Multiplication (a = B * c), where *a* and *c* are vectors, and *B* is a matrix.

```
int m, n;
double A[n], B[n][n], C[n]
int i, j;
for (i=0; i<m; i++) {
  A[i] = 0.0;
  for (j=0; j<n; j++)
    A[i] += B[i*n+j]*C[j];
  }
}
```

Homework Exercises

OMP 4 (con't) Parallelize the for loop in line 05 by providing the appropriate line in OpenMP. Which variables have to be private and which variables have to be shared?

OMP 5. Which schedule would you propose for this parallelization? Explain your answer and briefly list the pros and contras for each of the three OpenMP worksharing schedules (namely *static*, *dynamic* and *guided*) for this specific case.

Homework Exercises

OMP6. Make each of the following loops parallel, or explain why it is not suitable for parallel execution:

```
for (i=0; i < (int) sqrt(x); i++) {
  a[i] = 2.3 * i;
  if (i < 10) b[i] = a[i];
}
--------------------------------
flag = 0;
for (i=0; (i < n) && (!flag); i++) {
  a[i] = 2.3 * i;
  if (a[i] < b[i]) flag = 1;
}
--------------------------------
for (i=0; i < n; i++)
  a[i] = foo(i);
--------------------------------
```

```
for (i=0; i < n; i++) {
  a[i] = foo(i);
  if (a[i] < b[i]) a[i] = b[i];
}
```

```
for (i=0; i < n; i++) {
  a[i] = foo(i);
  if (a[i] < b[i]) break;
}
```

```
dotp = 0;
for (i=0; i<n; i++)
  dotp += a[i] * b[i];
```

```
for (i=k; i < 2*k; i++)
  a[i] = a[i] + a[i-k];
```

```
for (i=k; i<n; i++)
  a[i] = b * a[i-k];
```

Homework Exercises

OMP7. Run each of the following programs and explain the differences in the output.

```
#define n 20
omp_set_num_threads(3);
#pragma omp parallel for private(tid) schedule(static,1)
for (i=0; i<n; i++) {
  tid = omp_get_thread_num();
  printf("Thread %d executing iteration %d\n", tid, i);
}
```

```
#define n 20
omp_set_num_threads(3);
#pragma omp parallel for private(tid)
for (i=0; i<n; i++) {
  tid = omp_get_thread_num();
  printf("Thread %d executing iteration %d\n", tid, i);
}
```

Homework Exercises

OMP 8. For each of the following portions of code, is there any loop-carried dependence? If yes, is it *flow*, *anti*, or *output* dependence?

(a)
```
for (i = 0; i < U1; i++)
    for (j = 0; j < U2; j++)
     a[i+4-j] = b[2*i-j] + i*j;
```

(b)
```
for(j = 0; j < n; j++)
      a[j] = a[j-1];
```

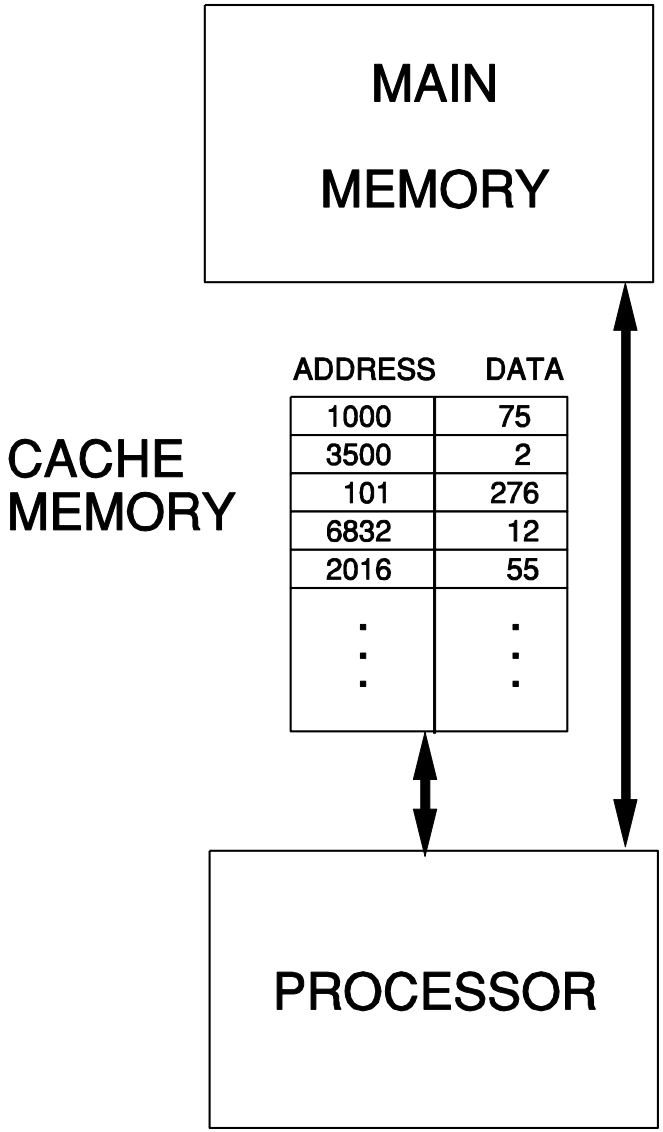
(c)
```
for(j = 0; j < n; j++) {
     c[j] = j;
     c[j+1] = 5;
    }
```

