Shared memory programming

Jorge Barbosa, FEUP

Parallel Programming
in C with MPI and OpenMP
Michael J. Quinn

Using OpenMP
Barbara Chapman, Gabriele Jost and Ruud van der Pas

Contents

- OpenMP
- Shared-memory model
- Parallel `for` loops
- Declaring private variables
- Critical sections
- Reductions
- Performance improvements
- More general data parallelism
- Functional parallelism

OpenMP

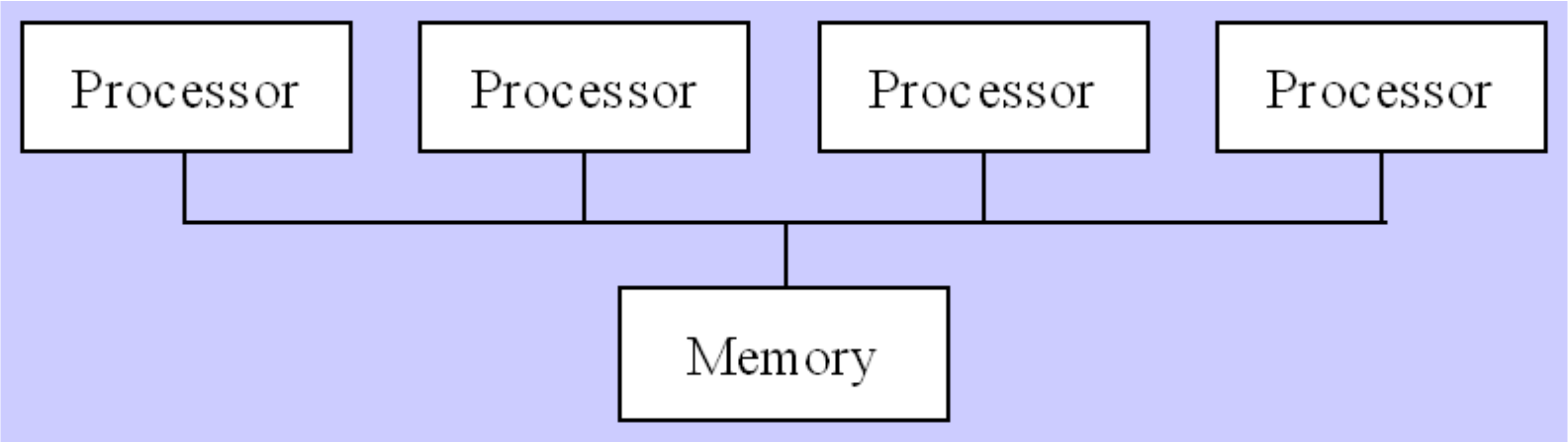
- OpenMP: An application programming interface (API) for parallel programming on multicores
  - Compiler directives
  - Library of support functions

- OpenMP works in conjunction with Fortran, C, or C++

What’s OpenMP Good For?

- C + OpenMP sufficient to program multicores
- Easy to convert sequential data parallel programs

Shared memory model

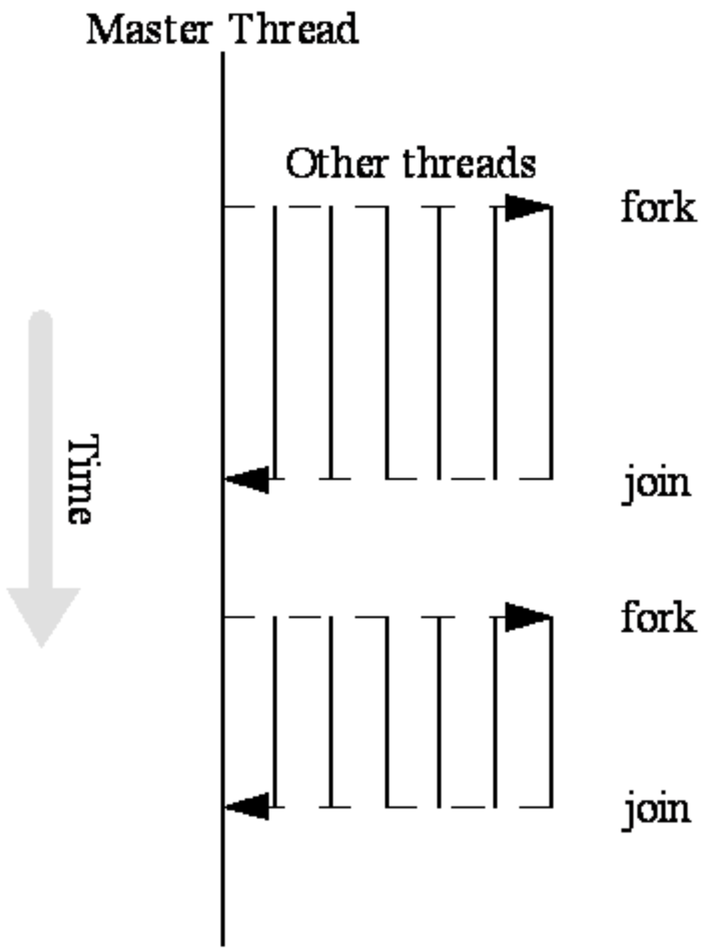


- Global address space
- There is no reference to communications

Fork/Join Parallelism

- Initially only master thread is active
  - Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At end of parallel code created threads die or are suspended

