西北工業大學

Parallel Computing

Threads and OpenMP

Outline

- Shared Memory Hardware
- Parallel Programming with Threads
- Parallel Programming with OpenMP
  - See http://www.nersc.gov/nusers/help/tutorials/openmp/
- Summary

PARALLEL PROGRAMMING IN OPENMP:

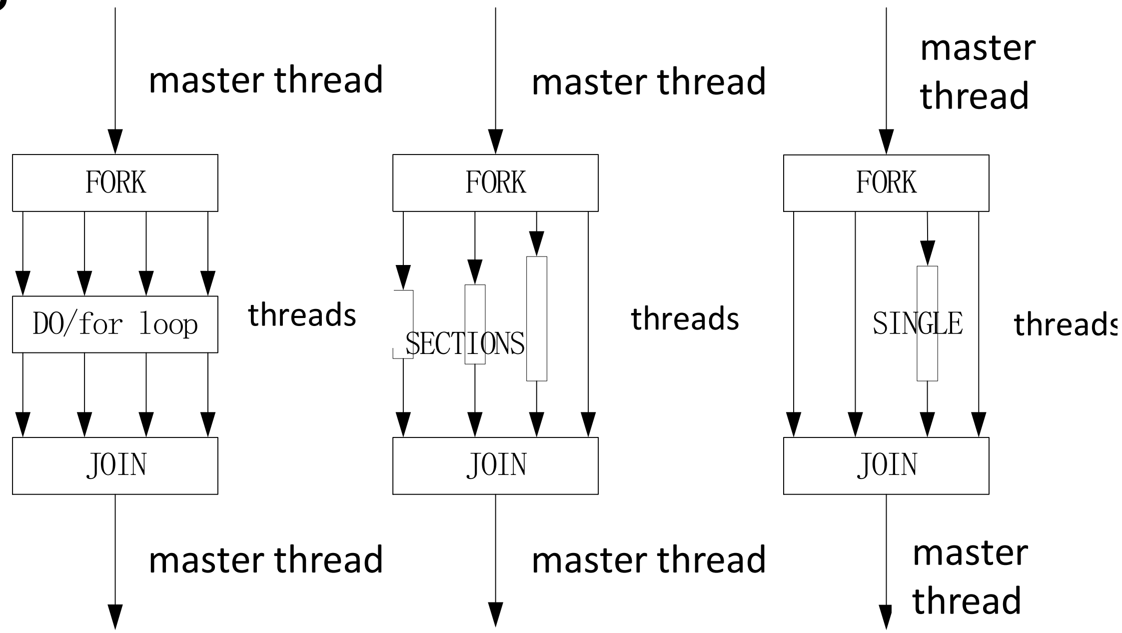
WORK-SHARING CONSTRUCT

Work-sharing Construct

```
#pragma omp parallel
#pragma omp for
      for (I=0;I<N;I++){
            Do_Work(I);
      }
```

- Splits loop iterations into threads
- Must be in the parallel region
- Must precede the loop

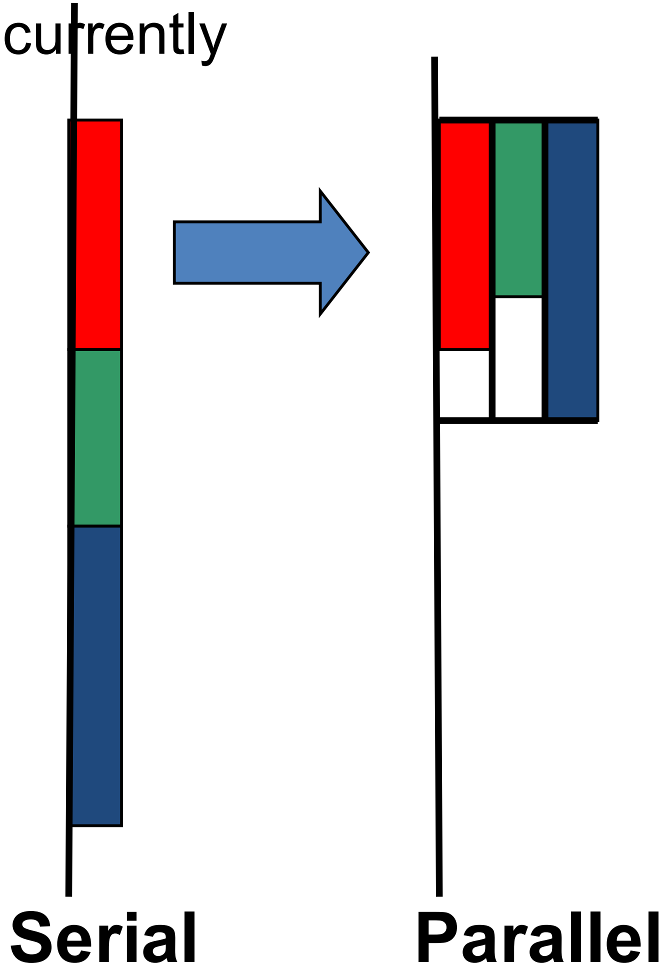
- **loop construct**
- **sections construct**
- **single construct**



Parallel Sections

- Independent sections of code can execute concurrently

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



section_count 1
section_count 1

```
#include <omp.h>
#include <stdio.h>
#define NT 4
int main( ) {
    int section_count = 0;
    omp_set_dynamic(0);
    omp_set_num_threads(NT);
#pragma omp parallel
#pragma omp sections firstprivate( section_count )
{
#pragma omp section
    {
        section_count++;
        /* may print the number one or two */
        printf( "section_count %d\n", section_count );
    }
#pragma omp section
    {
        section_count++;
        /* may print the number one or two */
        printf( "section_count %d\n", section_count );
    }
}
    return 0;
}
```

```
void main(int argc, char *argv)
{
     #pragma omp parallel sections
    {
          #pragma omp section
          printf("section 1 ThreadId = %d\n", omp_get_thread_num());
         #pragma omp section
          printf("section 2 ThreadId = %d\n", omp_get_thread_num());
         #pragma omp section
         printf("section 3 ThreadId = %d\n", omp_get_thread_num());
         #pragma omp section
         printf("section 4 ThreadId = %d\n", omp_get_thread_num());
    }
}
```

```
section 1 ThreadId = 0
section 2 ThreadId = 2
section 4 ThreadId = 3
section 3 ThreadId = 1
```

```
void main(int argc, char *argv)
{
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            printf("section 1 ThreadId = %d\n", omp_get_thread_num());
            #pragma omp section
            printf("section 2 ThreadId = %d\n", omp_get_thread_num());
        }
        #pragma omp sections
        {
            #pragma omp section
            printf("section 3 ThreadId = %d\n", omp_get_thread_num());
            #pragma omp section
            printf("section 4 ThreadId = %d\n", omp_get_thread_num());
        }
    }
}
```

```
section 1  ThreadId = 0
section 2  ThreadId = 1
section 4  ThreadId = 1
Section 3 ThreadId = 0
```

Single Construct

- Denotes block of code to be executed by only one thread
  - First thread to arrive is chosen
- Implicit barrier at end