(d)
```
for(j = 0; j < n; j++)
      b[j] = b[j+1];
```

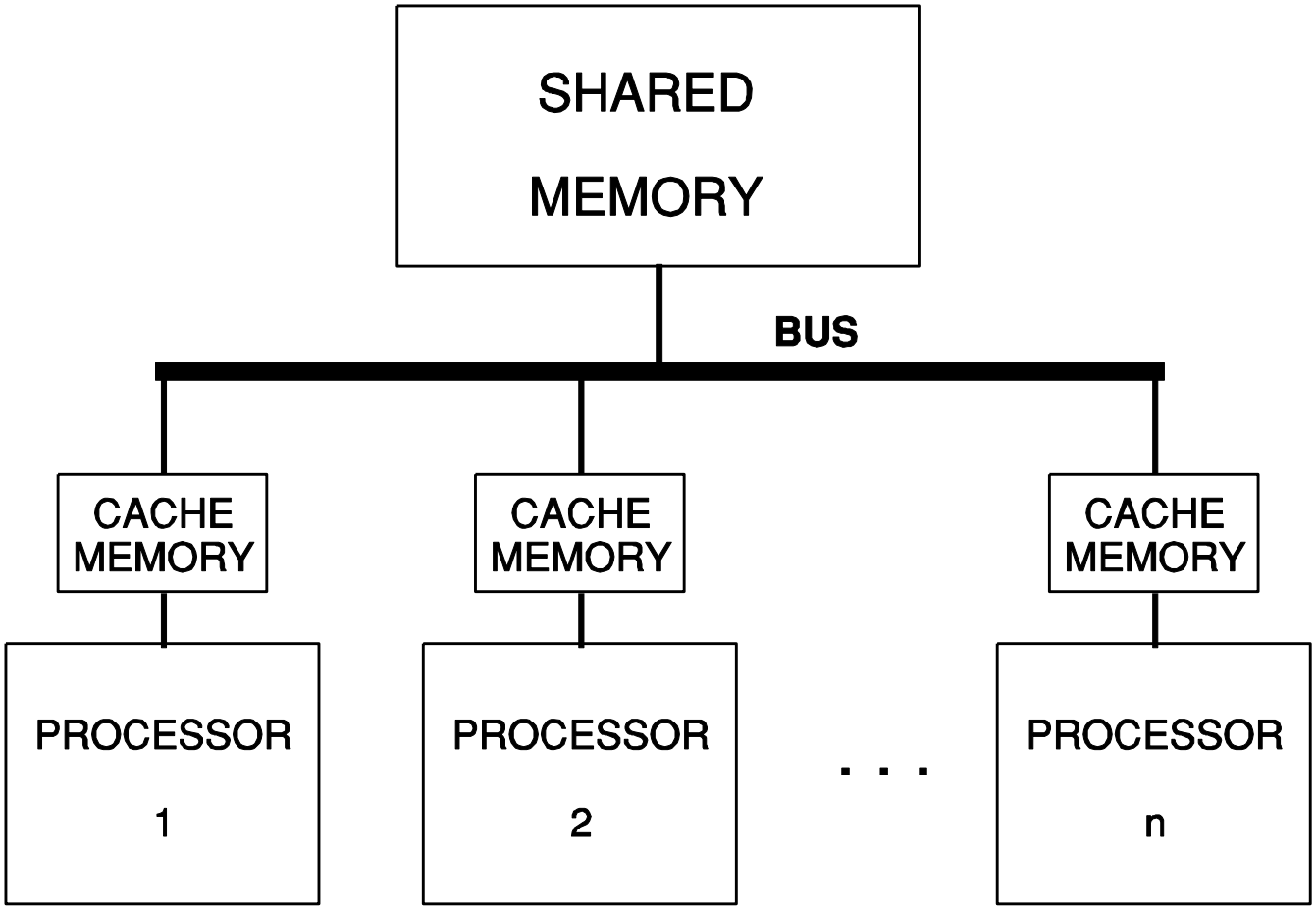
Lesson 4

Memory Issues

Slide Presentation
to accompany
Parallel Programming
Bruce P. Lester



Cache memory in uniprocessor computer



Shared memory with multiple modules

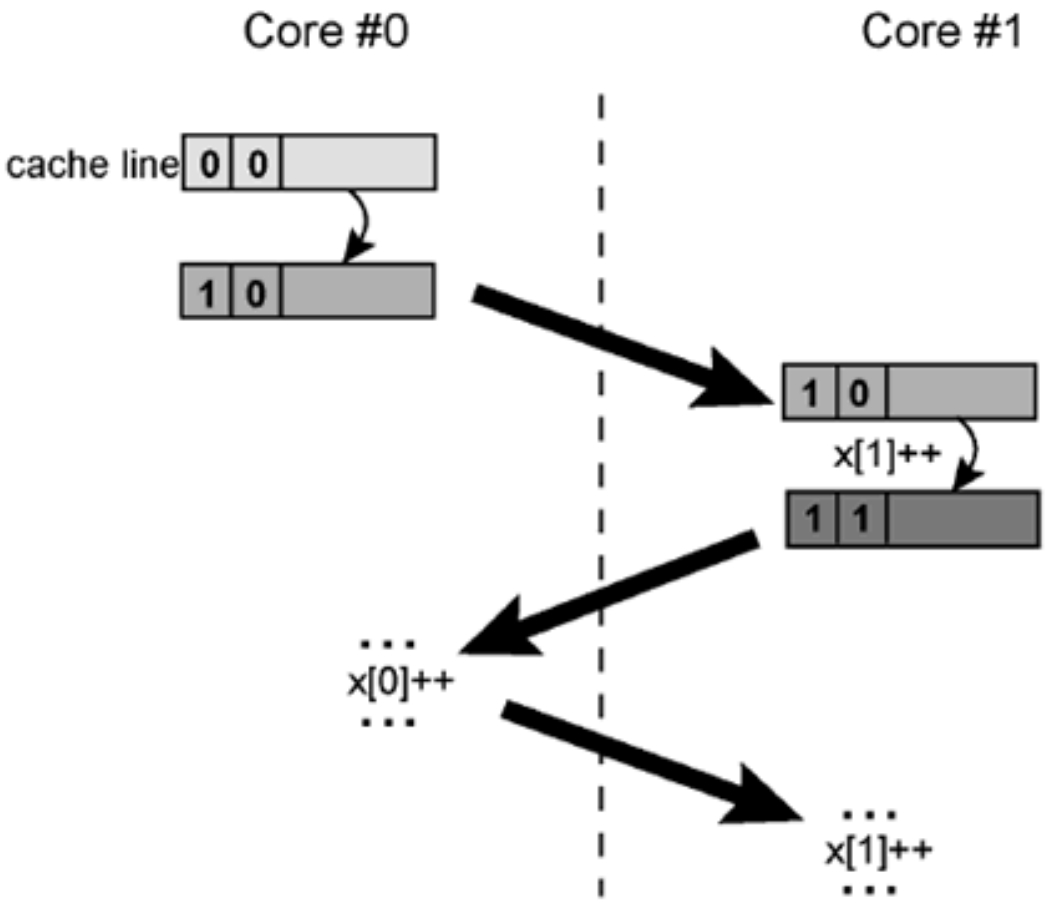
Cache Memory Performance

- In multi-core processors, each core has its own cache memory.
- Whenever a core reads/writes a memory location, it is copied into the cache of that core.
- If all cores only read a memory location, many caches may contain a copy of that memory location.
