Multicore Computing

Instructor:
Arash Tavakkol

Department of Computer Engineering
Sharif University of Technology
Spring 2016
Shared Memory Programming
Using OpenMP

Some Slides come From Parallel Programming in C with MPI and OpenMP By Michael J. Quinn &
An Overview of OpenMP By Ruud van der Pas – Sun Microsystems
Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - “Standard” API for defining multi-threaded shared-memory programs
  - openmp.org – Talks, examples, forums, etc.

- High-level API
  - Preprocessor (compiler) directives ( ~ 80% )
  - Library Calls ( ~ 19% )
  - Environment Variables ( ~ 1% )
A Programmer’s View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races
Motivation

- Thread libraries are hard to use
  - PThreads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness
- Wouldn’t it be nice to write serial programs and somehow parallelize them “automatically”?  
  - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence
  - It is not automatic: you can still make errors in your annotations
Motivation (Cont’d)

- Good performance and scalability
  - If you do it right ....
- De-facto standard
- An OpenMP program is portable
  - Supported by a large number of compilers
- Requires little programming effort
- Allows the program to be parallelized incrementally
- Maps naturally onto a multicore architecture:
  - Lightweight
  - Each OpenMP thread in the program can be executed by a hardware thread
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
The OpenMP Execution Model
What’s OpenMP Good For?

- C + OpenMP sufficient to program multiprocessors
- C + MPI + OpenMP a good way to program multicomputers built out of multiprocessors
OpenMP Core Syntax

- Most of the constructs are compiler directives:
  - `#pragma omp construct [clause [clause] ...]`

- Examples:
  - `#pragma omp parallel num_threads(4)`

- Function prototypes and types in the file
  - `#include <omp.h>`

- Most OpenMP constructs apply to a “structured block”
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
Hello world

- Write a multithreaded program that prints “hello world”.
- Switches for compiling and linking
  - `-fopenmp gcc`
```c
#include <omp.h

void main() {

#pragma omp parallel
{
  int ID = omp_get_thread_num();

  printf(" hello(%d)" , ID);
  printf(" world(%d)\n", ID);

}
}
```

- **OpenMP include file**
- **Parallel region with default number of threads**
- **Runtime library function to return ID.**

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```
Another OpenMP example

for-loop with independent Iteration

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

for-loop parallelized using OpenMP pragma

```c
#pragma omp parallel for 
    shared(n, a, b, c) 
    private(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```
Example Parallel Execution

\[\text{for } n = 1000\]
Terminology

- OpenMP Team := Master + Workers
- A Parallel Region is a block of code executed by all threads simultaneously
  - The master thread always has thread ID 0
  - Parallel regions can be nested, but support for this is implementation dependent
  - An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially
- A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work
Terminology

- Master thread + Workers
Number of threads

- In Unix, the environment variable OMP_NUM_THREADS provides a default number of threads.
- The number of threads is important. Each thread incurs an overhead. Too many threads may actually slow down the execution of a program.
```c
#include <omp.h>

void main() {
    double A[1000];
    #pragma omp parallel num_threads(4)
    {
        int id = omp_get_thread_num();
        somfunc(id, A);
    }
}
```

```c
#include <omp.h>

void main() {
    omp_set_num_threads(40);
    double A[1000];
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        somfunc(id, A);
    }
}
```
Parallel for Loops

- C programs often express data-parallel operations as `for` loops
  
  ```c
  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:

  ```c
  #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 information it needs to schedule loop iterations.
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 Memory programming model**
  - Most variables are shared by default
  - Shared variable: has same address in execution context of every thread
  - Global variables are SHARED among threads
  - C: File scope variables, static

- **Private variable: has different address in execution context of every thread**
  - Stack variables in functions called from parallel regions are PRIVATE
  - A thread cannot access the private variables of another thread
  - Attributes of construct variables can be changed
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;
}

Shared and Private Variables

Changing Storage Attributes

- One can selectively change storage attributes for constructs using the following clauses:
  - **SHARED**
  - **PRIVATE**
  - **FIRSTPRIVATE**

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - **LASTPRIVATE**

- The default attributes can be overridden with:
  - **DEFAULT (PRIVATE | SHARED | NONE)**
The private/shared clauses

- **Clause**: an optional, additional component to a pragma

  - **private (list)**
    - No storage association with original object
    - All references are to the local object
    - Values are undefined on entry and exit