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp single
   {
     ExchangeBoundaries();
   }  // threads wait here for single
   DoManyMoreThings();
}
```

Master Construct

- Denotes block of code to be executed only by the master thread
- No implicit barrier at end

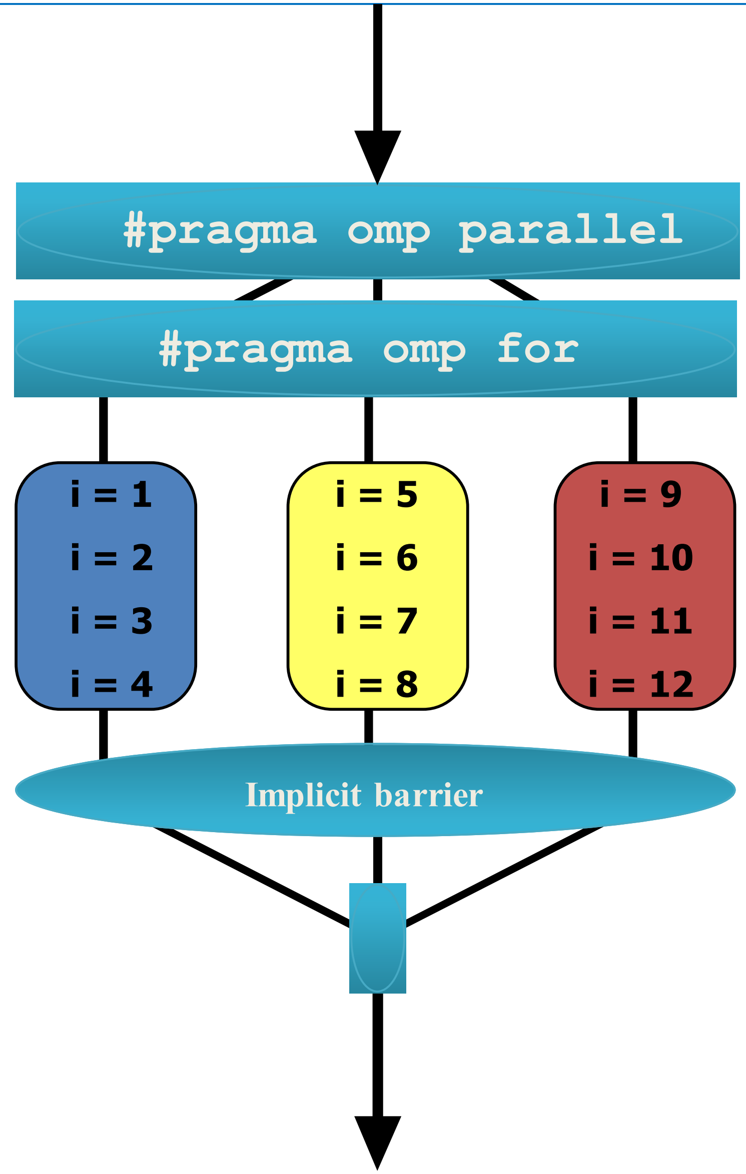
```
#pragma omp parallel
{
   DoManyThings();
#pragma omp master
   {                    // if not master skip to next stmt
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```

Implicit Barriers

- Several OpenMP* constructs have implicit barriers
  - `parallel`
  - `for`
  - `single`
- Unnecessary barriers hurt performance
  - Waiting threads accomplish no work!
- Suppress implicit barriers, when safe, with the `nowait` clause

Work-sharing Construct

```
#pragma omp parallel
#pragma omp for
   for(i = 1, i < 13, i++)
      c[i] = a[i] + b[i]
```



- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

Loop worksharing Constructs

A motivating example

Sequential code

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

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
         Nthrds =
        omp_get_num_threads(); istart =
        id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region and a worksharing for construct

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

Combining pragmas

- These two code segments are equivalent

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

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

Loop Construct

- The syntax of the loop construct is as follows:

#pragma omp for *[clause[[,] clause] ...] new-line*
for-loops

where clause is one of the following:

private(*list)*
firstprivate(*list)*
lastprivate(*list)*
linear(*list[: linear-step])*
reduction(*reduction-identifier : list)*
schedule(*[modifier [, modifier]:]kind[, chunk_size])*
collapse(*n)*
ordered*[(n)]*
Nowait

- **NO WAIT / nowait**: If specified, then threads do not synchronize at the end of the parallel loop.
- **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

- The **schedule clause specifies**
  - how iterations of these associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team.
  - **Static: schedule(static, chunk_size)**
    - divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.
  - **Dynamic: schedule(dynamic, *chunk_size)***
    - . Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.
  - **Guided: schedule(guided, *chunk_size)***
    - Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned.
  - **RUNTIME**
    - The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
  - **AUTO**
    - The scheduling decision is delegated to the compiler and/or runtime system.

loop work-sharing constructs:

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can “learn” from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
}
```

Note: loop index "i" is private by default

Remove loop carried dependence

```
int i,  A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
    int j = 5 + 2*(i+1);
    A[i] = big(j);
}
```

Nested loops

- **For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:**

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
   for (int j=0; j<M; j++) {
          .....
   }
}
```

Number of loops to be parallelized, counting from the outside

- **Will form a single loop of length NxM and then parallelize that.**
- **Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.**

example：

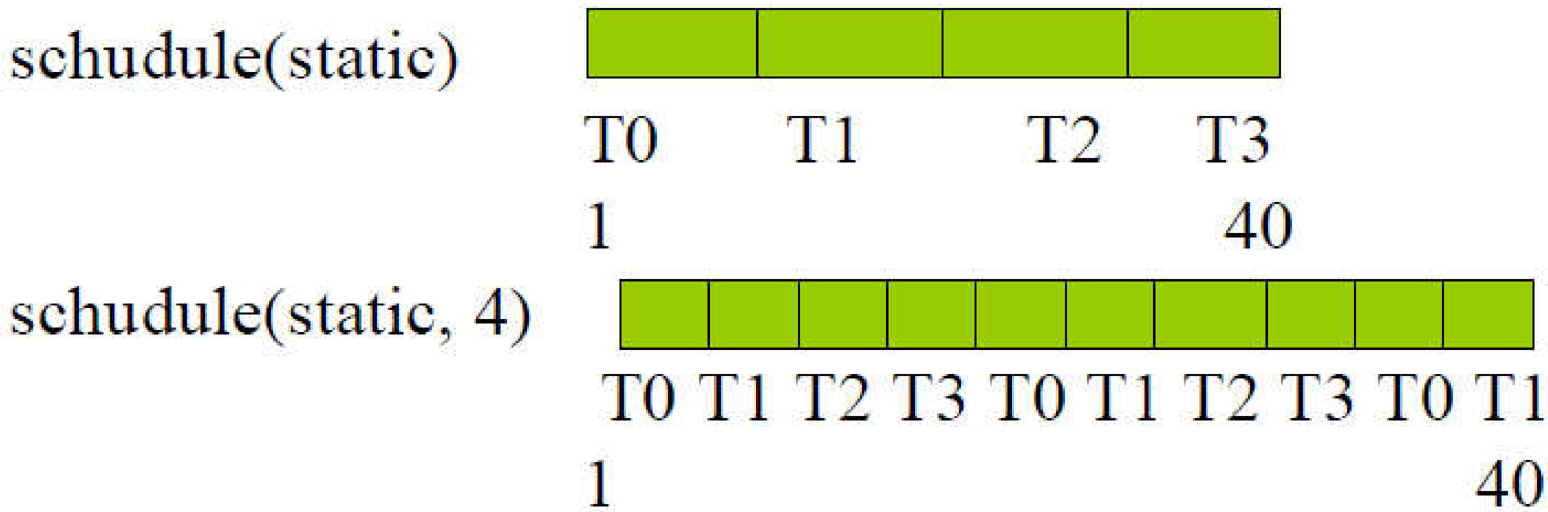
```
int i, j;
int a[100][100] = {0};
for ( i =0; i < 100; i++)
{
    for( j = i; j < 100; j++ )
    {
        a[i][j] = i*j;
     }
}
```