Paralelismo Fork/Join



Shared-memory Model vs. Message-passing Model (#1)

- Shared-memory model
  - Number active threads is 1 at start and finish of program, changes dynamically during execution
- Message-passing model
  - All processes active throughout execution of program

Incremental Parallelization

- Sequential programming is a special case of a shared-memory parallel program
- Parallel shared-memory programs may only have a single parallel loop
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time

Shared-memory Model vs. Message-passing Model (#2)

- Shared-memory model
  - Execute and profile sequential program
  - Incrementally make it parallel
  - Stop when further effort not warranted

- Message-passing model
  - Sequential-to-parallel transformation requires major effort
  - Transformation done in one giant step rather than many tiny steps

Parallel for Loops

- C programs often express data-parallel operations as **for** loops

```
for (i = first; i < size; i += prime)
        marked[i] = 1;
```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel

- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads

Pragmas

- Pragma: a compiler directive in C or C++
- Stands for "pragmatic information"
- A way for the programmer to communicate with the compiler
- Compiler free to ignore pragmas
- Syntax:

  `#pragma omp` <*rest of pragma*>

Parallel for Pragma

- Format:

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

Canonical Shape of for Loop Control Clause

$$\text{for}\left(\text{index} = start; \text{index} \geq \begin{Bmatrix} < \\ <= \\ >= \\ > \end{Bmatrix} end; \begin{Bmatrix} \text{index}++ \\ ++\text{index} \\ \text{index}-- \\ --\text{index} \\ \text{index} += inc \\ \text{index} -= inc \\ \text{index} = \text{index} + inc \\ \text{index} = inc + \text{index} \\ \text{index} = \text{index} - inc \end{Bmatrix}\right)$$

Execution Context

- Every thread has its own execution context
- Execution context: address space containing all of the variables a thread may access
- Contents of execution context:
  - static variables
  - dynamically allocated data structures in the heap
  - variables on the run-time stack
  - additional run-time stack for functions invoked by the thread

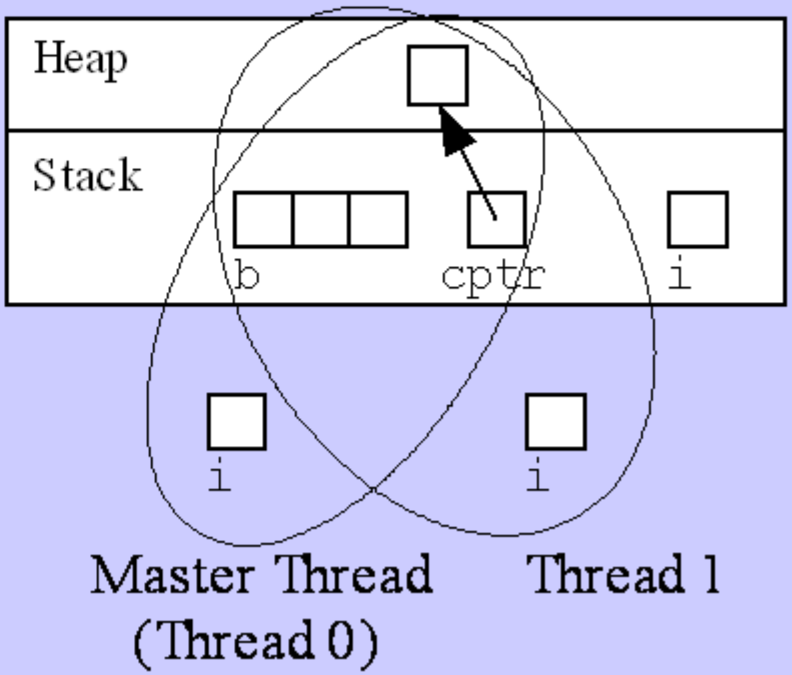
Shared and Private Variables

- Shared variable: has same address in execution context of every thread
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread

Shared and Private Variables

```
int main (int argc, char *argv[])
{
   int b[3];
   char *cptr;
   int i;

   cptr = malloc(1);
#pragma omp parallel for
   for (i = 0; i < 3; i++)
      b[i] = i;
```



Function omp_get_num_procs

- Returns number of physical processors available for use by the parallel program

```
int omp_get_num_procs (void)
```

Function omp_set_num_threads

- Uses the parameter value to set the number of threads to be active in parallel sections of code
- May be called at multiple points in a program

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

Parallel Blocks

```
#pragma omp parallel [num_threads(n)]
{
   ...
}
```

This command overwrites: **omp_set_num_threads**

Declaring Private Variables

```
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
   for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j],a[i][k]+tmp);
```

- Either loop could be executed in parallel
- We prefer to make outer loop parallel, to reduce number of forks/joins
- We then must give each thread its own private copy of variable **j**

private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

```
private ( <variable list> )
```

Example Use of private Clause

```
#pragma omp parallel for private(j)
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
    for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j],a[i][k]+tmp);
```

firstprivate Clause

- Private variables are undefined on thread entry
- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
- Variables are initialized once per thread, not once per loop iteration
- If a thread modifies a variable's value in an iteration, subsequent iterations will get the modified value