- However, it any core writes to a memory location, the copies in the other caches must be invalidated to maintain *cache coherence*.
- If many cores are both reading and writing the same memory location, the copy in the cache will be invalidated quickly, and most read/write operations will have to use the main memory rather than the cache.

False Sharing

- When a data value is copied from the main memory to a cache memory, a whole block of data is copied, called a *cache line* (usually 64 bytes long).
- Two different memory locations (A and B) may be in the same cache line.
- If one core is frequently writing A, and another core is frequently writing B, the cores will be continually invalidating each other's cache line. This is called the ***cache ping pong*** effect.
- False sharing can be corrected by creating more space between A and B in memory, so they do not occupy the same cache line.

Cache Ping Pong Effect



Programming Project: Matrix Multiplication

```
for (i = 0; i < n; ++)
   for (j = 0; j < n; j++)
     for  (k = 0; k < n; k++)
       C[i][j] += A[i][k] * B[k][j];
```

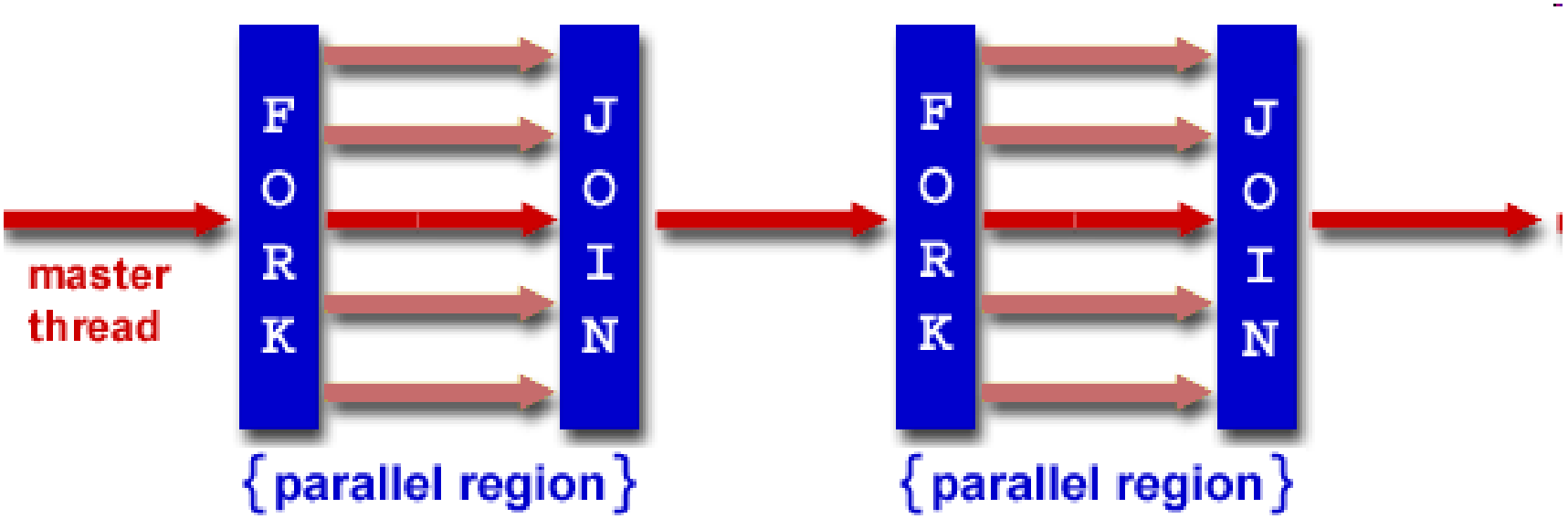
- Run and test a sequential version.
- Invert the j, k loops and run again, comparing the execution time (explain reasons for the change).
- Write and test performance of parallel versions of original and inverted loops.