If the outer loop is paralleled, such as using 4 threads, there is 100 times computing difference when I is 0 and I is 99 respectively, then each thread may appear larger load imbalance.

The **schedule clause specifies**
how iterations of these associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team. Each thread executes its assigned chunk(s) in the context of its implicit task.

Static schedule

- **schedule Clause :** schedule (STATIC [, size]）
  - no *chunk_size is specified*
    - the iteration space is divided into chunks that are approximately equal in size
  - *chunk_size is specified*
    - iterations are divided into chunks of size *chunk_size*
  - example： thread_num is 4

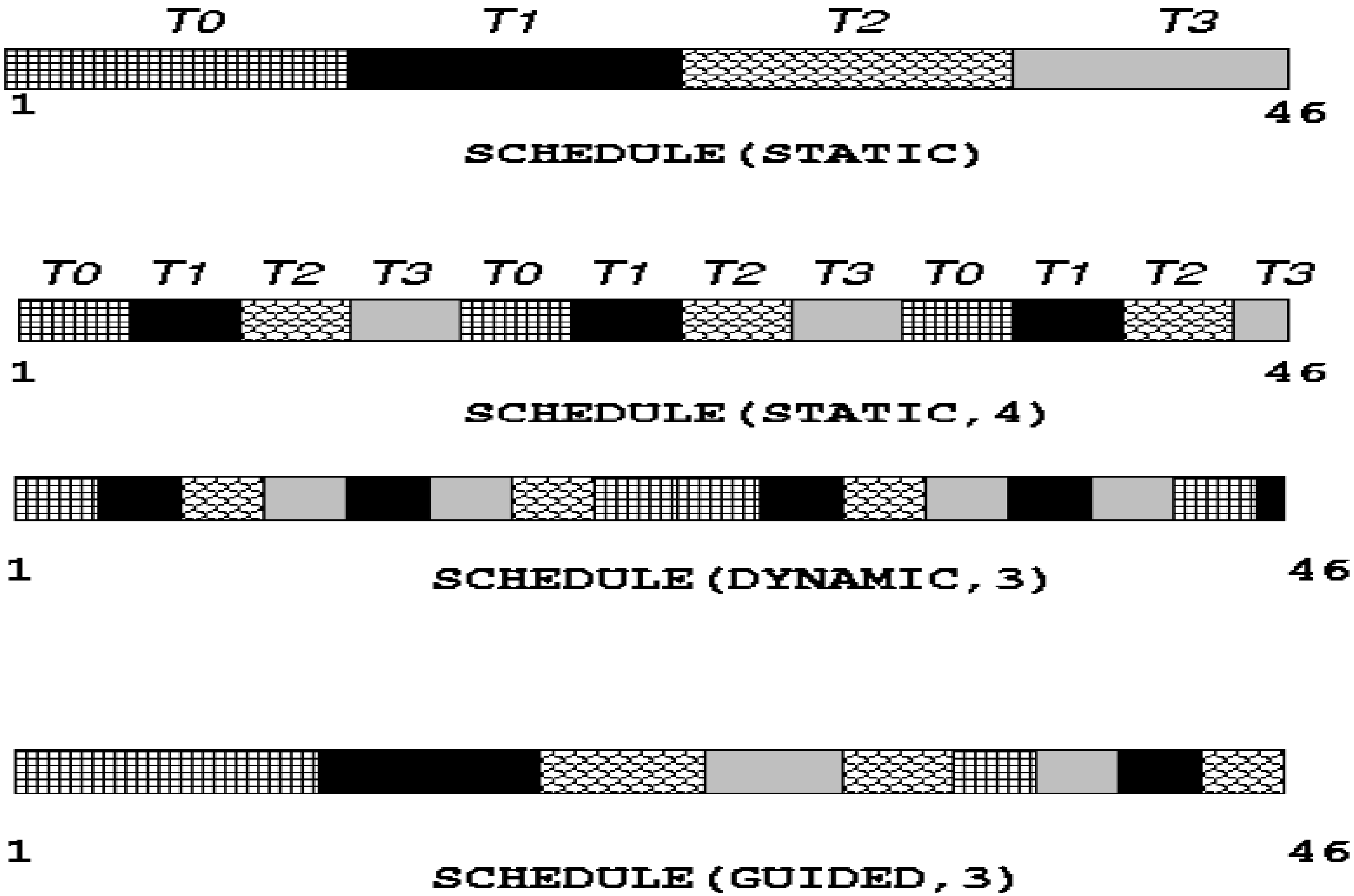


- **schedule (DYNAMIC[, size]）：**
  - the iterations are distributed to threads in the team in chunks.
  - Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.
  - no *chunk_size is specified, it defaults to 1.*
- **schedule (GUIDED[, chunksize]**）
  - Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned.

$$S_k = \left\lceil \frac{R_k}{2N} \right\rceil$$

  - Where N is the number of theads，Sk is the size of the kth chunk，Rk is the number of remaining iterations。

SCHEDULE example



parallel for vector sum

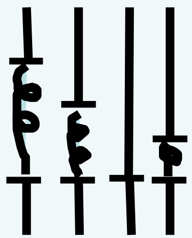
```
#include <omp.h> //ex2
#define   N       1000
#define CHUNKSIZE    100
int main () {
      int i, chunk;
      float a[N], b[N], c[N];
      /* Some initializations */
      for (i=0; i < N; i++)
                a[i] = b[i] = i * 1.0;
      chunk = CHUNKSIZE;
      #pragma omp parallel for \
                shared(a,b,c,chunk) private(i) \
                schedule(static,chunk)
                for (i=0; i <N; i++)    {
                                c[i] = a[i] + b[i];
      }
}
```

PARALLEL PROGRAMMING IN OPENMP:

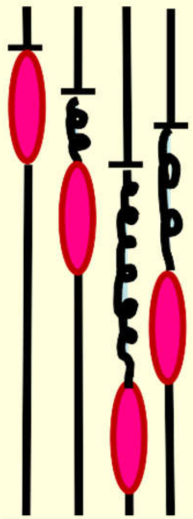
SYNCHRONIZATION CONSTRUCT

Synchronization

- Synchronization: bringing one or more threads to a well defined and known point in their execution.
- The two most common forms of synchronization are:



Barrier: each thread wait at the barrier until all threads arrive.



