Table of Contents

1 Misc Information
2 Introduction to Shared Memory Programming with OpenMP
   - OpenMP Overview
   - Compiling and Running OpenMP Code: Hello World
   - OpenMP: The PRAGMA Directive
   - Forking and Joining Threads
   - Binding OpenMP Thread to a Processor
   - OpenMP Thread to Processor Bindings
   - Trapezoidal Rule with OpenMP
   - Variable Scope
   - Reduction Clause
3 Next Time
HW5 Due Today.

New Topic: Shared memory programming with Open MP
  Pacheco, CH 5.1-5.4
What is OpenMP?

- OpenMP = Open MultiProcessing
- an API that supports multi-platform shared memory multiprocessing programming.
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPUs, all of which have access to main memory
- Applications built using hybrid model of parallel programming:
  - Runs on a computer cluster using both OpenMP and Message Passing Interface (MPI)
  - OR through the use of OpenMP extensions for non-shared memory systems.
- See:
  - http://openmp.org/
What is OpenMP?

- OpenMP grew out of the need to standardize different vendor specific directives related to parallelism.
- Pthreads not scaleable to large systems and does not support incremental parallelism very well.
- Correlates with evolution of hybrid architectures: shared memory and multi PE architectures being developed in early '90s.
- Structured around parallel loops and was meant to handle dense numerical applications.

Source: https://computing.llnl.gov/tutorials/openMP
OpenMP is an implementation of *multithreading*

- Method of parallelizing where a master thread forks a specified number of slave threads
- Tasks are divided among them.
- Threads run concurrently.

Source: http://en.wikipedia.org/wiki/OpenMP
Shared memory architecture 1
Non Uniform Memory Access (NUMA)

- Hierarchical Scheme: processors are grouped by physical location located on separate multi-core (PE) CPU packages or nodes.
- Processors (PEs) within a node share access to memory modules via UMA shared memory architecture.
- PE’s may also access memory from the remote node using a shared interconnect.

Source: https://software.intel.com/en-us/articles/optimizing-applications-for-numa
OpenMP Overview

OpenMP Features & Advantages

- Portable, threaded, shared-memory programming specification with light syntax
- Exact behavior depends on OpenMP implementation!
- Requires compiler support (C or Fortran)
- Allows programmer to define and separate serial and parallel regions
- Does not "detect" parallel dependencies or guarantee speedup
- Can use OpenMP to parallelize many serial for loops with only small changes to the source code.
- Task parallelism.
- Explicit thread synchronization.
- Standard problems in shared-memory programming
OpenMP Challenges

- Currently only runs efficiently in shared-memory multiprocessor platforms
- Scalability is limited by memory architecture.
- Cache memories
- Dealing with serial libraries
- Thread safety
- Unreliable error handling.
- Mostly used for loop parallelization
- Requires a compiler that supports OpenMP
- Lacks fine-grained mechanisms to control thread-processor mapping.
- Synchronization between subsets of threads is not allowed.
- Can be difficult to debug, due to implicit communication between threads via shared variables.
OpenMP: General Code Structure

```c
#include <omp.h>
main () {  
    int var1, var2, var3;
    Serial code
    ...
    /* Beginning of parallel section.
    Fork a team of threads. Specify variable scoping*/
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        /* Parallel section executed by all threads */
        ...
        /* All threads join master thread and disband*/
    }
    Resume serial code
    ...
}
```
OpenMP: Data Model

- Private and shared variables
- Global data space: accessed by all parallel threads.
- Private space: only be accessed by the thread.
- Parallel for loop index private by default.

```c
#pragma omp parallel for private(
                     privIndx, privDbl )
    for ( i = 0; i < arraySize; i++){
        for(privdx=0; privdx <16;privdx++){
            privDb1= ( (double)privdx)/16;
            y[i]=sin(exp(cos( -exp(sin(x[i])))))
            + cos( privDbl );
        }
    }
```
# OpenMP: Hello World

