Parallel Programming

in C with MPI and OpenMP

Michael J. Quinn

(Wonyong Sung modification)

McGraw Hill

Chapter 17

Shared-memory Programming

Outline

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

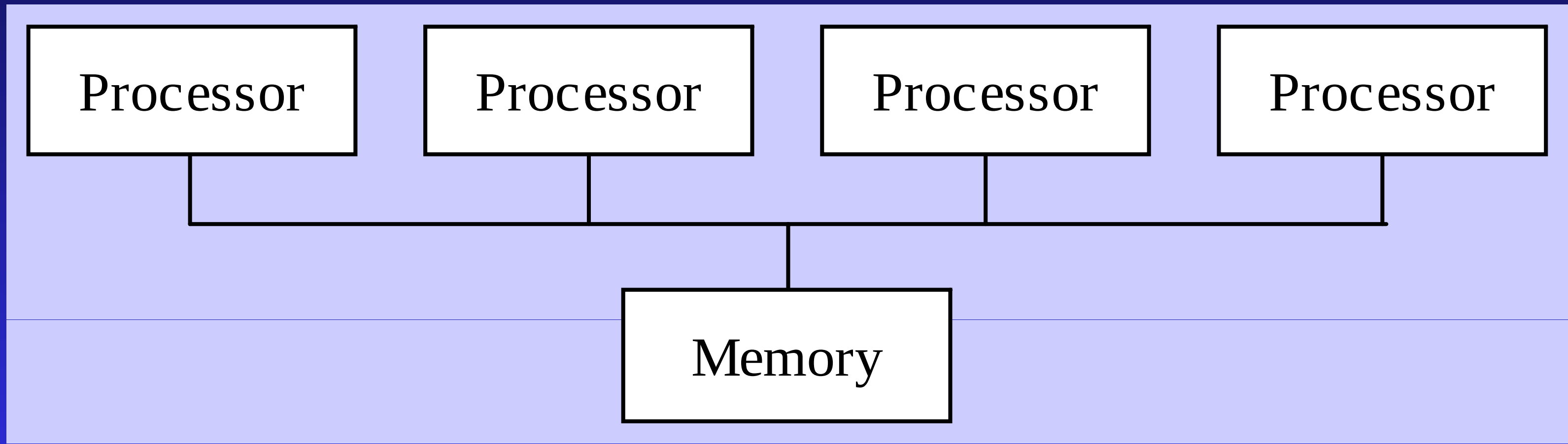
OpenMP

- OpenMP: An application programming interface (API) for parallel programming on multiprocessors
  - Compiler directives
  - Library of support functions
  - Environment variables
- OpenMP works in conjunction with Fortran, C, or C++

What's OpenMP Good For?

- C + OpenMP sufficient to program multiprocessors
- C + MPI + OpenMP a good way to program multicomputers built out of multiprocessors
  - IBM RS/6000 SP
  - Fujitsu AP3000
  - Dell High Performance Computing Cluster

Shared-memory Model



Processors interact and synchronize with each other through shared variables.