Exemple - firstprivate

```
int main()
{     int TID;
      #pragma omp parallel private(TID)
      {
          TID = omp_get_thread_num();

          printf("Thread %d executes the outer parallel region\n",TID);

         #pragma omp parallel num_threads(3) firstprivate(TID)
         {        printf("TID %d: Thread %d executes inner parallel region\n",
                         TID,omp_get_thread_num());
         }  /*-- End of inner parallel region --*/
      }   /*-- End of outer parallel region --*/

      return(0);
}
```

nested-parallel-mod.c

Lab work

lastprivate Clause

- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially

- **`lastprivate`** clause: used to copy back to the master thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration

lastprivate Clause

```
#pragma omp parallel for lastprivate(a)
  for (i=0; i<5; i++)
  {   a=i+1;
      printf("Thread %d has value a=%d for i=%d\n",
      omp_get_thread_num(), a, i);
  }
Printf("value after loop a=%d",a)
Output:
Thread 0 has value a=1 for i=0
Thread 1 has value a=2 for i=1
Thread 4 has value a=5 for i=4
Thread 2 has value a=3 for i=2
Thread 3 has value a=4 for i=3
value after loop a=5
```

Critical Sections

- Consider the program to compute π using the rectangle rule:

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition

- If we simply parallelize the loop...

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition (cont.)

- … we set up a race condition in which one process may "race ahead" of another and not see its change to shared variable **`area`**

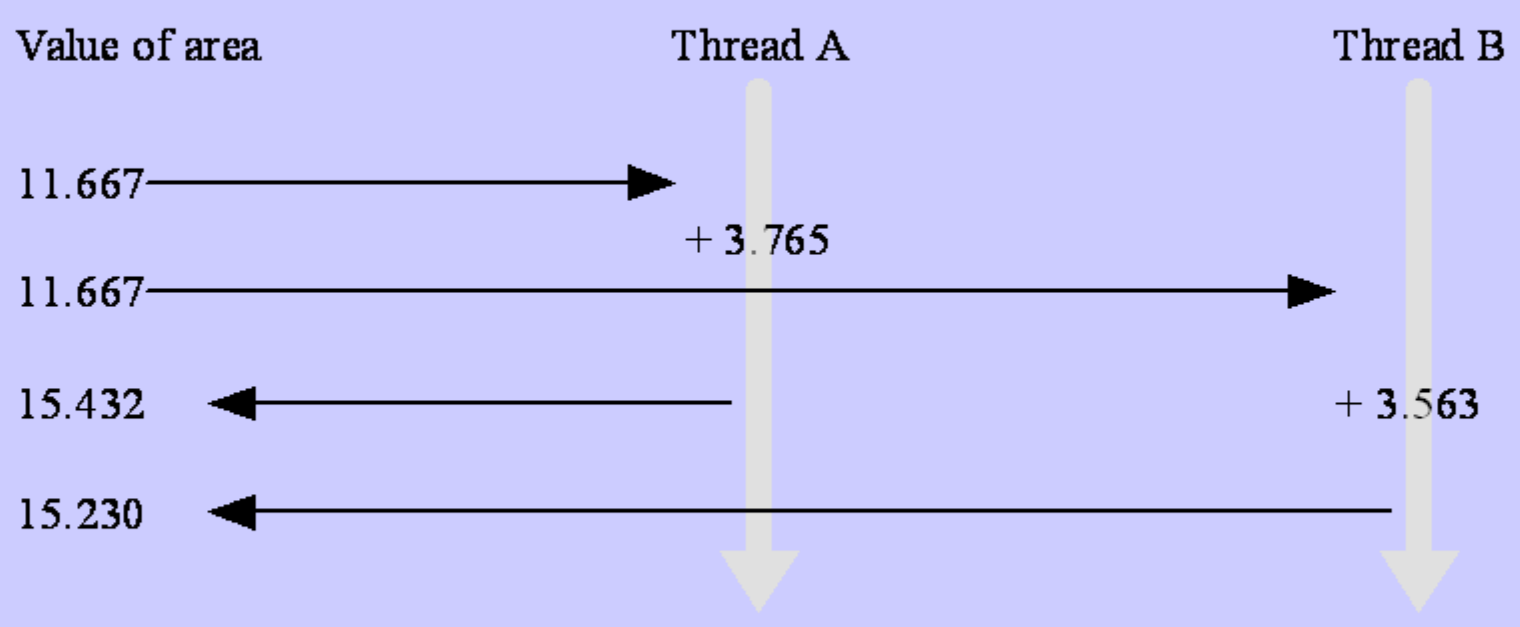
`area` 15.230 **Answer should be 18.995**

Thread A 15.432

Thread B 15.230

```
area += 4.0/(1.0 + x*x)
```

Race Condition Time Line



critical Pragma

- Critical section: a portion of code that only a thread at a time may execute
- We denote a critical section by putting the pragma

```
#pragma omp critical
```

in front of a block of C code

Correct, But Inefficient, Code

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
#pragma omp critical
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Source of Inefficiency

- Update to **area** inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop
- By Amdahl's Law we know speedup will be severely constrained

