OpenMP

Shared-memory Programming
Using OpenMP compiler directives

Outline

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

What is OpenMP

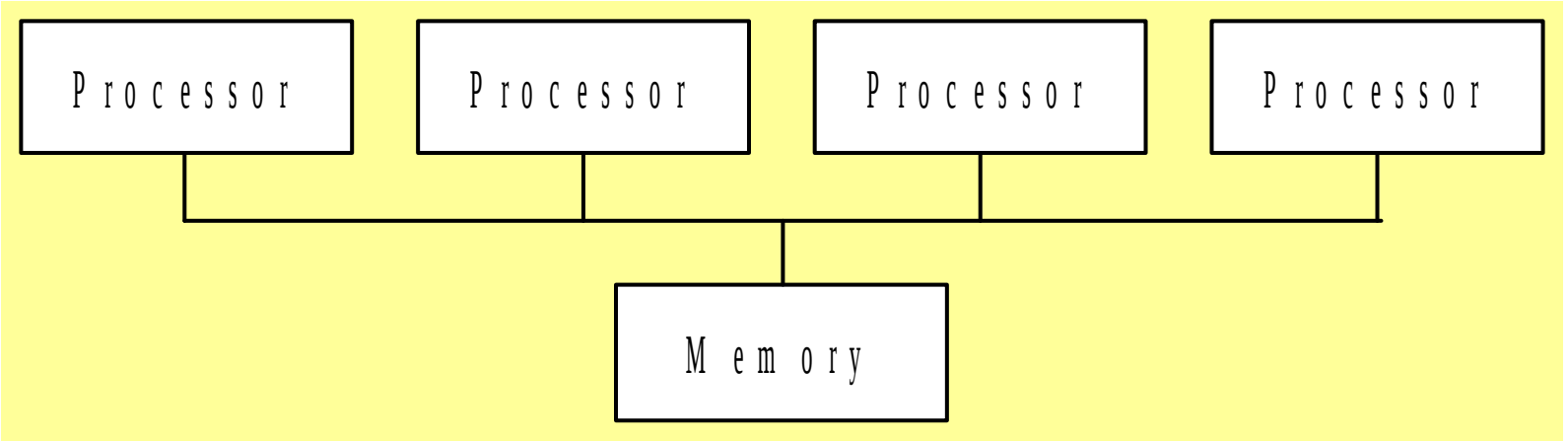
OpenMP: An application programming interface (API) for parallel programming on multiprocessors

- Compiler directives
- Library of support functions

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

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications.

Shared memory model

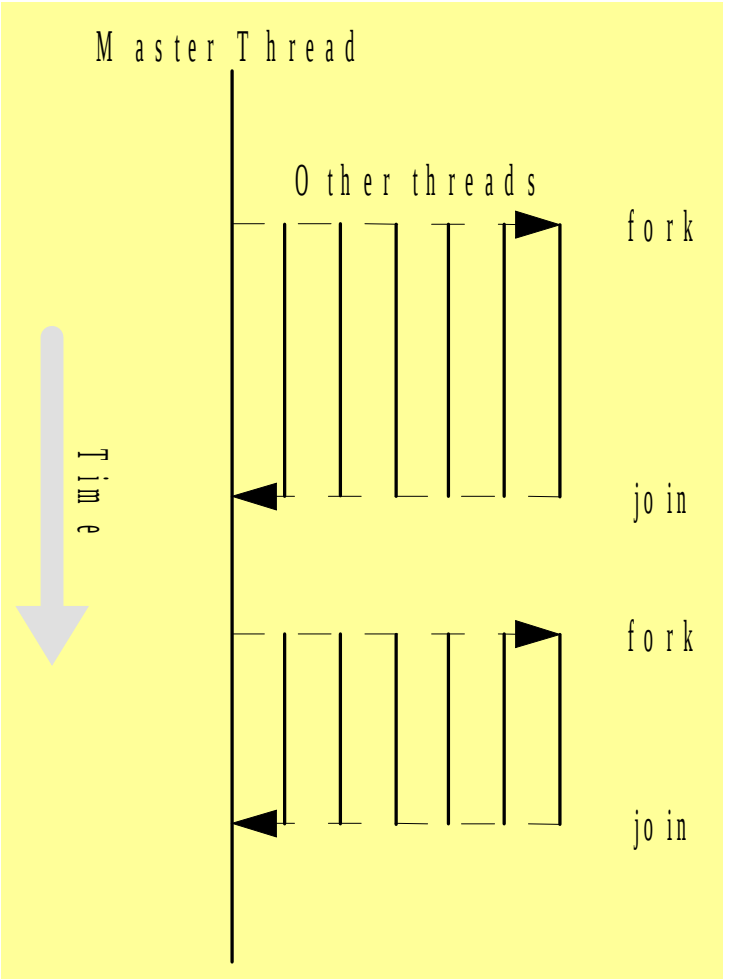


- We know from using pthreads that processors interact and synchronize with each other through shared variables.