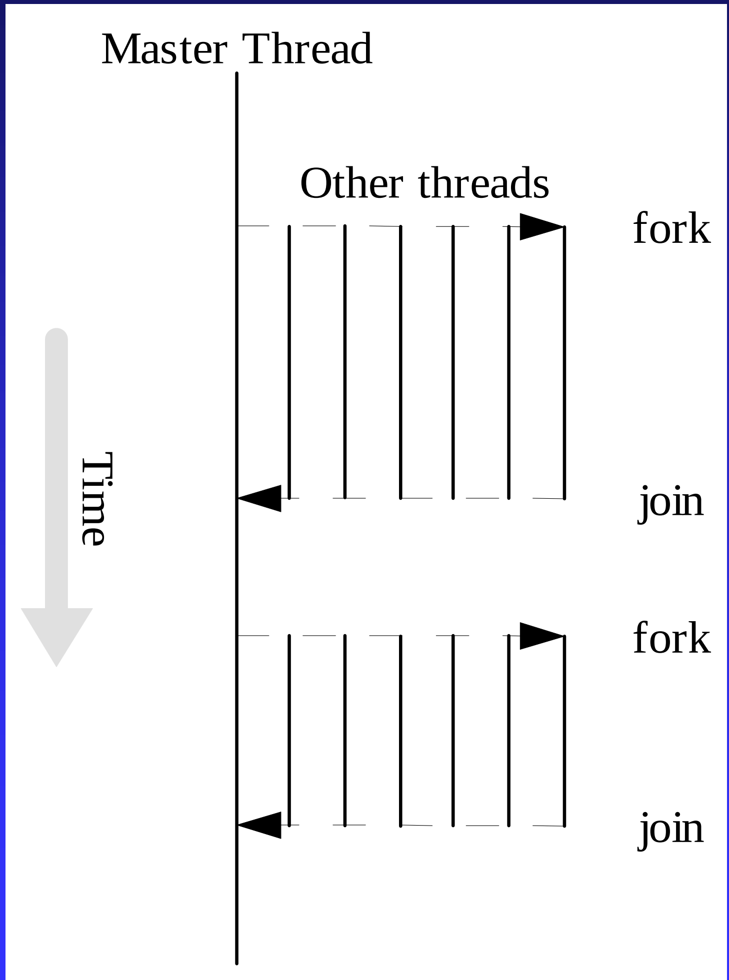
no need of explicit communication

but need to synchronize and protect private date

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
- Advantages and Disadvantages:
  - Support incremental parallelization (allows begin with sequential program)
  - Speed-up limited by the master-only part

Fork/Join Parallelism



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

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

Incremental Parallelization

- Sequential program 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
  - For loop
  - Parallel execution of functions

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

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

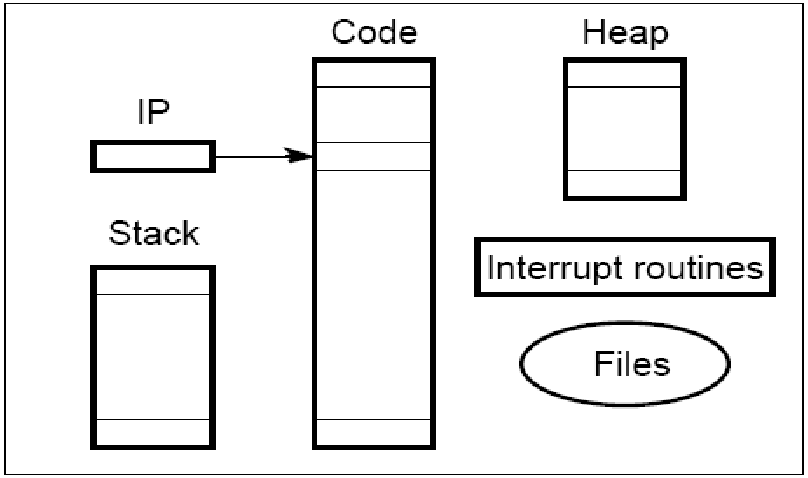
- Uses the parameter value to set the number of threads to be active in parallel sections of code
- May be called at multiple points in a program