Reductions

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to **parallel for** pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop

reduction Clause

- The reduction clause has this syntax:
  `reduction (<op> :<variable>)`
- Operators
  - \+ Sum
  - * Product
  - & Bitwise and
  - | Bitwise or
  - ^ Bitwise exclusive or
  - && Logical and
  - || Logical or

π-finding Code with Reduction Clause

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
        private(x) reduction(+:area)
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

nowait Clause

- Compiler puts a barrier synchronization at end of every parallel for statement

```
#pragma omp parallel
{

   #pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];

   other();
}
```

parallel Pragma

- The `parallel` pragma precedes a block of code that should be executed by *all* of the threads
- Note: execution is replicated among all threads

Use of `parallel` Pragma

```
#pragma omp parallel private(task_ptr)
{
   task_ptr = get_next_task (&job_ptr);
   while (task_ptr != NULL) {
      complete_task (task_ptr);
      task_ptr = get_next_task (&job_ptr);
   }
}
```

Blocks executed by a single thread

- #pragma omp master
- #pragma omp single
- #pragma omp barrier

Blocks executed by a single thread

```
int main( )
{   int a[5], i;

    #pragma omp parallel
    {   // Perform some computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

        // Print intermediate results.
        #pragma omp master   // single
            for (i = 0; i < 5; i++)
                printf_s("a[%d] = %d\n", i, a[i]);

        // Wait.
        #pragma omp barrier

        // Continue with the computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

Performance Improvement #1

- Too many fork/joins can lower performance
  - Inverting loops may help performance if
    - Parallelism is in inner loop
    - After inversion, the outer loop can be made parallel
  - Or, by defining outside the parallel region

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        #pragma omp parallel for
        for (k=0; k<n; k++)
            { .........}
```

```
#pragma omp parallel private(i,j)
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        #pragma omp for
        for (k=0; k<n; k++)
            { .........}
```

Performance Improvement #1

- Maximize parallel regions
  - Reduces the number of fork/joins

```
#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop 1 --*/
}

#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop 2 --*/
}
       .........

#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop N --*/
}
```

```
#pragma omp parallel
{
    #pragma omp for  /*-- Work-sharing loop 1 --*/
    { ...... }

    #pragma omp for  /*-- Work-sharing loop 2 --*/
    { ...... }

       .........

    #pragma omp for  /*-- Work-sharing loop N --*/
    { ...... }
}
```

Figures 5.23 and 5.24 from “Using OpenMP”

Performance Improvement #2
Conditional Parallelism

- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
- The **`if`** clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,

```
#pragma omp parallel for if(n > 5000)
```

Performance Improvement #3 Optimize Barrier Use

- #pragma omp for loop – has an implicit barrier

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

        #pragma omp for nowait
        for (i=0; i<n; i++)
            c[i] += d[i];

        #pragma omp barrier

        #pragma omp for nowait reduction(+:sum)
        for (i=0; i<n; i++)
            sum += a[i] + c[i];
} /*-- End of parallel region --*/
```

nowait do not
Influences here

Performance Improvement #4
Load Balance

- We can use **`schedule`** clause to specify how iterations of a loop should be allocated to threads
- Static schedule: all iterations allocated to threads before any iterations executed
- Dynamic schedule: only some iterations allocated to threads at beginning of loop's execution. Remaining iterations allocated to threads that complete their assigned iterations.

Static vs. Dynamic Scheduling

- Static scheduling
  - Low overhead
  - May exhibit high workload imbalance
- Dynamic scheduling
  - Higher overhead
  - Can reduce workload imbalance

Chunks

- A chunk is a contiguous range of iterations
- Increasing chunk size reduces overhead and may increase cache hit rate
- Decreasing chunk size allows finer balancing of workloads

schedule Clause

- Syntax of schedule clause