  ```cpp
  #pragma omp parallel shared(n,x,y)
  private(i)
  {
    #pragma omp for
    for (i=0; i<n; i++)
      x[i] += y[i];
  } /*-- End of parallel region --*/
  ```

- **shared (list)**
  - Data is accessible by all threads in the team
  - All threads access the same address space
Declaring Private Variables

```c
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

- Private clause: directs compiler to make one or more variables private

```plaintext
private ( <variable list> )
```
Example Use of private Clause

```c
#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[j]);
```
About storage association

- Private variables are **undefined** on entry and exit of the parallel region.
- The value of the original variable (before the parallel region) is **undefined** after the parallel region!
- A private variable within a parallel region has no storage association with the same variable outside of the region.
- Use the **first/last private** clause to override this behavior.
- We illustrate these concepts with an example.
firstprivate Clause

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered (the value the original object had before entering the parallel construct)

- Variables are initialized once per thread, not once per loop iteration
Example firstprivate

```c
x[0]=complex_function();
#pragma omp parallel for private(j) firstprivate(x)
for (i = 0; i < n; i++){
    for (j = 0; 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.
Example lastprivate

- Each thread gets its own tmp with an initial value of 0
- tmp is defined as its value at the “last sequential” iteration (i.e. for j=999)

```c
int tmp = 0;
#pragma omp parallel for firstprivate(tmp)\   
        lastprivate(tmp)
for (i = 0; i < 1000; j++)
    tmp += j;
printf(“%d\n”, tmp);
```
Consider this example of PRIVATE and FIRSTPRIVATE:

variables A, B, and C = 1
# pragma omp parallel private(B) firstprivate(C)

Inside this parallel region:
- A is shared by all threads; equals 1
- B and C are local to each thread.
- B’s initial value is undefined
- C’s initial value equals 1
Default Clause

- Note that the default storage attribute is default(shared)
- To change default: default (private)
- each variable in the construct is made private as if specified in a private clause
- mostly saves typing
- default(none): nodefault for variables
- C/C++ only has default(shared) or default(none)
Example: Numerical Integration

- Mathematically, we know that:
  \[ \int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi \]

- We can approximate the integral as a sum of rectangles:
  \[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]
Example: Numerical Integration

- Serial Program:

```c
step_num = 100000;
double area, pi, x;
int i;
area = 0.0;
for (i = 0; i < step_num; i++) {
    x += (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / step_num;
```
Example: Numerical Integration

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - `int omp_get_num_threads();`
    - Number of threads in the team
  - `int omp_get_thread_num();`
    - Thread ID or rank
Critical Sections

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;
Consider this C program segment to compute $\pi$ using the rectangle rule:

```c
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;
```
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 Time Line

Value of area

Thread A

11.667 → + 3.765
11.667
15.432
15.230

Thread B

+ 3.563
15.432
15.230
Critical Pragma

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

```c
#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.
The reduction clause has this syntax:

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

Operators

- `+` Sum
- `*` Product
- `&` Bitwise and
- `|` Bitwise or
- `^` Bitwise exclusive or
- `&&` Logical and
- `||` Logical or
\[\pi\text{-finding Code with Reduction Clause}\]

```c
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;
```
Reduction

- **reduction** (<op> :<variable>)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Compiler finds standard reduction expressions containing “op” and uses them to update the local copy.
  - Local copies are reduced into a single value and combined with the original global value.