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

Differences between a process and threads

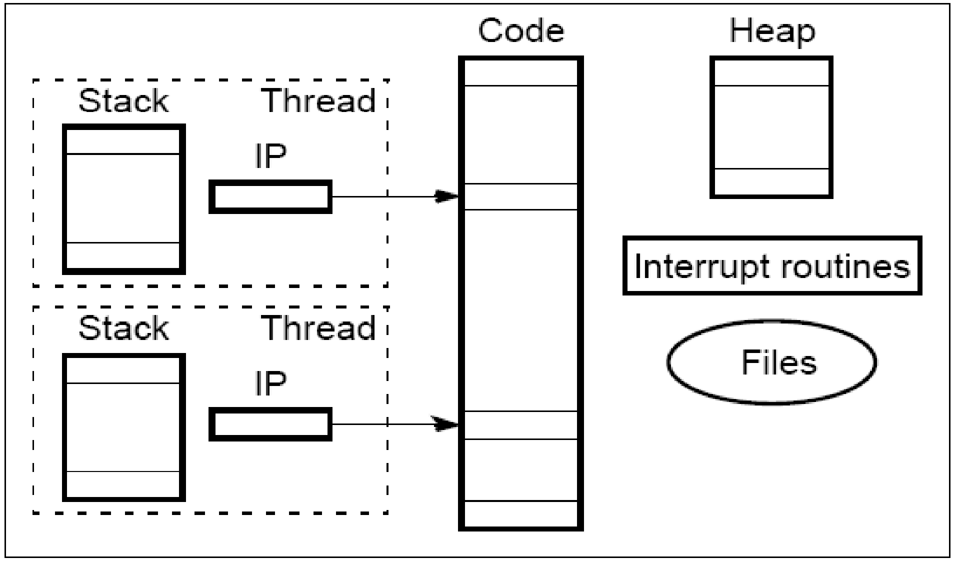
"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



Threads - shares the same memory space and global variables between routines.

(b) Threads



Pop Quiz:

Write a C program segment that sets the number of threads equal to the number of processors that are available.

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>*

Parallelization

- For loop
- Functions

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

Parallel for Pragma

- Format:

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

- Compiler must be able to verify the run-time system will have information it needs to schedule loop iterations

Canonical Shape of for Loop Control Clause

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

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

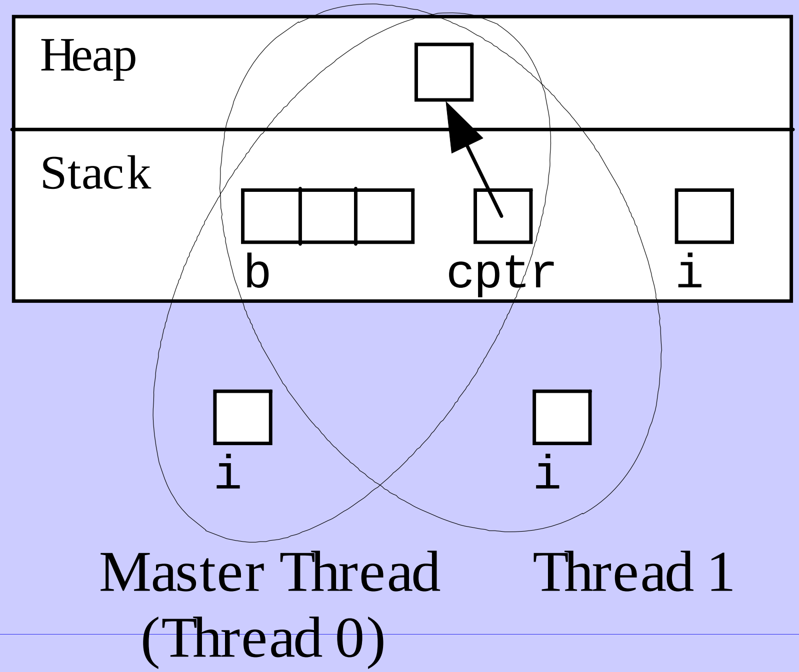
Shared and Private Variables

- 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;
```



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);

<- the program is divided for i,
    no need of declaring i as a private
```

firstprivate Clause

- 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

- ```
  x[0] = complex_function();
  #pragma omp parallel for private(j) firstprivate(x)
  for (i=0; i<n; i++){
     for (j=1; j<4; j++)
           x[j] = g(i, x[j-1]);
     answer[i] = x[1]-x[3];
  }
  ```
  ```

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

Critical Sections – a portion of code that only one thread at a time may execute

`#pragma omp critical`

- PI calculation using arctan function
- Integration of 1/1+x*x is arctan