Mutual exclusion: Define a block of code that only one thread at a time can execute.

Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data
- High-Level Synchronization
  – critical
  – atomic
  – barrier
  – ordered
- Low-Level Synchronization
  – locks

Data race

- example：
- Serial progam

```
Int i, max_num=1;
for (i=0;i<n;i++)
   if (ar[i]>max_num)
      max_num=ar[i];
```

- Using openmp：

```
int i, max_num=1;
#pragma omp parallel for
for (i=0;i<n;i++)
   if (ar[i]>max_num)
      max_num=ar[i];
```

X

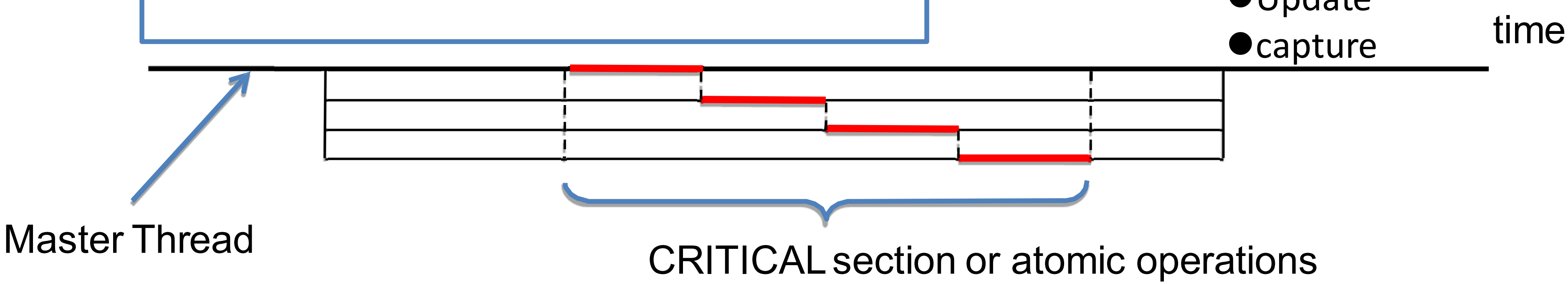
Synchronization: Critical/Atomic Directives

- When each thread must execute a section of code serially the region must be marked with critical/end critical directives
- Use the #pragma omp atomic directive if executing only one operation serially

```
#pragma omp parallel shared(sum,x,y)
 ...
#pragma omp critical
   {
      update(x);
      update(y);
      sum=sum+1;
   }
...
```

```
#pragma omp parallel shared(sum)
...
    {
#pragma omp atomic
       sum=sum+1;
       ...
    }
```

- Read
- Write
- Update
- capture



Synchronization: Atomic (basic form)

- **Atomic** provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
        double tmp, B;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
          X +=  tmp;
}
```

The statement inside the atomic must be one of the following forms:

- x binop= expr
- x++
- ++x
- x—
- --x

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

Additional forms of atomic were added in OpenMP 3.1.

Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```
#pragma omp parallel

{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
   #pragma omp barrier

        B[id] = big_calc2(id, A);
}
```

Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive

```
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
    for(i=0;i<N;i++){
           C[i]=big_calc3(i,A);
    }                              Implicit barrier
    #pragma omp for nowait
    for(i=0;i<N;i++){
            B[i]=big_calc2(C, i);
    }                              No implicit barrier due to nowait
    A[id] = big_calc4(id);
}                                  Implicit barrier
```

Synchronization: Barrier

master construct

```
#pragma omp parallel
{
    do_many_things();
#pragma omp master
    {
        exchange_boundries();
    }
#pragma omp barrier
    do_many_other_things();
}
```

No implicit barrier

single construct

```
#pragma omp parallel
{
    do_many_things();
#pragma omp single
    {
        exchange_boundries();
    }
   do_many_other_things();
}
```

implicit barrier

NOWAIT

- When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.
- By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
#pragma omp parallel
     {
#pragma omp for nowait
      {
         for (i=0; i<n; i++)
           {work(i);}
      }
#pragma omp for schedule(dynamic,k)
      {
         for (i=0; i<m; i++)
           {x[i]=y[i]+z[i];}
      }
     }
```

Synchronization: Ordered

- The ordered region executes in the sequential order

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:count)
for (i=0;i<N;i++){
   tmp = foo(i);
   #pragma omp ordered
   count += consume(tmp);
}
```

PARALLEL PROGRAMMING IN OPENMP:

TASKS CONSTRUCT

list traversal

- When we first created OpenMP, we focused on common use cases in HPC … Fortran arrays processed over "regular" loops.
- Recursion and "pointer chasing" were so far removed from our Fortan focus that we didn't even consider more general structures.
- Hence, even a simple list traversal is exceedingly difficult with the original versions of OpenMP.

```
p=head;
while (p) {
   process(p);
   p = p->next;
}
```

Linked lists without tasks

```
while (p !=
NULL) {
p = p->next;
 count++;
}
p = head;
 for(i=0; i<count;
 i++) {
  parr[i] = p;
 p = p->next;
 }
#pragma omp parallel
{
   #pragma omp for schedule(static,1)
   for(i=0; i<count; i++)
     processwork(parr[i]);
}
```

Count number of items in the linked list

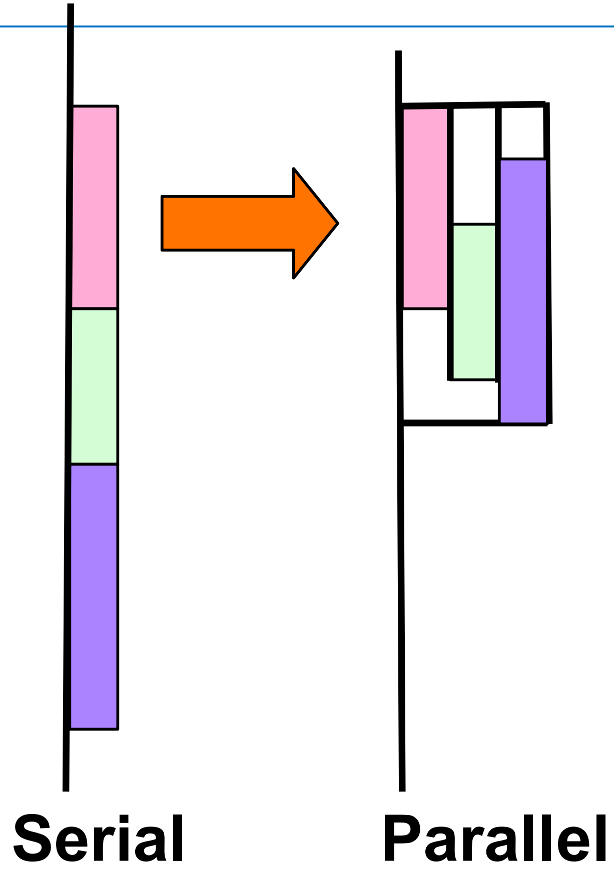
Copy pointer to each node into an array

Process nodes in parallel with a for loop

	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

What are tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.
  - The thread that encounters the task construct may execute the task immediately.
  - The threads may defer execution until later
- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Tasks can be nested: i.e. a task may itself generate tasks.