- The variables in “list” must be shared in the enclosing parallel region.
Reduction operands/initial-values

- Operators
  - + 0
  - * 1
  - - 0
  - & ~0
  - | 0
  - ^ 0
  - && 1
  - || 0
Synchronization

- **Critical** pragma: discussed previously
- **Atomic** provides mutual exclusion but only applies to the update of a memory location (the update of area in the following example)

```c
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
#pragma omp atomic
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```
Suppose we run each of these two loops in parallel over i:

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

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

This may give us a wrong answer

Why?
We need to have updated all of $a[\ ]$ first, before using $a[\ ]$

\[
\text{for (i=0; i < N; i++)}
\]
\[
\quad a[i] = b[i] + c[i];
\]

\[
\text{for (i=0; i < N; i++)}
\]
\[
\quad d[i] = a[i] + b[i];
\]

Wait!
Barrier

All threads wait at the barrier point and only continue when all threads have reached the barrier point
Synchronization: Barrier (Cont’d)

#pragma omp barrier
When to use barriers?

- When data is updated asynchronously and the data integrity is at risk
  - Examples:
    - Between parts in the code that read and write the same section of memory
    - After one timestep/iteration in a solver
- Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors
- Therefore, use them with care
Synchronization: Barrier

```c
#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_calc1(i,A);
    }  //implicit barrier at the end of for construct
    #pragma omp for
    for (i = 0; i < N; i++) {
        B[i] = big_calc2(C,i);
    }  //implicit barrier at the end of for construct
    A[id] = big_calc4(id);
}  //implicit barrier at the end of a parallel region
```
nowait Clause

- Compiler puts a barrier synchronization at end of every parallel for statement
- If there is no race condition or critical section, it would be okay to let threads move ahead, which could reduce execution time
nowait Clause

```
#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_calc1(i,A);
        //implicit barrier at the end of for construct
    } //pragma omp for nowait
    //no implicit barrier due to nowait clause
    for (i = 0; i < N; i++) {
        B[i] = big_calc2(C,i);
    }
    A[id] = big_calc4(id);
} //implicit barrier at the end of a parallel region
```
Synchronization: Lock routines

- Simple Lock routines
  - A simple lock is available if it is unset.

- `omp_init_lock()`
  - This subroutine initializes a lock associated with the lock variable.
  - The initial state is unlocked

- `omp_destroy_lock()`
  - This subroutine disassociates the given lock variable from any locks.
  - It is illegal to call this routine with a lock variable that is not initialized.
Synchronization: Lock routines

- **omp_set_lock()**
  - This subroutine forces the executing thread to wait until the specified lock is available. A thread is granted ownership of a lock when it becomes available.
  - It is illegal to call this routine with a lock variable that is not initialized.

- **omp_unset_lock()**
  - This subroutine releases the lock from the executing subroutine.
  - It is illegal to call this routine with a lock variable that is not initialized.
Synchronization: Lock routines

- **omp_test_lock()**
  - This subroutine attempts to set a lock, but does not block if the lock is unavailable.
  - For C/C++, non-zero is returned if the lock was set successfully, otherwise zero is returned.

- **Nested Locks**
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function.
  - **omp_init_nest_lock()**, **omp_set_nest_lock()**, **omp_unset_nest_lock()**, **omp_test_nest_lock()**, **omp_destroy_nest_lock()**
Synchronization: Lock routines

```c
omp_lock_t lck;
omp_init_lock(&lck);

#pragma omp parallel private(tmp, id)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
    omp_set_lock(&lck); // wait for your turn
    printf("%d %d", id, tmp);
    omp_unset_lock(&lck); // release the lock for next thread
}
omp_destroy_lock(&lck); // free up storage
```
The Parallel Region

- A parallel region is a block of code executed by multiple threads simultaneously.
- A parallel construct by itself creates an SPMD or “Single Program Multiple Data” program ... i.e., each thread redundantly executes the same code.