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

Fork/join parallelism (cont.)



Incremental Parallelization

- Start with a serial program that is organized to perform its analysis using a looping construct
- Execute and profile the sequential program
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit (one loop) at a time
- Stop when further effort not warranted – no increase in speed or efficiency is observed
- The resulting parallel shared-memory program may only have a single parallel loop

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 for an OpenMP pragma:

#pragma omp *<rest of pragma>*

{block}

There is an implicit barrier at the end of the parallel block.

Parallelizing a for loop

Format:

```
//Insert a compiler directive before the loop
#pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = b[i] + c[i];
```

The compiler must be able to verify the run-time system will have the information it needs to schedule the loop iterations using a group of threads.

Different versions of a for loop can be parallelized

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

Review of pthread properties

- Every thread has its own execution context
- Execution context: address space containing all of the local 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

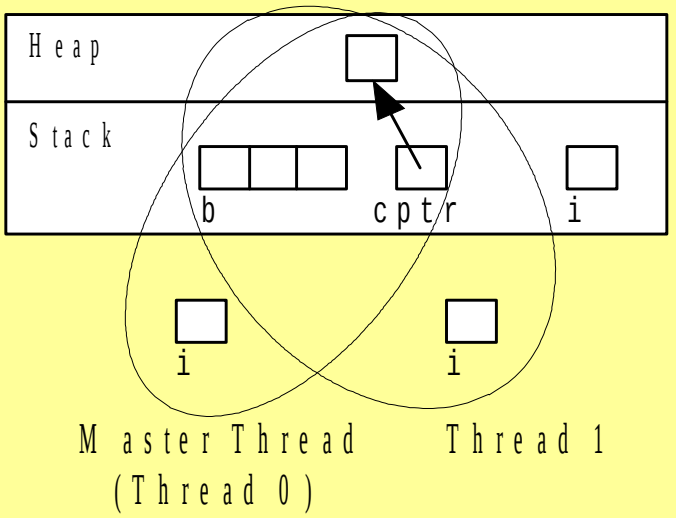
Shared vs local (private) variables

- Shared variable: has the same address in the execution context of every thread
- Local (private) variable: has a unique address in the execution context of every thread
- A thread cannot access the local variables of another thread

Threads in OpenMP have the same definition of shared and locale variables as pthreads

```
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;
```



Useful Openmp functions

1. A function that returns number of physical processors available for use by the parallel program.

int omp_get_num_procs (void)

2. A function that uses its parameter value to set the number of active threads in parallel sections of code.

May be called at multiple points in a program

void omp_set_num_threads (int t)

3. A function that returns how many threads are currently in use:

int omp_get_num_threads (void)

4. Find out the current thread ID,

int omp_get_thread_num (void)

The master thread is 0.

How is the number of threads specified?

- A clause after the parallel pragma
- The function **omp_set_num_threads()**
- An environmental variable, **OMP_NUM_THREADS**

Why are private variables needed

```
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
- Making the outer loop parallel reduces the number of forks/joins
- Each thread is given its own private copy of variable **`j`** so there is no confusion about which thread is doing a calculation

How to declare a private variable

A clause is an optional, additional component to a pragma

The private clause directs the compiler to make one or more variables private

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