```
schedule (<type>[,<chunk> ])
```

- Schedule type required, chunk size optional
- Allowable schedule types
  - static: static allocation
  - dynamic: dynamic allocation
  - guided: guided self-scheduling
  - runtime: type chosen at run-time based on value of environment variable OMP_SCHEDULE

Scheduling Options

- schedule(static): block allocation of about n/t contiguous iterations to each thread
- schedule(static,C): interleaved allocation of chunks of size C to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,C): dynamic allocation of C iterations at a time to threads

Scheduling Options (cont.)

- schedule(guided, C): dynamic allocation of chunks to tasks using guided self-scheduling heuristic. Initial chunks are bigger, later chunks are smaller, minimum chunk size is C.

- schedule(guided): guided self-scheduling with minimum chunk size 1

- schedule(runtime): schedule chosen at run-time based on value of OMP_SCHEDULE; Unix example: **`setenv OMP_SCHEDULE "static,1"`**

Scheduling Options (cont.)

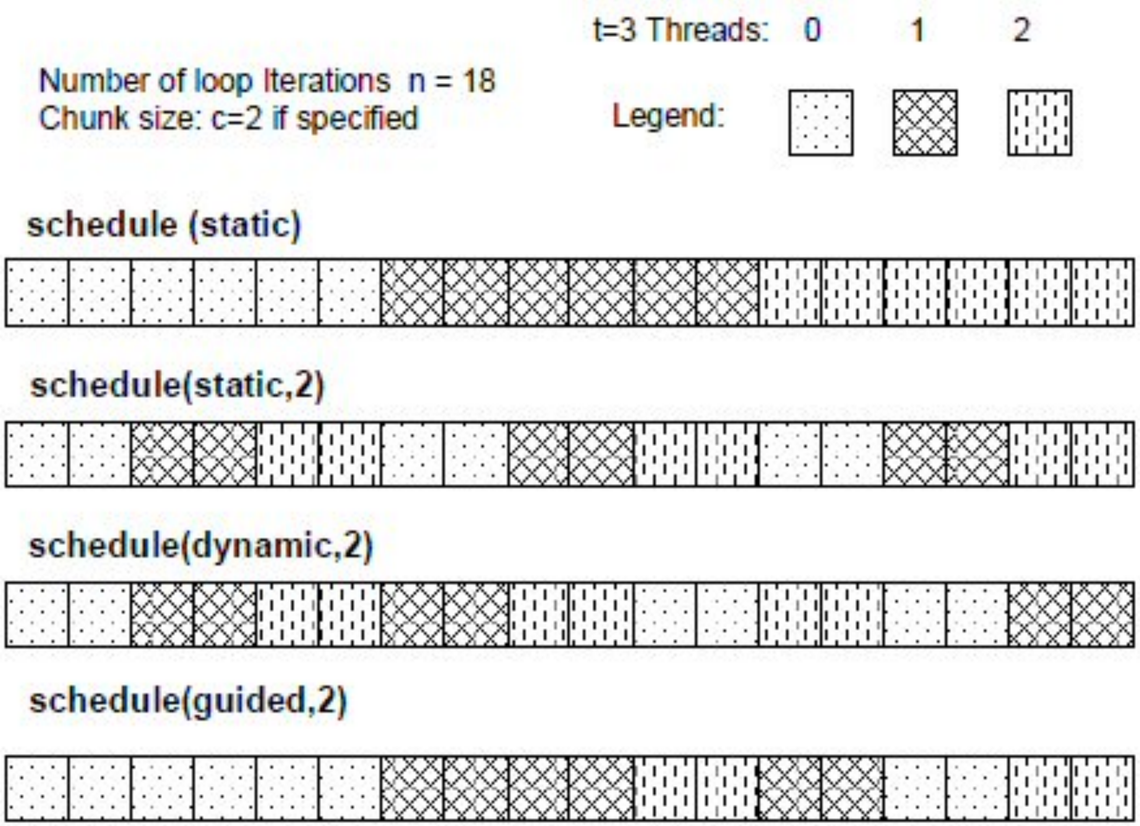


Figure 8.15: **Different kinds of loop schedules** – OpenUH has different strategies for handling static, dynamic, and guided schedules.

Scheduling example (1)

```
int main (int argc, char *argv[])
{   int n = 10, int i;

    #pragma omp parallel num_threads(4)
    {       #pragma omp master
            {           cout << endl << "my thread " << omp_get_thread_num();
                        cout << endl << "Num thread: " << omp_get_num_threads();
            }
            #pragma omp barrier                         // try if clause
            #pragma omp for schedule (dynamic, 4)  // try (static,4)   (dynamic,1)
            for (i = 0; i < n; i++) {
                #pragma omp critical                    // why is it used?
                cout << endl << "inside: " << omp_get_thread_num() << "   i= " << i ;
                Sleep(1000*omp_get_thread_num());
            }
    }
}
```

How would it be using pthreads?

loop.cpp

Scheduling example (2)

```
for (i=0; i<N; i++) {
        ReadFromFile(i,...);

        for (j=0; j<ProcessingNum; j++)
                ProcessData(); /* here is the work */

        WriteResultsToFile(i);
}
```

pipeline1.c

Scheduling example

```
#pragma omp parallel private(i)
{
    /* preload data to be used in first iteration of the i-loop */
    #pragma omp single
            {ReadFromFile(0,...);}

    for (i=0; i<N; i++) {
            /* preload data for next iteration of the i-loop */
            #pragma omp single nowait
            {ReadFromFile(i+1...);}

            #pragma omp for schedule(dynamic)
            for (j=0; j<ProcessingNum; j++)
                    ProcessChunkOfData(); /* here is the work */
            /* there is a barrier at the end of this loop */

            #pragma omp single nowait
                    {WriteResultsToFile(i);}

    } /* threads immediately move on to next iteration of i-loop */

} /* one parallel region encloses all the work */
/* Fig 5.28 from "Using OpenMP" */
```

Creates team of threads

A single thread reads data for 1st iteration

Sequential *for*

pipeline2.c

Lab work

Functions for SPMD-style Programming

Functional Parallelism

- The *parallel pragma* allows us to write SPMD-style programs
- In these programs we often need to know number of threads and thread ID number
- OpenMP provides functions to retrieve this information

Function omp_get_thread_num

- This function returns the thread identification number
- If there are *t* threads, the ID numbers range from 0 to *t*-1
- The master thread has ID number 0

```
int omp_get_thread_num (void)
```

Function omp_get_num_threads

- Function omp_get_num_threads returns the number of active threads
- If call this function from sequential portion of program, it will return 1