Lesson 5

Parallel Regions

Slide Presentation
to accompany
Parallel Programming
Bruce P. Lester

Parallel Regions



OpenMP uses the fork-join model of parallel execution

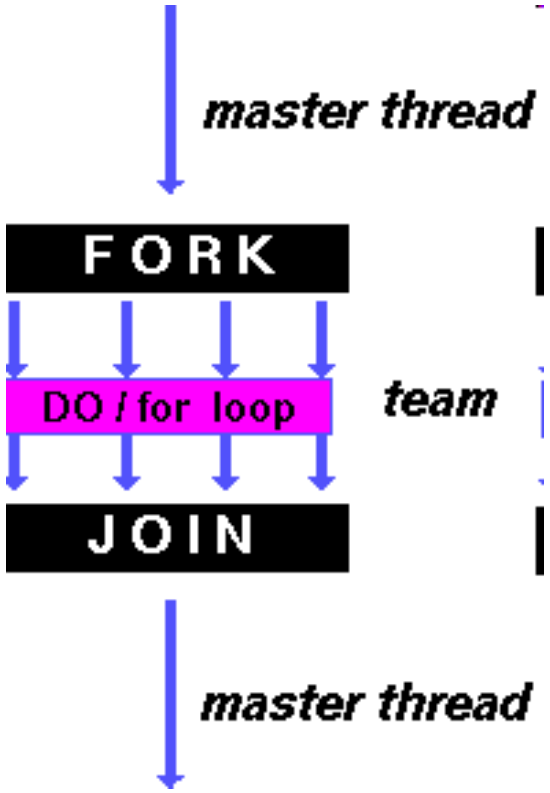
Parallel Regions

- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK:** the master thread then creates a ***team*** of parallel threads
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

Parallel Regions Example

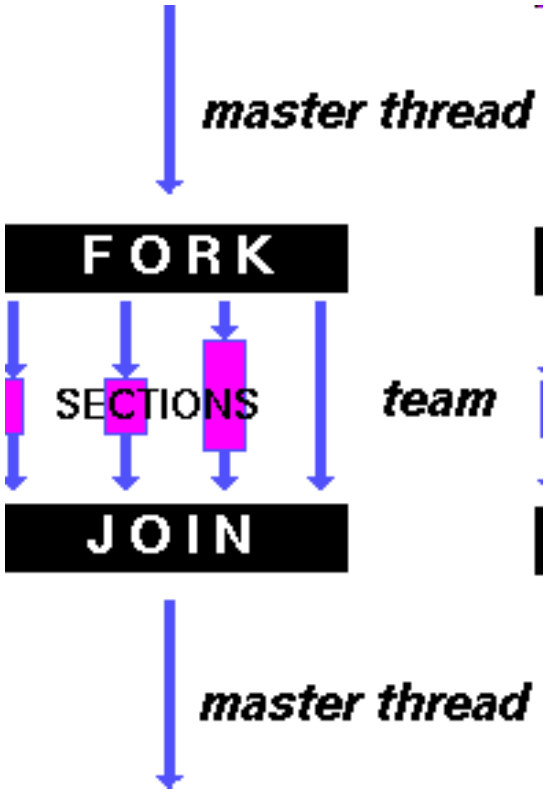
```
main () {
  int nthreads, tid;
  /*Fork team of threads with private tid variable*/
  #pragma omp parallel private(tid) {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } /*All threads join master thread and terminate*/
}
```