```c
#pragma omp parallel [clause[[,], clause] ...]
{
    "this is executed in parallel"
}
```
for Construct

- The loop worksharing Constructs
- 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 Construct

```c
#pragma omp parallel
{
    #pragma omp for
    //the variable i is made private
    for (i = 0; i < m; i++) {
        NEAT_STUFF(i);
    }
}
```
for Construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

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

```c
double res[MAX]; int i;
#pragma omp parallel for
{
    for (i = 0; i < MAX; i++)
    {
        res[i] = huge();
    }
}
```
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
Working with loops

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

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

- The master construct denotes a structured block that is only executed by the master thread.

- The other threads just skip it (no synchronization is implied).

```c
#pragma omp parallel
{
  do_many_things();
  #pragma omp master
  { exchange_boundaries(); }
  #pragma omp barrier
  do_many_other_things();
}
```
Single Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A **barrier** is implied at the end of the single block (can remove the barrier with a nowait clause).
- Syntax:

```c
#pragma omp parallel
{
    do_many_things();
    #pragma omp single
    { exchange_boundaries();}
    do_many_other_things();
}
```
Sections Construct

- Is a non-iterative work-sharing construct
- Gives a different structured block to each thread.
- It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent section directives are nested within a sections directive.
- Each section is executed once by a thread
- Different sections may be executed by different threads.
- It is possible that for a thread to execute more than one section.
There is an implied barrier at the end of a sections directive, unless the nowait clause is used.

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        X_calculation();
        #pragma omp section
        Y_calculation();
        #pragma omp section
        Z_calculation();
    }
}
```
The if clauses

- Only executes in parallel if expression evaluates to true
- Otherwise, executes serially

```c
#pragma omp parallel if (n > threshold)
{
    #pragma omp for
    for (i=0; i<n; i++)
        x[i] += y[i];
} /*-- End of parallel region --*/
```
Performance Improvement #1

- 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 #2

- 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
  \texttt{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 exponentially 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"`
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)

```c
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);
    }
    ...
}
```
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.
Using parallel construct

- 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

```c
#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);
    }
}
```
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.

```c
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

```c
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

May execute alpha, beta, and delta in parallel
Usage of Sections Construct

- Precedes a block of $k$ blocks of code that may be executed concurrently by $k$ threads
- 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
Example of parallel sections

```c
#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.
#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));
Final Example

- Monte Carlo Calculations
  - Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
  - Computing π with a digital dart board

- Throw darts at the circle/square.
  - Chance of falling in circle is proportional to ratio of areas:
    
    \[
    \begin{align*}
    A_c &= r^2 \cdot \pi \\
    A_s &= (2r)(2r) = 4r^2 \\
    P &= A_c / A_s = \pi / 4
    \end{align*}
    \]

  - Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.
static long num_trials = 10000;

int main()
{
    long i; long Ncirc = 0; double pi, x, y;
    double r = 1.0; // radius of circle.

    for(i=0; i<num_trials; i++) {
        x = random(); y = random();
        if ( x*x + y*y) <= r*r) Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```c
#include "omp.h"

static long num_trials = 10000;
int main()
{
    long i; long Ncirc = 0; double pi, x, y;
    double r = 1.0; // radius of circle.
    #pragma omp parallel for private (x, y)\
        reduction (+:Ncirc)
    for(i=0; i<num_trials; i++)
    {
        x = random(); y = random();
        if ( x*x + y*y) <= r*r) Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```
Threadprivate

- Makes global data private to a thread
- File scope and static variables, static class members
- Different from making them PRIVATE
  - with PRIVATE global variables are masked
  - THREADPRIVATE preserves global scope within each thread

```c
#include "omp.h"

int counter = 0;
#pragma omp 
threadprivate(counter)

int increment_counter(){
    counter++;
    return counter;
}
```
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
  - Conditionally parallelizing loops
  - Changing loop scheduling
### Summary (3/3)

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>OpenMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suitable for multiprocessors</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Suitable for multicomputers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports incremental parallelization</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Minimal extra code</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Explicit control of memory hierarchy</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
QUESTIONS?