When are tasks guaranteed to complete

- Tasks are guaranteed to be complete at thread barriers:

  #pragma omp barrier

- or task barriers

  #pragma omp taskwait

```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

The task construct (OpenMP 4.5)

#pragma omp task *[clause[[,]clause]...]*
structured-block

Generates an explicit task

where *clause* is one of the following:

if(*[* **task :***]scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

final(scalar-expression)
mergeable

depend(*dependence-type* **:** *list***)**
priority(*priority-value***)**

The evolution of the task construct

OpenMP 3.0 (May'08)

OpenMP 3.1 (Jul'11)

OpenMP 4.0 (Jul'13)

OpenMP 4.5 (Nov'15)

The task construct (OpenMP 4.5)

```
#pragma omp task [clause[[,]clause]...]
            structured-block
```

Generates an explicit task

where *clause* is one of the following:

if([**task :**]*scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

Consider the data environment associated with a task

final(scalar-expression)

mergeable

depend(*dependence-type* **:** *list***)**

priority(*priority-value***)**

The evolution of the task construct

OpenMP 3.0

OpenMP 3.1

OpenMP 4.0

OpenMP 4.5

Data scoping with tasks

- Variables can be shared, private or firstprivate with respect to task
- These concepts are a little bit different compared with threads:
  - If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  - If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Data scoping defaults

- The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered are firstprivate by default
- Variables that are shared in all constructs starting from the inner most enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
#pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared
B is firstprivate
C is private

Exercise: traversing linked lists

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program selecting from the following list of constructs:

```
#pragma omp parallel
#pragma omp single
#pragma omp task
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
private(), firstprivate()
```

- Hint: Just worry about the contents of main(). You don't need to make any changes to the "list functions"

Parallel linked list traversal

Only one thread packages tasks

```
#pragma omp parallel
{
  #pragma omp single
   {
    p = listhead ;
    while (p) {
        #pragma omp task firstprivate(p)
              {
                process (p);
              }
        p=next (p) ;
     }
   }
}
```

makes a copy of **p** when the task is packaged

Example

```
#pragma omp parallel
{
  #pragma omp single
   {
      #pragma omp task
         fred();
      #pragma omp task
     daisy();
     #pragma omp taskwait
      #pragma omp task
         billy();
   }
}
```

`fred()` and `daisy()` must complete before `billy()` starts, but this does not include tasks created inside **fred()** and **daisy()**

`All tasks` including those created inside **fred()** and **daisy()** must complete before exiting this barrier

Example

```
#pragma omp parallel
{
  #pragma omp single nowait
   {
       #pragma omp task
          fred();
       #pragma omp task
          daisy();
     #pragma omp taskwait
       #pragma omp task
          billy();
   }
}
```

The barrier at the end of the single is expensive and not needed since you get the barrier at the end of the parallel region. So use nowait to turn it off.

`All tasks` including those created inside **fred()** and **daisy()** must complete before exiting this barrier

Example: Fibonacci numbers

```
int fib (int n)
{
  int x,y;
  if (n < 2) return n;

  x = fib(n-1);
  y = fib (n-2);
  return (x+y);
}

Int main()
{
  int NW = 5000;
  fib(NW);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient $O(n^2)$ recursive implementation!

Data Scoping with tasks: Fibonacci example.

This is an instance of the divide and conquer design pattern

```
int fib ( int n )
{

int x,y;
   if ( n < 2 ) return n;
#pragma omp task
   x = fib(n-1);
#pragma omp task
  y = fib(n-2);
#pragma omp taskwait
   return x+y
}
```

n is private in both tasks

x is a private variable
y is a private variable

What's wrong here?

A task's private variables are undefined outside the task

Data Scoping with tasks: Fibonacci example.

```
int fib ( int n )
{

int x,y;
   if ( n < 2 ) return n;
#pragma omp task shared (x)
   x = fib(n-1);
#pragma omp task shared(y)
   y = fib(n-2);
#pragma omp taskwait
   return x+y;
}
```

n is private in both tasks

x & y are shared
Good solution
we need both values to compute the sum

Parallel Fibonacci

```
int fib (int n)
{   int x,y;
  if (n < 2) return n;

#pragma omp task shared(x)
 x = fib(n-1);
#pragma omp task shared(y)
y = fib (n-2);
#pragma omp taskwait
return (x+y);
}

Int main()
{  int NW = 5000;
  #pragma omp parallel
  {
     #pragma omp master fib(NW);
  }
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- **x,y** are local, and so by default they are private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

Data Scoping with tasks: List Traversal example

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

What's wrong here?

Possible data race !
Shared variable e
updated by multiple tasks

Data Scoping with tasks: List Traversal example

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task firstprivate(e)
      process(e);
}
```

Good solution – e is firstprivate

Task Construct – Explicit Tasks

1. Create a team of threads.

```
#pragma omp parallel
{
  #pragma omp single
  {
    node * p = head;
    while (p) {
    #pragma omp task firstprivate(p)
      process(p);
    p = p->next;
    }
  }
}
```