```
int omp_get_num_threads (void)
```

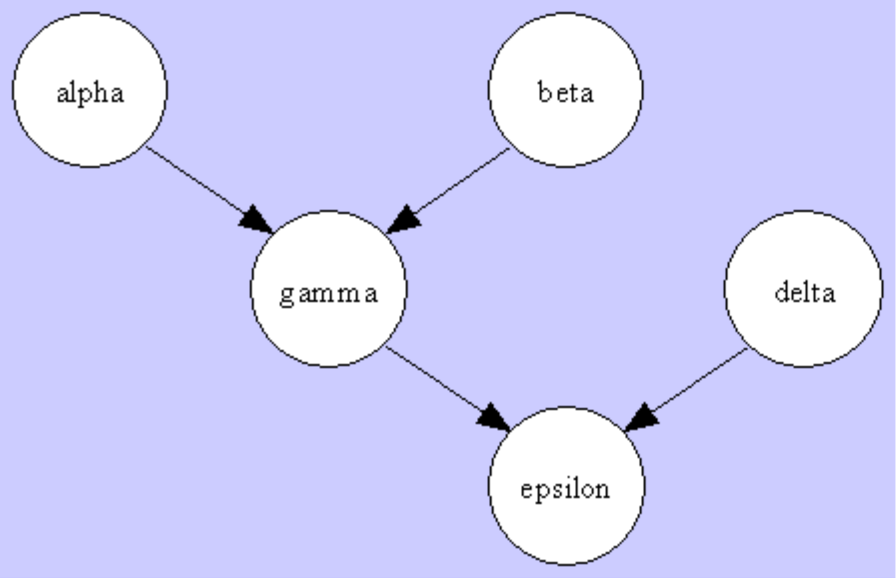
Functional Parallelism

- To this point all of our focus has been on exploiting data parallelism
- OpenMP allows us to assign different threads to different portions of code (functional parallelism)

Functional Parallelism Example

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```

May execute alpha, beta, and delta in parallel



parallel sections Pragma

- Precedes a block of *k* blocks of code that may be executed concurrently by *k* threads
- Syntax:

```
#pragma omp parallel sections
```

section Pragma

- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma
- Syntax:

```
#pragma omp section
```

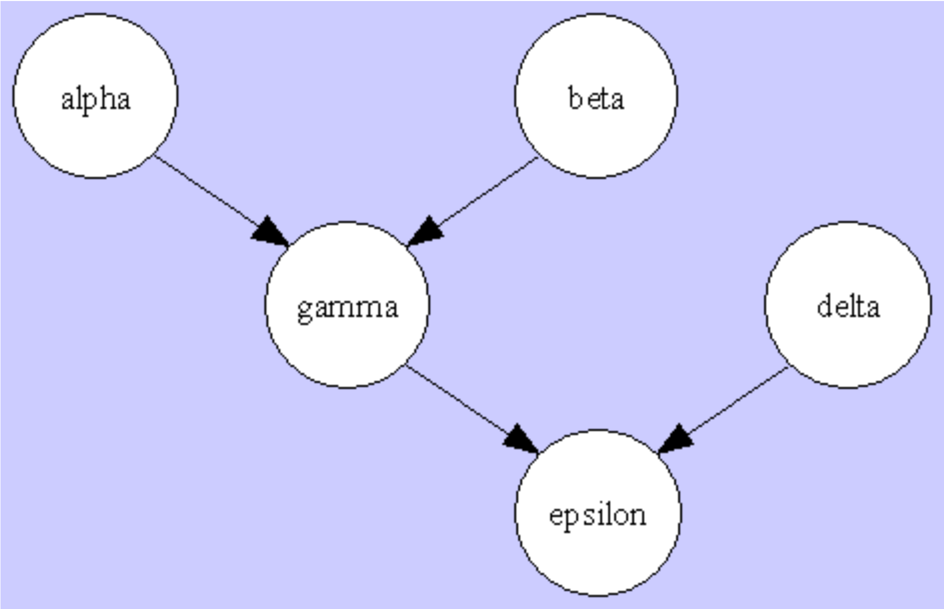
Example of `parallel sections`