```
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)
```

Correct, But Inefficient, Code

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

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
  - Inversion does not significantly lower cache hit rate

```
for (i=1; i<m; i++)
   for (j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];

#pragma omp parallel for private(i)
for (j=0; j<n; j++)
   for (i=1; i<m; i++)
      a[i][j] = 2 * a[i-1][j];
```

Performance Improvement #2

- 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

- 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

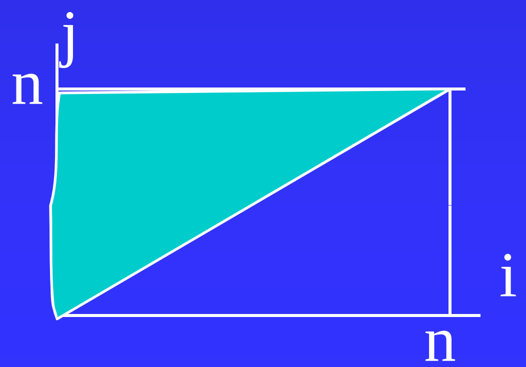
Scheduling Options

- 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”`

- #pragma omp parallel for private(j) schedule(*,*)
  for (i=0; i<n; i++)
    for (j=i; j<n; j++)
      a[i][j] = alpha_omega(I,j);

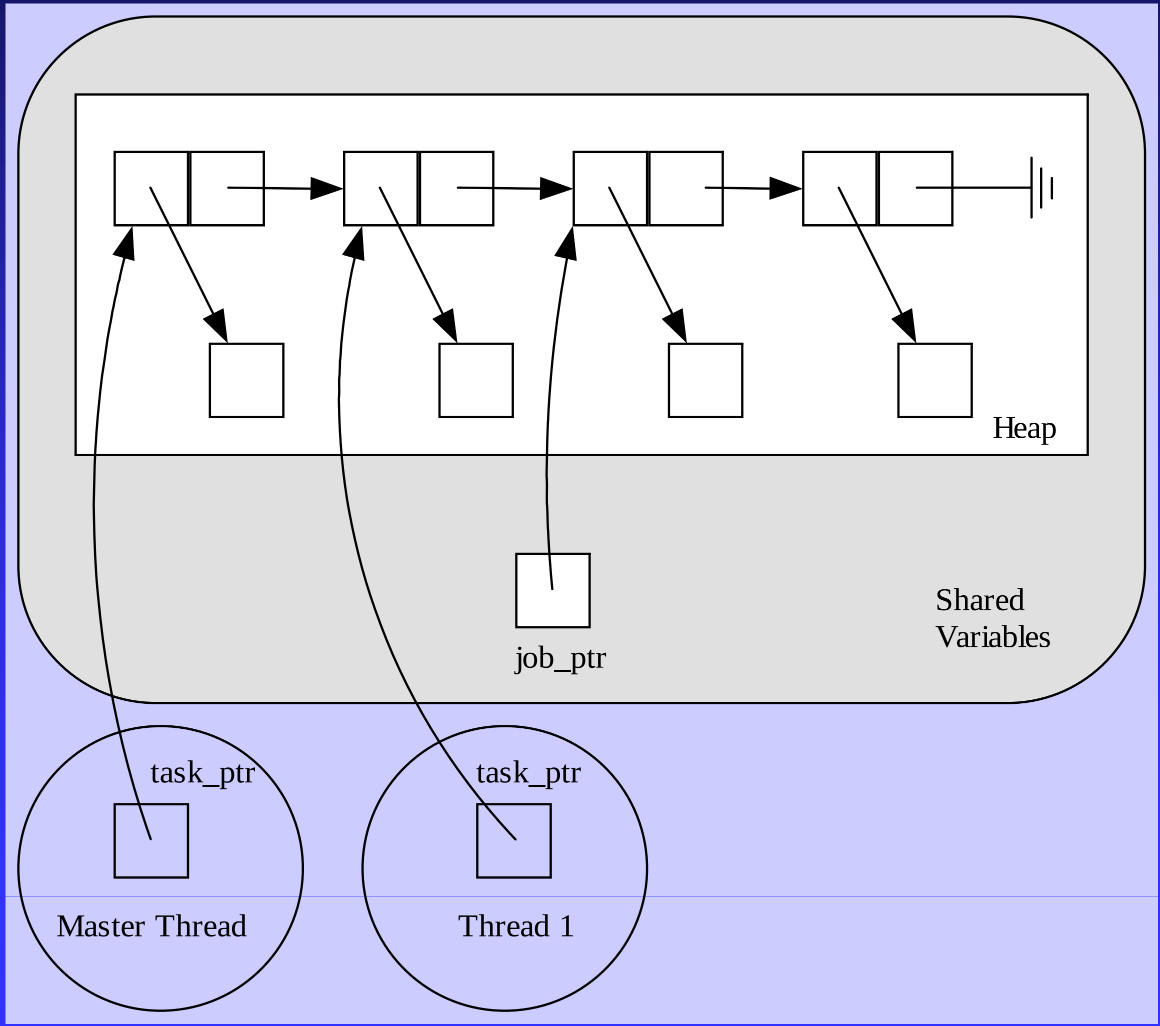


Static schedule with n/p chunks makes unalanced load distribution

More General Data Parallelism

- Our focus has been on the parallelization of `for` loops
- Other opportunities for data parallelism
  - processing items on a "to do" list
  - `for` loop + additional code outside of loop

Processing a “To Do” List



Sequential Code (1/2)