2. One thread executes the **single** construct

… other threads wait at the implied barrier at the end of the single construct

3. The “single” thread creates a task with its own value for the pointer p

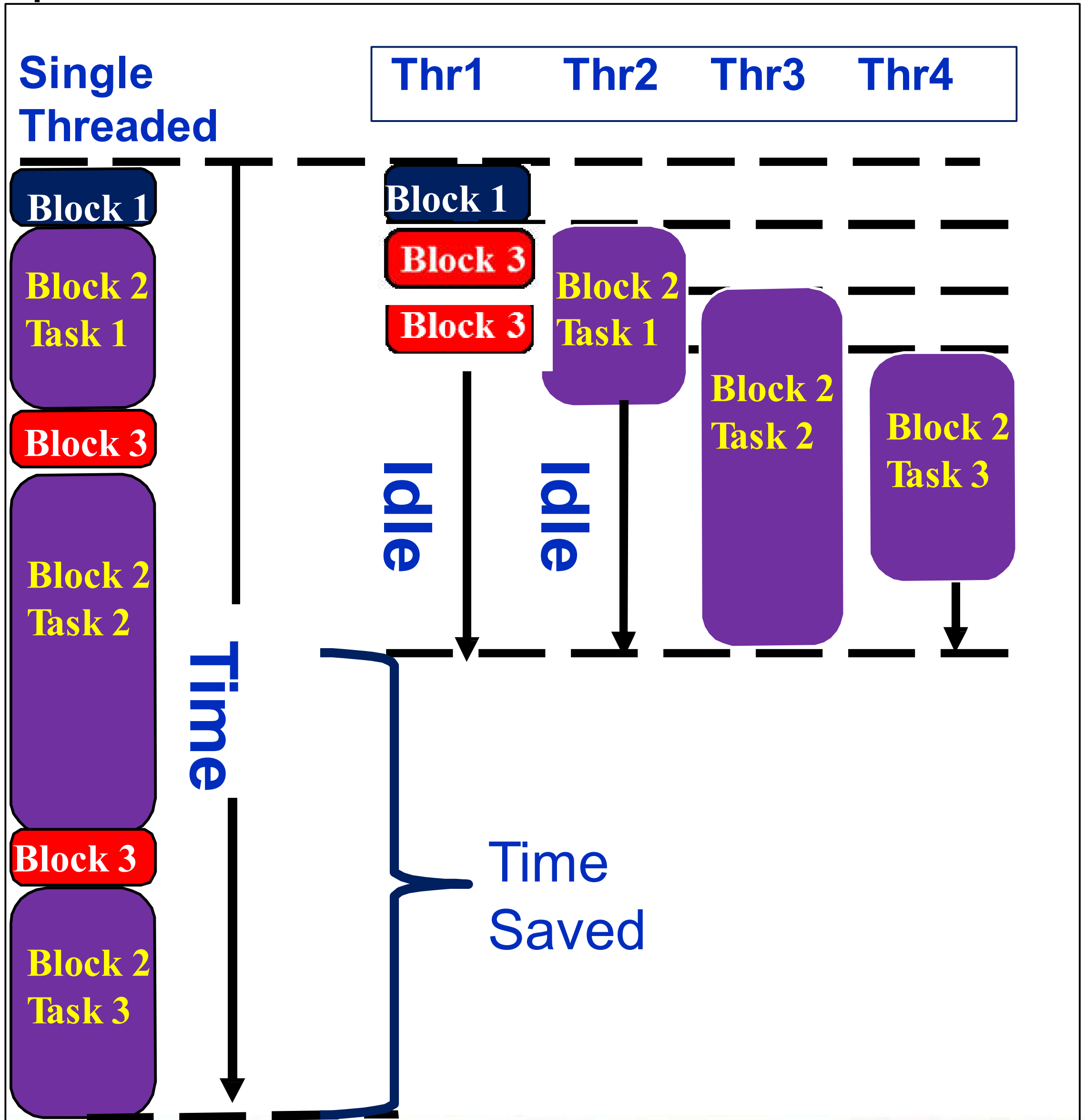
4. Threads waiting at the barrier execute tasks.