Parallel Regions *for* Construct



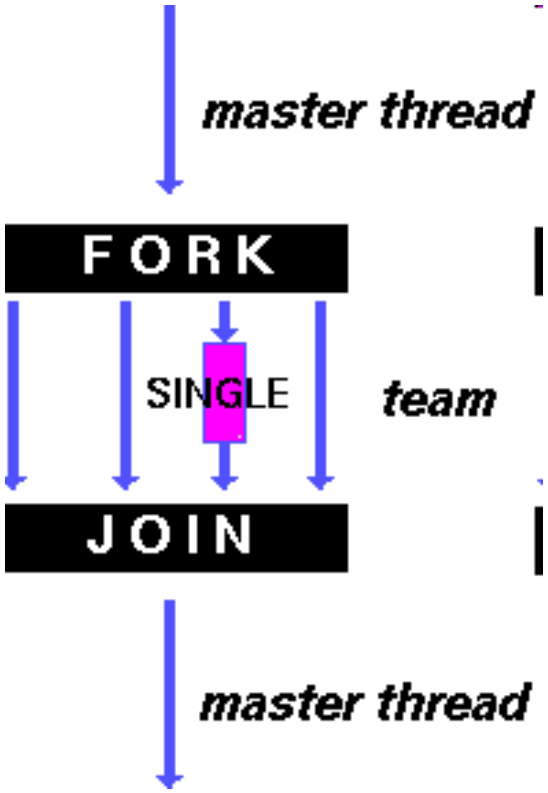
for construct - shares iterations of a loop across the team.
Represents a type of "data parallelism"

Parallel Regions *sections* Construct



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

Parallel Regions *single* Construct



Single – serializes a section of code
Only executed by one of the threads

Constructs for Parallel Regions

- **`#pragma omp for`**: divides the loop iterations among the threads
- **`#pragma omp sections {`**
  `#pragma section { ... }`
  `#pragma section { ... }`
  `#pragma section { ... }`
  `}`
  - Assigns one section to each thread
  - If more sections than threads, threads assigned new sections after they finish earlier ones
- **`#pragma omp single { ... }`**
  Executed by only one thread

Constructs for Parallel Regions

- **`#pragma omp critical`**
  `{ ... }`

- The Threads execute the code enclosed in brackets one at a time.
- The enclosed code becomes an Atomic Operation.
- This *critical section* may replace the use of locks.

- A useful library function:

Function for each Thread in a Parallel Region to find its identifier number (numbered 0 to *n*-1 for *n* Threads):

```
int omp_get_thread_num( )
```

Ordinary Code in Parallel Region

- Ordinary sequential code inside parallel region is executed by each thread in parallel.
- Example:

```
#pragma omp parallel  {
  int tid = omp_get_thread_num();
  if (tid == 0)  y = f(x);
    else z = g(x);
  ...
}
```

Master Thread

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
- It is illegal to branch into or out of a parallel region

For Construct Example

```
#define CHUNKSIZE 100
#define N 1000
main () {
  int i, chunk;
  float a[N], b[N], c[N];
  for (i=0; i < N; i++)
     a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
  #pragma omp parallel private(i) {
    #pragma omp for schedule(dynamic,chunk)
      for (i=0;i < N;i++) c[i] = a[i] + b[i];
 }
```

Sections Construct Example

```
#define N 1000
main () {
  int i;
  float a[N], b[N], c[N], d[N];
  for (i=0; i < N; i++) { /* Initialization */
    a[i] = i * 1.5; b[i] = i + 22.35;
  }
  #pragma omp parallel private(i) {
    #pragma omp sections nowait {
      #pragma omp section
        for (i=0; i < N; i++) c[i] = a[i] + b[i];
      #pragma omp section
        for (i=0; i < N; i++) d[i] = a[i] * b[i];
    }
  }
}
```

Homework Exercise

OMP 9 Write a program with two threads that increment x[0] and x[1] respectively, many times. Then change the second thread to increment x[100] instead. Record the execution times to observe the false sharing and cache ping pong effect.

Programming Project: Numerical Integration

Numerical Integration

- In this project, you will write a Parallel Numerical Integration program using C and OpenMP.
- Use the Numerical Integration program shown in Lesson 2 of these slides as a model for your OpenMP program.
- To create the parallelism use OpenMP Parallel Regions.
- Do not use #pragma omp for
- Do not use #pragma omp parallel for
- Use #pragma omp critical, instead of locks.

Numerical Integration

- For a test case, use the following function *f(x)* in the range a = 0 to b = 2:

```
f(x) = sqrt(4.0 – x*x)
```

- The resulting value of the integral should be an approximation to the value of pi: 3.1415926535
- Use a total of 100,000,000 sample points for computing the integral.

Numerical Integration

- Write a sequential version also, and compute the speedup of the parallel version.
- Prepare an executable version that is scalable to any number of cores.