```
#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);
```

Specialized clauses: firstprivate clause

- The firstprivate clause is 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

Lastprivate clause

In a sequential calculation, the last iteration occurs before the loop is exited. A for loop variable will be given its last value before the condition is tested.

The **`lastprivate`** clause is used to copy back the value of a private variable in the thread that executes the sequentially last iteration to the master thread's copy of a variable.

Reductions

- Reductions, combining many values to a single value, are so common that OpenMP provides support for them
- A reduction clause can be added to the **`parallel for`** pragma
- Both the reduction operation and the reduction variable must be specified
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop

The reduction clause

The reduction clause has this syntax:

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

Possible 0perators

- `+` Sum
- `*` Product
- `&` Bitwise and
- `|` Bitwise or
- `^` Bitwise exclusive or
- `&&` Logical and
- `||` Logical or

Example of a reduction

C program segment to compute π using the rectangle rule

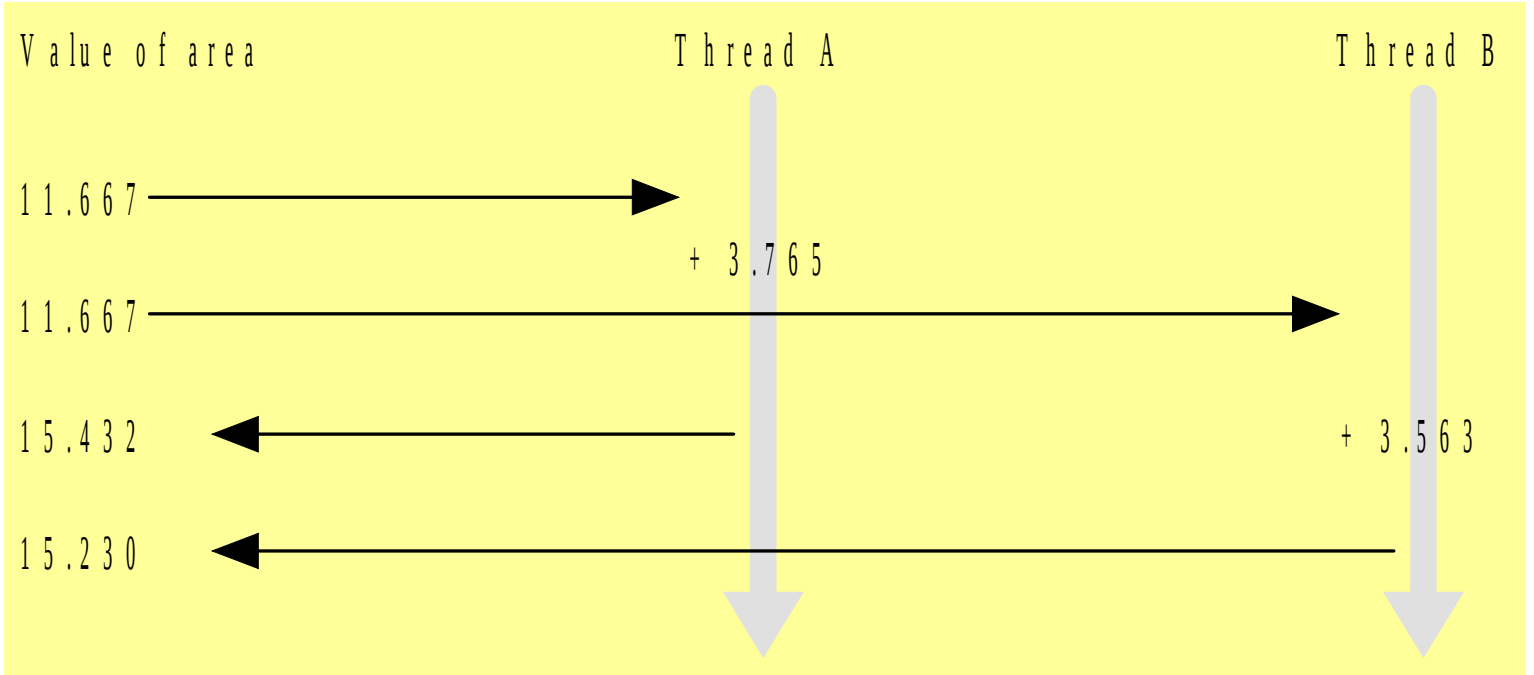
```
    double area, pi, x;
    int i, n;
...
    area = 0.0;
    for (i = 0; i < n; i++) {
       x += (i+0.5)/n;
       area += 4.0/(1.0 + x*x);
    }
    pi = area / n;
```

Insert 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;
```

Synchronization and critical sections

A race condition is one in which one thread may “race ahead” of another and not see the change to shared variable or access a shared resource inappropriately:



In the calculation of π

A race condition may occur 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)
```

The critical pragma

Critical section: a portion of code that only one thread at a time may execute

A critical section is identified 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) reduction(+:area)
    for (i = 0; i < n; i++) {
      x += (i+0.5)/n;
    #pragma omp critical
      area += 4.0/(1.0 + x*x);
    }
    pi = area / n;
```

Sources 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

Performance Improvements

- Too many fork/joins can lower performance
- Inverting loops may help performance if the parallelism is in the inner loop
- After inversion, the outer loop can be made parallel
- Inversion does not significantly lower cache hit rate
- If a 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)
```

Use scheduling to improve performance

The schedule clause can be used to specify how the iterations of a loop should be allocated to threads. Why? There may be an imbalance in the workload per thread:

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

The imbalance:

i = 0, j ranges from 0 to n-1
i = 1, j ranges from 1 to n-1
i = n-2, j ranges from n-2 to n-1

Static versus dynamic scheduling

Static scheduling: all iterations allocated to threads before any iterations executed.

- Low overhead
- May exhibit high workload imbalance

Dynamic scheduling: only some iterations allocated to threads at the beginning of the loop's execution. Remaining iterations allocated to threads that complete their assigned iterations.

- Higher overhead
- Can reduce workload imbalance

Granularity in parallel programming - chunks

A chunk is a contiguous range of iterations of a loop – e.g. a group of 50 iterations

- Increasing chunk size reduces overhead and but may increase cache hit rate
- Decreasing chunk size allows finer balancing of workloads per thread

The 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/numthreads contiguous iterations to each thread
- schedule(static,m): interleaved allocation of chunks of size m to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,m): dynamic allocation of m iterations at a time to threads

Scheduling options (cont.)

- schedule(guided, m): dynamic allocation of chunks to tasks using a guided, self-scheduling algorithm. Initial chunks are bigger, later chunks are smaller, minimum chunk size is m.
- schedule(guided): guided self-scheduling with minimum chunk size 1
- schedule(runtime): schedule chosen at run-time based on value of OMP_SCHEDULE; Linux example:

  setenv OMP_SCHEDULE “static,1”

Other forms of data parallelism

Focus has been on the parallelization of `for` loops

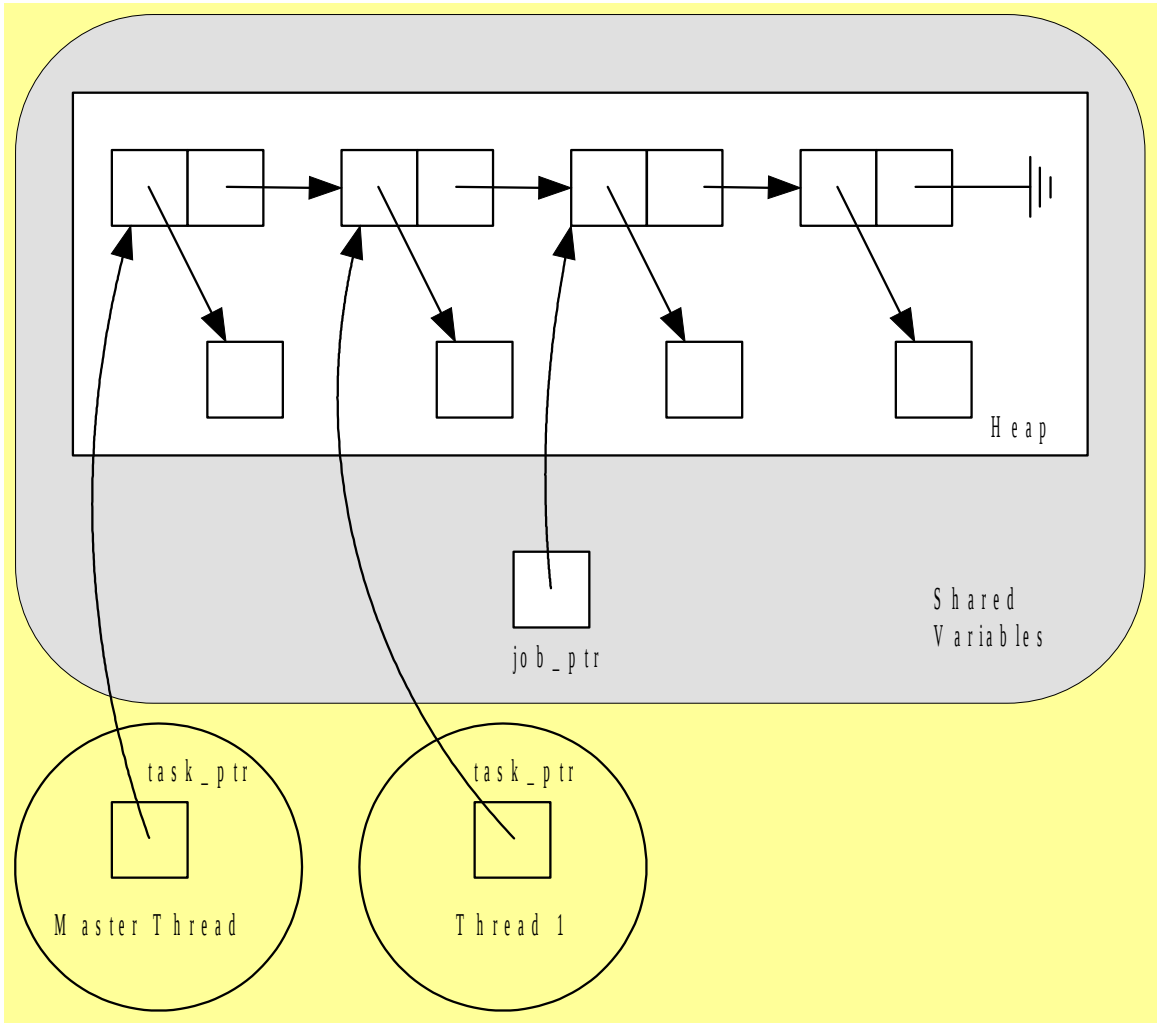
Other opportunities for data parallelism exist:

- processing items on a “to do” list
- `for` loop + additional code outside of loop

Handling a list of tasks

- Every thread takes a next task from the list and completes it.
- The threads continue to remove tasks from the list until there are no more tasks.
- Must ensure no two threads take the same task from the list, e.g., a critical section can be declared.

The task list represented as a linked list



A sequential code version of the task list