Execution moves beyond the barrier once all the tasks are complete

Execution of tasks

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
  #pragma omp single
  {   //block 1
    node * p = head;
    while (p) { // block 2
    #pragma omp task
      process(p);
    p = p->next;   //block 3
    }
  }
}
```

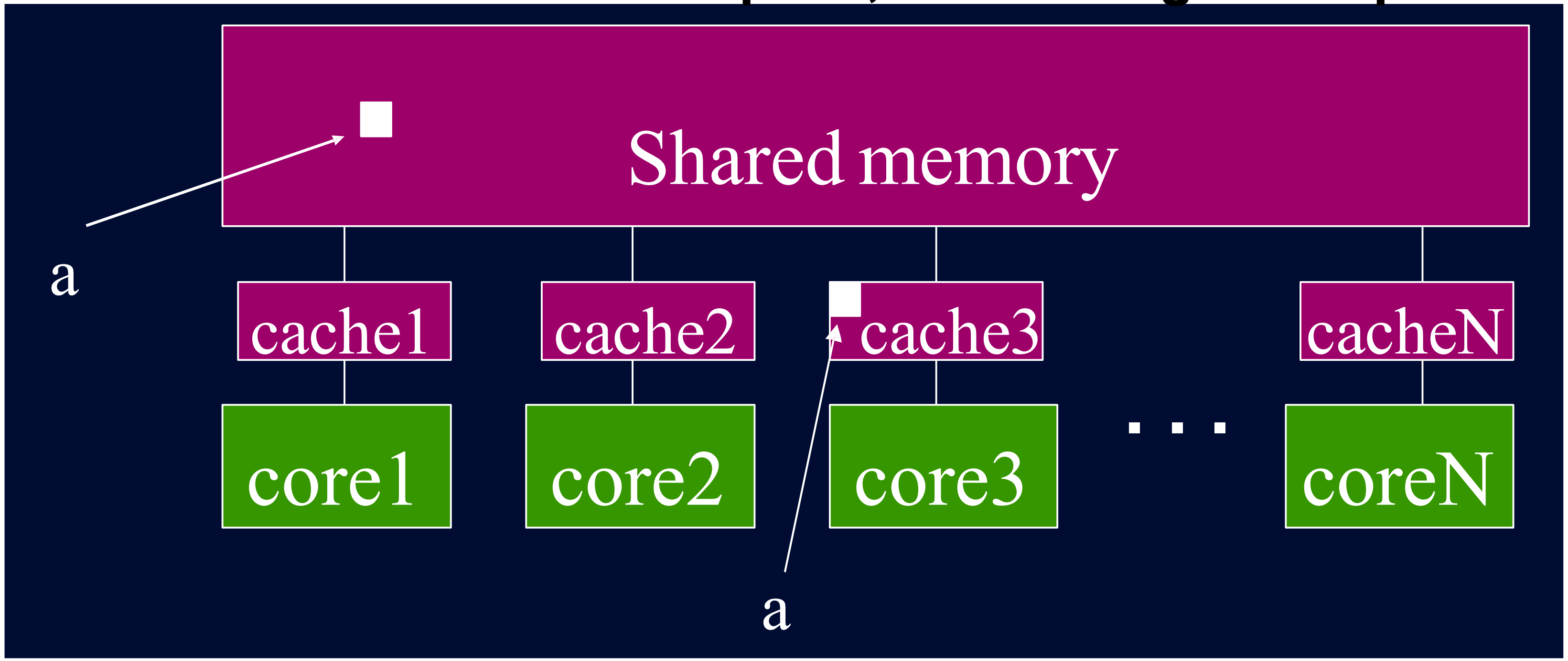


PARALLEL PROGRAMMING IN OPENMP:

MEMORY MODEL

OpenMP memory model

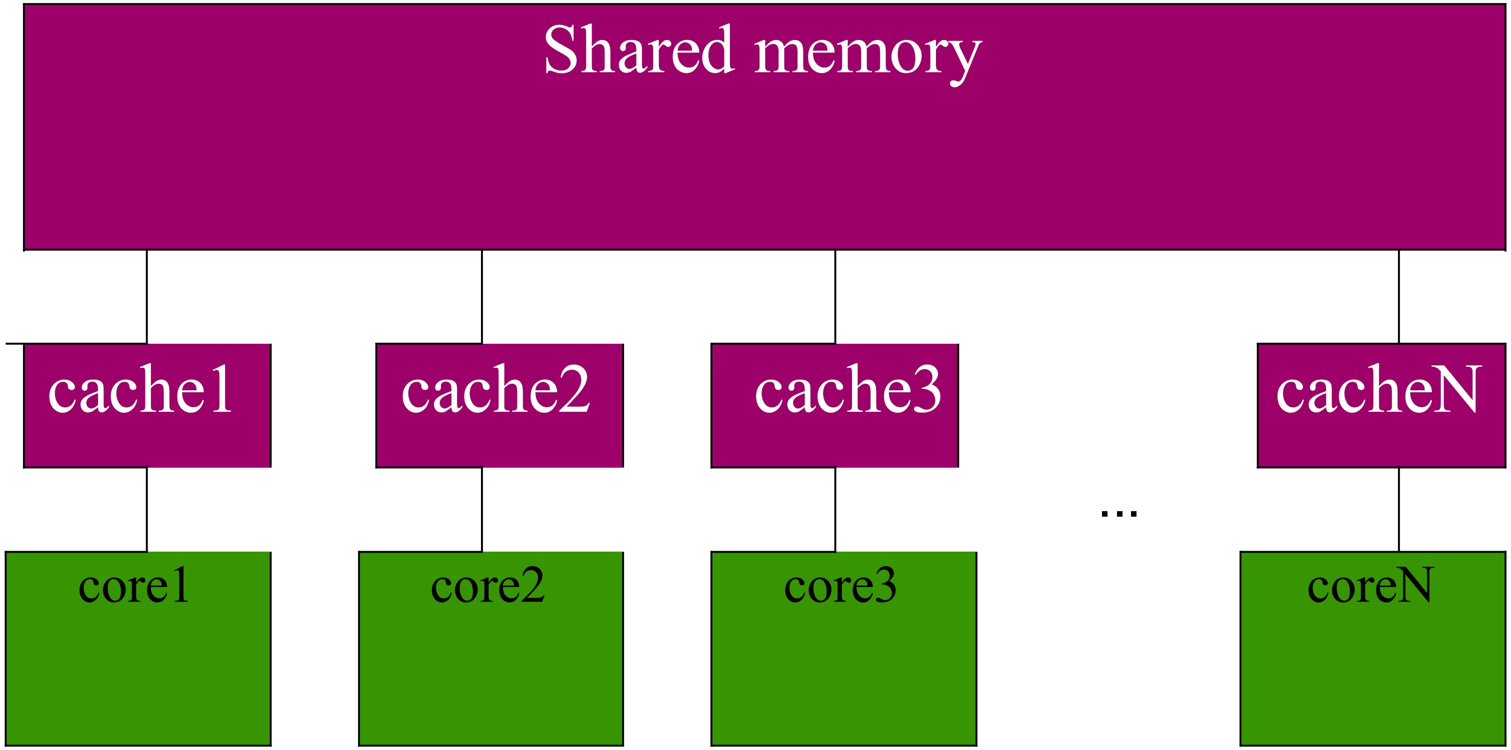
- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:



- memory model is defined in terms of:
  - Coherence: Behavior of the memory system when a single address is accessed by multiple threads.
  - Consistency: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.

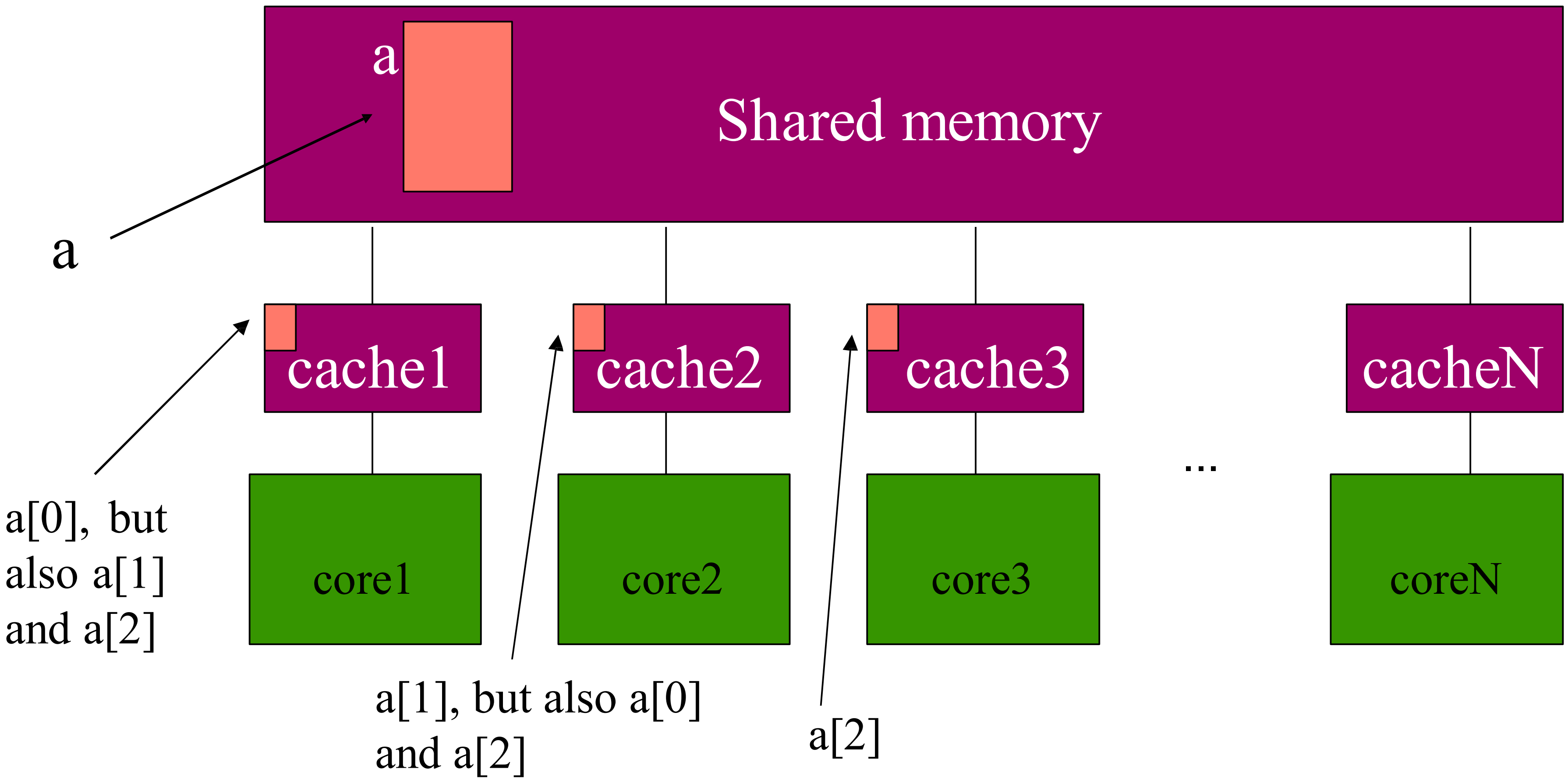
Cache Coherence and False Sharing

- Blocks of data are fetched into cache lines
- What happens if multiple threads access different data, but on same cache line, at same time?

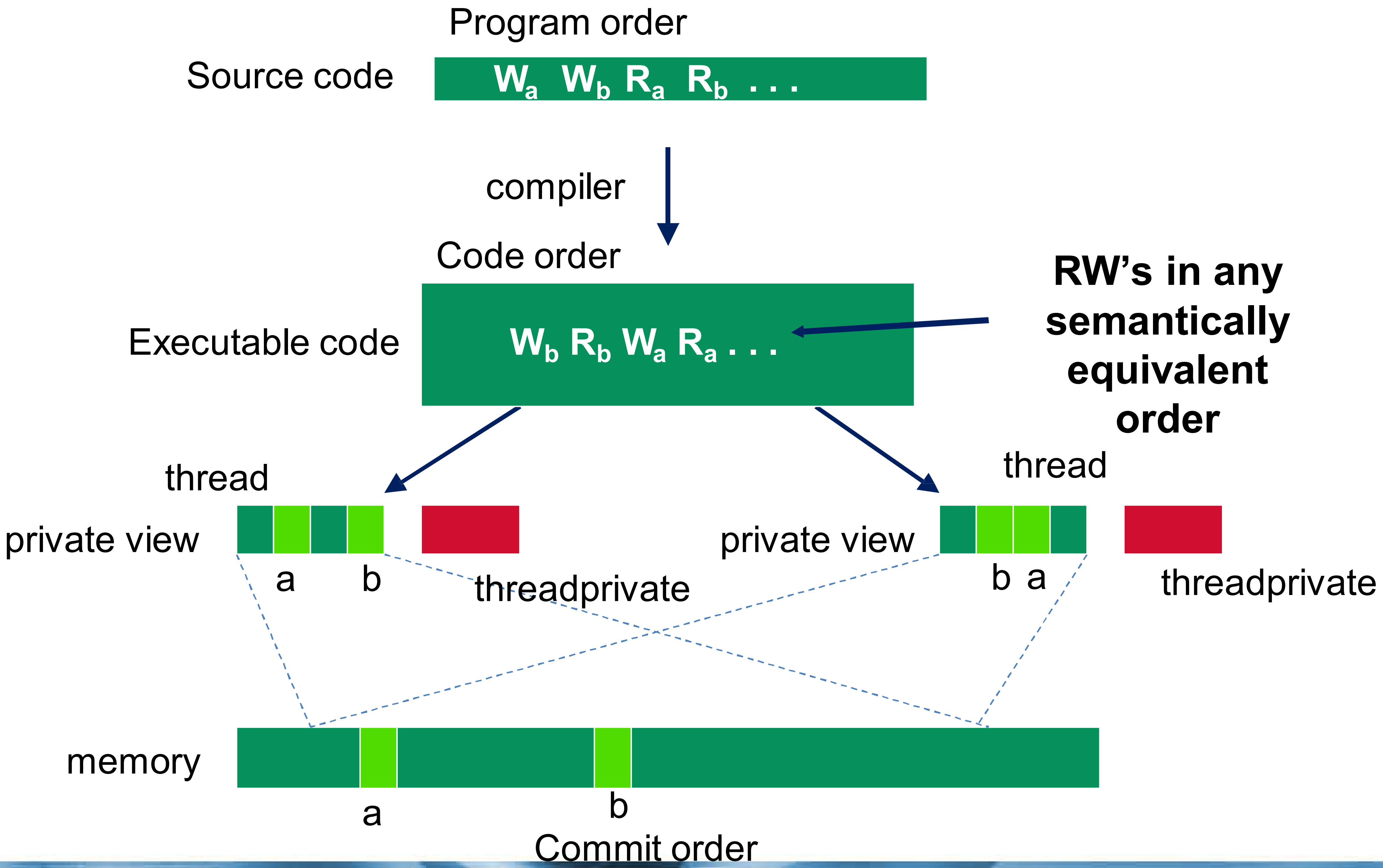


Updates to Shared Data

- Blocks of data are transferred to cache lines
- When an element of cache line is updated, the entire line is invalidated: local copies are reloaded from main memory



OpenMP Memory Model: Basic Terms



Consistency: Memory Access Re-ordering

- **Re-ordering:**
  - **Compiler re-orders program order to the code order**
  - **Machine re-orders code order to the memory commit order**
- **At a given point in time, the "private view" seen by a thread may be different from the view in shared memory.**
- **Consistency Models define constraints on the orders of Reads (R), Writes (W) and Synchronizations (S)**
  - **… i.e. how do the values "seen" by a thread change as you change how ops follow (→) other ops.**
  - **Possibilities include:**
    - **R→R, W→W, R→W, R→S, S→S, W→S**

Consistency

- Sequential Consistency:
  - In a multi-processor, ops (R, W, S) are sequentially consistent if:
    - **They remain in program order for each**
  - processor.
    - **They are seen to be in the same overall order by each of the other processors.**
- Program order = code order = commit order
- Relaxed consistency:
  - Remove some of the ordering constraints for memory ops (R, W, S).

OpenMP and Relaxed Consistency

- **OpenMP defines consistency as a variant of <u>weak consistency</u>:**
  - **Can not reorder S ops with R or W ops on the same thread**
    - **Weak consistency guarantees**

      $S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
- **The Synchronization operation relevant discussion is flush.**

Flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the "flush set".
- The flush set is:
  - "all thread visible variables" for a flush construct without an argument list.
  - a list of variables when the "flush(list)" construct is used.
- The action of Flush is to guarantee that:
  - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  - All R,W ops that overlap the flush set and occur after the flush don't execute until after the flush.
  - Flushes with overlapping flush sets can not be reordered.

Memory ops: R = Read, W = write, S = synchronization

Synchronization: flush example

- **Flush forces data to be updated in memory so other threads see the most recent value**