```
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);
   }
   ...
}
```

Sequential Code (2/2)

```
char *get_next_task(struct job_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;
}
```

Parallelization Strategy

- Every thread should repeatedly take next task from list and complete it, until there are no more tasks
- We must ensure no two threads take same task from the list; i.e., must declare a critical section

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

Parallel Regions

- Fundamental OpenMP construct:
  - `#pragma omp parallel`

```
#pragma omp parallel
{
  printf( "hello world from thread %d of
      %d\n", omp_get_thread_num(),
      omp_get_num_threads() );
}
```

- From an 8-processor machine:

```
hello world from thread 0 of 8
hello world from thread 2 of 8
hello world from thread 3 of 8
hello world from thread 7 of 8
hello world from thread 6 of 8
hello world from thread 1 of 8
hello world from thread 4 of 8
hello world from thread 5 of 8
```

Linked list processing

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

char *get_next_task(struct job_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;
}
```

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);
   }
}
```

Critical Section for **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;
}
```

Functions for SPMD-style Programming

- 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)
```

for Pragma

- The **`parallel`** pragma instructs every thread to execute all of the code inside the block
- If we encounter a **`for`** loop that we want to divide among threads, we use the **`for`** pragma

```
#pragma omp for
```

Example Use of for Pragma

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

single Pragma

- Suppose we only want to see the output once
- The **`single`** pragma directs compiler that only a single thread should execute the block of code the pragma precedes
- Syntax:

```
#pragma omp single
```

Use of 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 ("Exiting (%d)\n", i);
      break;
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

nowait Clause

- Compiler puts a barrier synchronization at end of every parallel for statement
- In our example, this is necessary: if a thread leaves loop and changes **`low`** or **`high`**, it may affect behavior of another thread
- If we make these private variables, then it would be okay to let threads move ahead, which could reduce execution time

Use of 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 ("Exiting (%d)\n", i);
      break;
   }
#pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

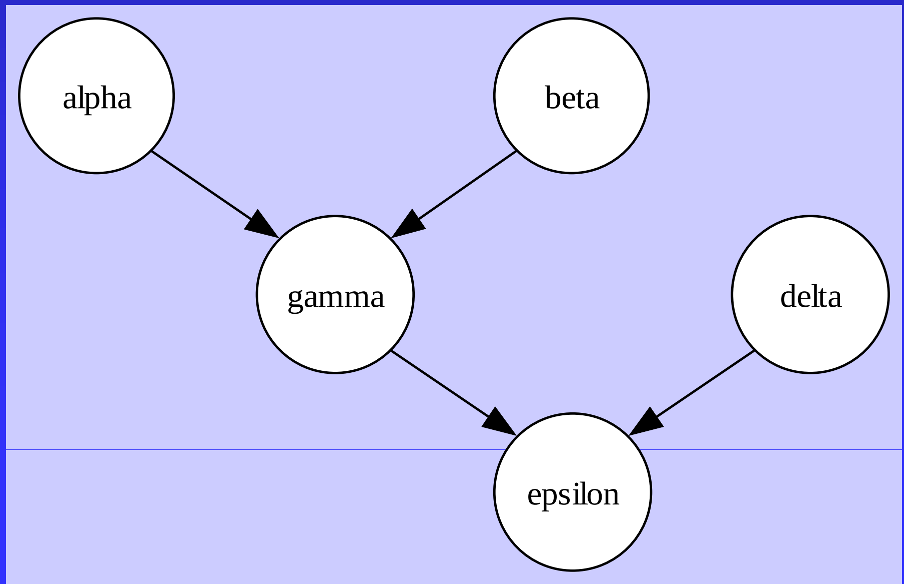
Functional Parallelism

- 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

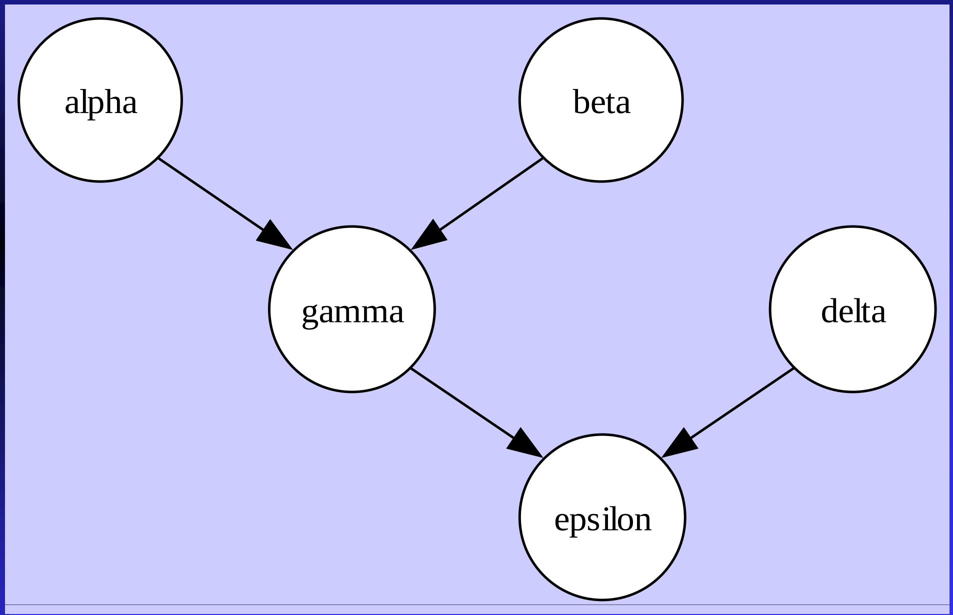
- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma
- Syntax:

```
#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));
```

Another Approach



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

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

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));
```

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 pragma)
- SPMD-style programming (parallel pragma)
- Critical sections (critical pragma)
- Enhancing performance of parallel for loops
  - Inverting loops
  - 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