```
#pragma omp parallel sections
    {
#pragma omp section  /* Optional */
        v = alpha();
#pragma omp section
        w = beta();
#pragma omp section
        y = delta();
    }
    x = gamma(v, w);
    printf ("%6.2f\n", epsilon(x,y));
```

sections Pragma

- Appears inside a parallel block of code
- Has same meaning as the **`parallel sections`** pragma
- If multiple **`sections`** pragmas inside one parallel block, may reduce fork/join costs

Another Approach

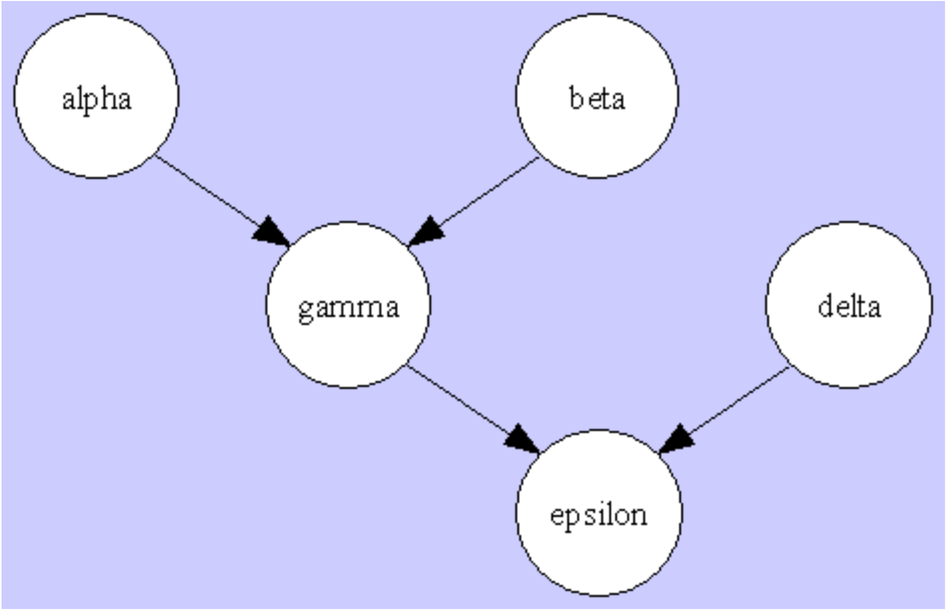


Execute alpha and beta in parallel. Execute gamma and delta in parallel.

Use of sections Pragma

```
#pragma omp parallel
   {
   #pragma omp sections
      {
         v = alpha();
      #pragma omp section
         w = beta();
      }
   #pragma omp sections
      {
         x = gamma(v, w);
      #pragma omp section
         y = delta();
      }
   }
   printf ("%6.2f\n", epsilon(x,y));
```

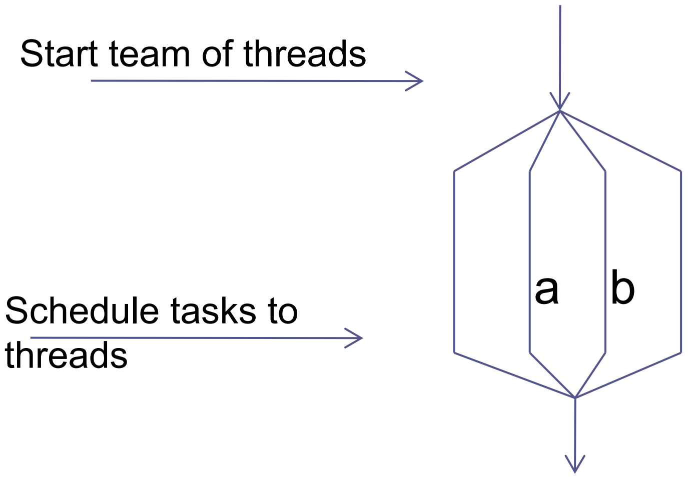
Another Approach



task Construct

- Binding
  - Thread set → inner most parallel region (current parallel team)

```
#pragma omp parallel
{
   #pragma omp single nowait
   {
        #pragma omp task
            b = beta();
        #pragma omp task
            a = alpha();
   }
}
```



Why is single used to start the parallel region?

task Construct

- Variables
  - By default variables are *firstprivate*. To share we need to say it explicitly.

```
int a=2, b=3, c;
#pragma omp parallel
{
   #pragma omp single nowait
   {
        #pragma omp task
            b = beta();
        #pragma omp task
             a = alpha();
   }
}
c = a + b;   // c = ?
```

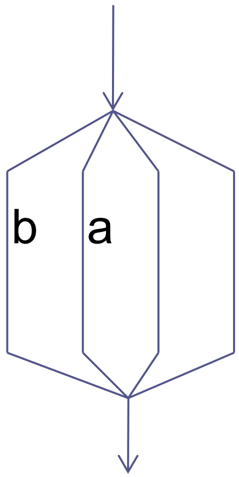
```
int a=2, b=3, c;
#pragma omp parallel
{
   #pragma omp single nowait
   {
        #pragma omp task shared(b)
            b = beta();
        #pragma omp task shared(a)
             a = alpha();
   }
}
c = a + b;   // c = ?
```

task Construct

- No need to create extra tasks
  - Child tasks are executed concurrently with their parent

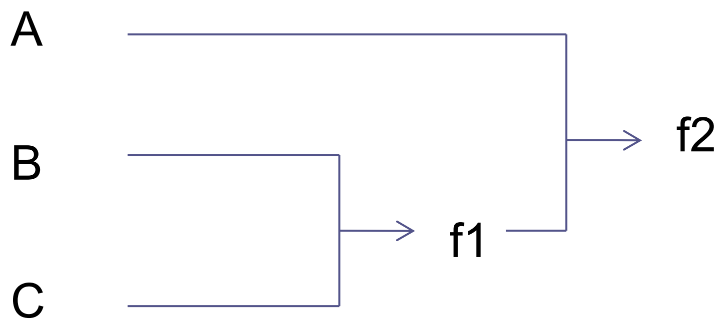
```
#pragma omp parallel
{
   #pragma omp single nowait
   {
        #pragma omp task
            b = beta();

        a = alpha();
   }
}
```



task Construct

- *taskwait*
  - Specifies a wait on the completion of child tasks generated since the beginning of the current task.



y = A()
v = B()
w = C()
x = f1(b, c)
e = f2(y,x)

task Construct - *taskwait*

1st version