```
int main (int argc, char *argv[])
{
   struct job_struct  *job_ptr;
   struct task_struct *task_ptr;


   ...
   task_ptr = get_next_task (&job_ptr);
   while (task_ptr != NULL) {
      complete_task (task_ptr);
      task_ptr = get_next_task (&job_ptr);
   }
   ...
}
```

get_next_task() function

```
char *get_next_task(structjob_struct**job_ptr)
{
   struct task_struct *answer;

   if (*job_ptr == NULL) answer = NULL;
   else {
      answer = (*job_ptr)->task;
      *job_ptr = (*job_ptr)->next;
   }
   return answer;
}
```

The parallel pragma

The parallel pragma precedes a block of code that should be executed by *all* of the threads.

```
#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);
   }
}
```

The critical section is inserted in the function get_next_task()

```
char *get_next_task(struct job_struct**job_ptr)
{
   struct task_struct *answer;
#pragma omp critical
   {
   if (*job_ptr == NULL) answer = NULL;
   else {
      answer = (*job_ptr)->task;
      *job_ptr = (*job_ptr)->next;
   }
   }
   return answer;
```

The single pragma

Suppose we only want to see the output of a calculation once rather than printed by every thread.

The **`single`** pragma indicates to the compiler that only a single thread should execute the block of code the pragma precedes.

Syntax:

#pragma omp single

Use of the single pragma

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("low > high at (%d)\n", i);
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

Note, low and high are not private variables.

The nowait clause

- The compiler puts a barrier synchronization at end of every parallel for statement, block or section.
- The nowait clause indicates that the barrier can be eliminated. That is, the threads can terminate or move ahead independently.
- In the previous example, the barrier was needed if a thread changes **`low`** or **`high`** which may affect the behavior of another thread.
- If **low** and **high** were private variables, then it would be okay to let threads move ahead, which could reduce execution time.

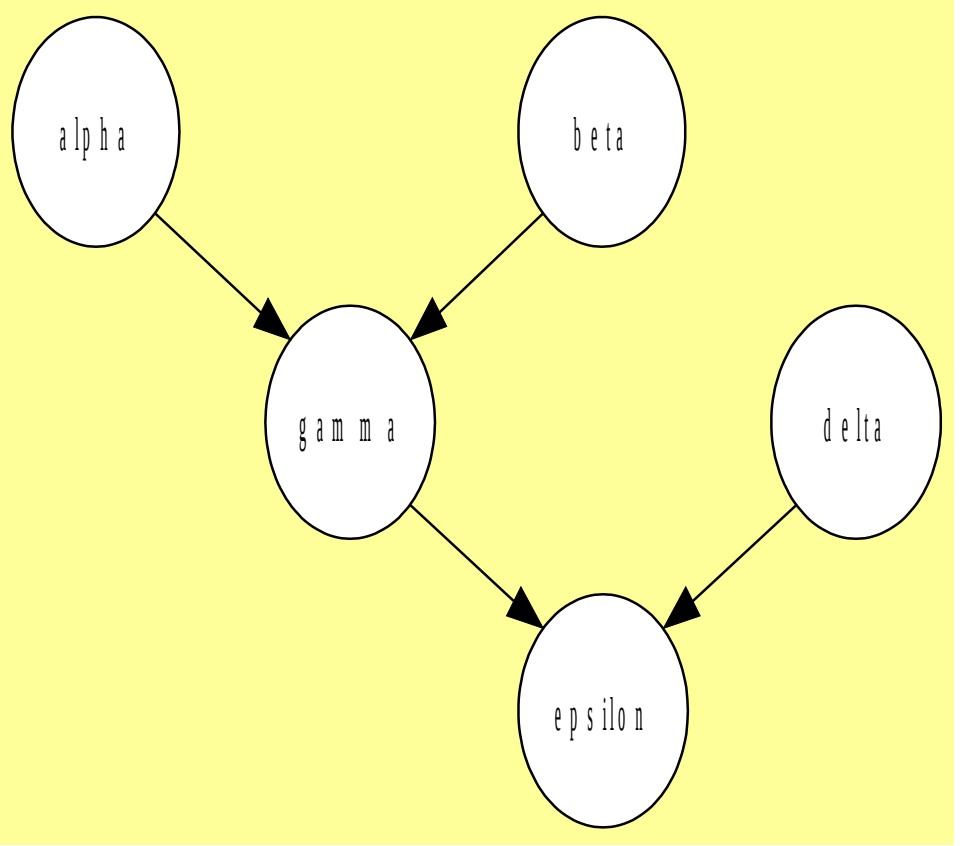
Use of the nowait clause

```
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("low > high at (%d)\n", i);
   }
#pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

Functional parallelism in OpenMP

```
    v = alpha();
    w = beta();
    x = gamma(v, w);
    y = delta();

printf("%6.2f\n",epsilon(x,y));
```



Functions alpha(), beta(), and delta() may be executed in parallel.

Working sharing using the parallel sections pragma

The parallel sections pragma allows a task to be shared be shared among a pool of threads.

It precedes a block of *k* blocks of code that may be executed concurrently by *k* threads

Syntax:

```
#pragma omp parallel sections
```

The section pragma

- The parallel sections pragma is followed by multiple blocks of code
- Each block of code is preceded by the section pragma
- It may be omitted for the first parallel section after the parallel sections pragma