```
double A;

A = compute();

flush(A);   // flush to memory to make sure other
            //  threads can pick up the right value
```

Note: OpenMP's flush is analogous to a fence in other shared memory API's.

Flush and synchronization

- **A flush operation is implied by OpenMP synchronizations, e.g.**
  - **at entry/exit of parallel regions**
  - **at implicit and explicit barriers**
  - **at entry/exit of critical regions**
  - **whenever a lock is set or unset**

….

(but not at entry to worksharing regions or entry/exit of master regions)

What is the Big Deal with Flush?

- Compilers routinely reorder instructions implementing a program
  - This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.
- Compiler generally cannot move instructions:
  - past a barrier
  - past a flush on all variables
- But it can move them past a flush with a list of variables so long as those variables are not accessed
- Keeping track of consistency when flushes are used can be confusing … especially if “flush(list)” is used.

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread’s values are made consistent with main memory.

Example

```
int main()
{
  double *A, sum, runtime;
  int flag = 0;

  A = (double *)malloc(N*sizeof(double));

  runtime = omp_get_wtime();

  fill_rand(N, A);          // Producer: fill an array of data

  sum = Sum_array(N, A);  // Consumer: sum the array

  runtime = omp_get_wtime() - runtime;

  printf(" In %f seconds, The sum is %f \n",runtime,sum);
}
```