```
#pragma omp parallel
{
   #pragma omp single nowait
   {
        #pragma omp task shared(a)
            y = A();
        #pragma omp task shared(b)
            v = B();
        #pragma omp task shared(c)
            w = C();

        #pragma omp taskwait

        x = f1(v,w)

        e = f2(y,x)
   }
}
```

Should be a taskgroup …

but there is no such construct

What can be improved?

Lab work

task Construct - *taskwait*

2nd version

```
#pragma omp parallel
{
   #pragma omp single nowait
   {
        #pragma omp task shared(a)
            y = A();
        #pragma omp task if(0) shared (b, c)
        {
                #pragma omp task shared(b)
                    v = B();
                #pragma omp task shared(c)
                    w = C();

                #pragma omp taskwait
        }
        x = f1(v,w)
        #pragma omp taskwait
        e = f2(y,x)
   }
}
```

Is this solution correct?

Compare to *sections pragma*, page 64.

Lab work

task Construct - *taskwait*

OpenMP Application Program Interface, page 60:

When an **if clause is present on a task construct** and the **if clause** expression evaluates to ***false***, *the encountering thread must suspend the current task region and* begin execution of the generated task immediately, and the suspended task region may not be resumed until the generated task is completed.

task Construct - reduction

```
int count_good (item_t *item)
{
    int n = 0;
    int pn[P]; /* P is the number of threads used. */
    #pragma omp parallel
    {
        pn[omp_get_thread_num()] = 0;
        #pragma omp single nowait
        {
            while (item) {
                #pragma omp task firstprivate(item)
                {
                    if (is_good(item)) {
                        pn[omp_get_thread_num()] ++;
                    }
                }
                item = item->next;
            }
        }
        #pragma omp barrier
        #pragma omp atomic
        n += pn[omp_get_thread_num()];
    }
    return n;
}
```

Credits to Yuan Lin, Sun Microsystems OpenMP team

Exercise: Pipeline with tasks

```
#pragma omp parallel private(i)
{
    /* preload data to be used in first iteration of the i-loop */
    #pragma omp single
            {ReadFromFile(0,...);}

    for (i=0; i<N; i++) {
            /* preload data for next iteration of the i-loop */
            #pragma omp single nowait
            {ReadFromFile(i+1...);}

            #pragma omp for schedule(dynamic)
            for (j=0; j<ProcessingNum; j++)
                    ProcessChunkOfData(); /* here is the work */
            /* there is a barrier at the end of this loop */

            #pragma omp single nowait
                    {WriteResultsToFile(i);}

    } /* threads immediately move on to next iteration of i-loop */

} /* one parallel region encloses all the work */
```

/* Fig 5.28 from "Using OpenMP" */

Lab work

Nested parallel regions: parallel regions vs tasks

```
void quick_sort (int p, int r, float *data)
{
        If (p < r) {
                Int q = partition (p, r, data);
                #pragma omp parallel sections firstprivate(data, p, q, r)
                {
                        #pragma omp section
                        quick_sort (p, q-1, data, low_limit);
                        #pragma omp section
                        quick_sort (q+1, r, data, low_limit);
                }
        }
}
void par_quick_sort (int n, float *data)
{
        quick_sort (0, n, data);
}
```

Only 2 threads have work.
What happens if we put 4, 6 or 8?

Nested parallel regions vs tasks

```
void quick_sort (int p, int r, float *data)
{
        If (p < r) {
                int q = partition (p, r, data);
                #pragma omp task
                quick_sort (p, q-1, data, low_limit);
                #pragma omp task
                quick_sort (q+1, r, data, low_limit);
                }
}
void par_quick_sort (int n, float *data)
{
        #pragma omp parallel
        {
                #pragma omp single nowait
                quick_sort (0, n, data);
        }
}
```

Here we have a single place
to select the number of threads

Main difference between parallel regions and tasks

Once a Parallel region is created:

- No threads in the team can leave the region until the end of the region
- No threads can join the parallel region

Results

OMP_NUM_THREADS	Task	Nested parallelism
2	2.6s	1.8s
4	1.7s	2.1s
8	1.2s	2.6s

Task version:

- any thread can work on any task resulting a more efficient thread usage
- There is only one parallel region and therefore the user can control better the number of threads used.

Credits to Richard Friedman, Sun Microsystems OpenMP team, 2007

Summary (1/3)

- OpenMP an API for shared-memory parallel programming
- Shared-memory model based on fork/join parallelism
- Data parallelism
  - parallel for pragma
  - reduction clause

Summary (2/3)

- Functional parallelism: parallel sections and task constructs.
- SPMD-style programming (parallel pragma)
- Critical sections (critical pragma)
- Atomic construct
- Enhancing performance of parallel for loops
  - Inverting loops
  - Nowait
  - Conditionally parallelizing loops
  - Changing loop scheduling

Summary (3/3)

Characteristic	*OpenMP*	*MPI*
Suitable for multiprocessors	Yes	Yes
Suitable for multicomputers	No	Yes
Supports incremental parallelization	Yes	No
Minimal extra code	Yes	No
Explicit control of memory hierarchy	No	Yes