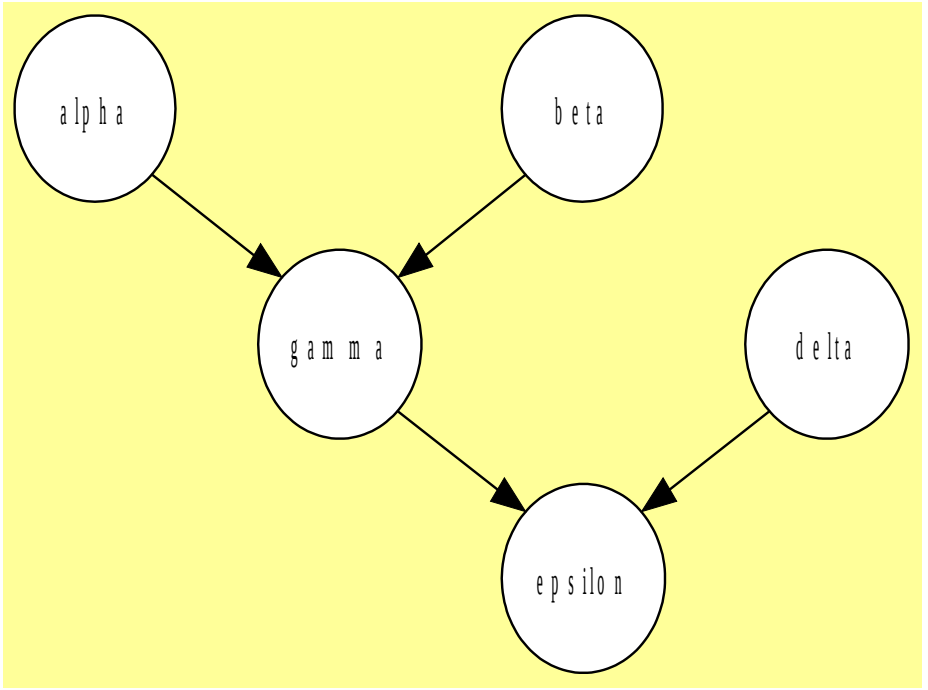
Syntax:

```
#pragma omp section
```

Usage of the parallel sections and section pragmas

```
#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));
```

Another approach to work-sharing in OpenMP



Execute the functions alpha() and beta() in parallel.

Execute gamma() and delta() in parallel.

Only epsilon() has dependencies that require waiting for gamma() and delta().

The sections pragma

The sections pragma is used separately from the parallel pragma.

- It appears inside a parallel block of code
- Used within a parallel pragma block, it has the same meaning as the parallel sections pragma
- If multiple sections pragmas are used inside one parallel block, the fork/join costs may be reduced.

Usage of the 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));
```

Barriers in OpenMP

The barrier clause causes execution to wait for all threads to finish the work of the loop, sections, or region before any go on to execute additional work.

Once all threads have reached the barrier, they can proceed. The form is given by:

```
#pragma omp barrier
```

This statement must be within another parallel pragma.

Example of the use of a barrier

```
#pragma omp parallel shared(x, y, z) num_threads(2)
   {
      int tid = omp_get_thread_num();
      if (tid == 0 )
          y= fn1(tid);
      else
          z = fn2(tid);
      #pragma omp barrier
      #pragma omp for
         for (j = 0; j < 100; j++)
             x[j] = y + z + fn1(j) + fn2(j);
   }
```

The code is executed by two threads. Thread 0 calculates the value assigned to the variable y; the other thread assigns a value to the variable z. Both y and z are needed in the for loop; hence, two data dependences exist. The barrier is needed to ensure that the values of y and z have been calculated before the loop is executed.

The atomic clause

With the atomic clause, the memory update (write, or read-modify-write) in the next instruction will be performed atomically – no interleaved instructions. It does not make the entire statement atomic; only the memory update is atomic.

```
#pragma omp atomic
        expression or statement
```

Possible forms: x op= expression

x++, x--

++x, --x

Atomic vs critical clauses

A critical section is completely general; it can surround any arbitrary block of code.

- It incurs significant overhead every time a thread enters and exits the critical section
- The critical section also serializes the code.
- There is only one lock for all unnamed critical sections

An atomic operation has much lower overhead.

- It relies on the hardware providing the atomic increment operation
- No lock/unlock needed on entering/exiting the line of code
- One thread being in an atomic operation doesn't block any other atomic operations.
- The calculation is still serialized.

Timing omp code blocks

A function is available that returns the elapsed wall clock time in seconds.

```
double omp_get_wtime(void);
```

The time is measured per thread; however, no guarantee can be made that two distinct threads measure the same time.

Time is measured from some "time in the past", which is an arbitrary time guaranteed not to change during the execution of the program.

Timing omp code blocks (cont.)

The usage is to call the function twice and subtract the two values:

```
    double start;
    double end;
    start = omp_get_wtime();
//... work to be timed ...
    end = omp_get_wtime();
    printf("Work took %f sec. time.\n", end-start);
```

The flush clause

The flush clause causes all threads in a parallel region to have a consistent view of a specific or all shared variables in memory.

#pragma omp flush (variable list)

^^ no list - all shared variables become consistent

- Current read/write operations on variables are allowed to complete.
- Values are written back to memory
- New memory operations are postponed

This clause must appear within a compound statement.

The threadprivate clause

Some variables used in parallel blocks need to be private to a specific thread – no sharing.

#pragma omp threadprivate (variable list)

- The variable will persist from one parallel region to the next – it assumes a fixed number of threads.
- It must be initialized before it is declared private.
- The directive must come before any parallel directive.