/* File: omp_hello.c
* Purpose: A parallel hello, world program that uses OpenMP
* Compile: gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
* Run: ./omp_hello <number of threads>
* Input: none
* Output: A message from each thread
* IPP: Section 5.1 (pp. 211 and ff.)
*/

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

/*--------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

/*-------------------------------------------------------------------*/

/* Function: Hello
* Purpose: Thread function that prints message
*/
void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}

} /* Hello */
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

possible outcomes
Running OpenMP Hello World

[mthomas]$
[mthomas@tuckoo]$ mpicc -g -Wall -fopenmp -o omp_hello omp_hello.c

[mthomas@tuckoo ch5]$ ./omp_hello 10
Hello from thread 6 of 10
Hello from thread 4 of 10
Hello from thread 5 of 10
Hello from thread 0 of 10
Hello from thread 1 of 10
Hello from thread 7 of 10
Hello from thread 2 of 10
Hello from thread 3 of 10
Hello from thread 9 of 10
Hello from thread 8 of 10
What to do if compiler does not support OpenMP

```c
#include <omp.h>

#ifdef _OPEN_MP
#include <omp.h>
#endif

int rank;
int thd_cnt;

#ifdef _OPEN_MP
rank=omp_get_thread_num();
thd_cnt=omp_get_num_threads();
#else
rank=0;
thd_cnt=1;
#endif
```
**OpenMP Directive: `#pragma`**

```c
#pragma omp parallel num_threads(thread_count)
Hello();
```

- `#pragma` is first OpenMP directive.
- Scope of a directive is one block of statements `{...}`
- OpenMP determines `#` threads to create, synchronize, destroy
- Start threads running thread function `Hello`
- `num_threads(thread_count)` is an OpenMP clause
- Similar (but less work) to the Pthread command:
  ```c
  pthread_create(&thread_handles[i], NULL, Thread_work, (void*) i);
  ```
- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C specification.
- Portable: compilers that don't support the pragmas ignore them.
OpenMP: Parallel Region Construct

- Defines a block of code to be executed by the threads:
  
  ```
  # pragma omp parallel num_threads(thread_count)
  {
    ...
  } (implied barrier)
  ```

- Example clauses:
  - if (expression): only in parallel if expression evaluates to true
  - private(list): everything private and local (no relation with variables outside the block).
  - shared(list): data accessed by all threads
  - default (none — shared)
  - reduction (operator: list)
  - firstprivate(list), lastprivate(list)
## OpenMP pragma directives

<table>
<thead>
<tr>
<th>#pragma omp</th>
<th>#Desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic</td>
<td>Identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.</td>
</tr>
<tr>
<td>atomic</td>
<td>Identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.</td>
</tr>
<tr>
<td>parallel</td>
<td>Defines crit. block to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.</td>
</tr>
<tr>
<td>for</td>
<td>Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.</td>
</tr>
<tr>
<td>parallel for</td>
<td>Shortcut combination ofomp parallel and omp for pragma directives, used to define a parallel region containing a single for directive.</td>
</tr>
<tr>
<td>ordered</td>
<td>Work-sharing construct identifying a structured block of code that must be executed in sequential order.</td>
</tr>
<tr>
<td>section(s)</td>
<td>Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.</td>
</tr>
<tr>
<td>parallel sections</td>
<td>Shortcut combination ofomp parallel and omp sectionspragma directives, used to define a parallel region containing a single sections directive.</td>
</tr>
<tr>
<td>single</td>
<td>Work-sharing construct identifying section of code to be run by a single avail. thread.</td>
</tr>
<tr>
<td>master</td>
<td>Synchronization construct identifying a section of code that must be run only by the master thread.</td>
</tr>
<tr>
<td>critical</td>
<td>Synchronization construct identifying a statement block that must be executed by a single thread at a time.</td>
</tr>
<tr>
<td>barrier</td>
<td>Synchronizes all the threads in a parallel region.</td>
</tr>
<tr>
<td>flush</td>
<td>Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.</td>
</tr>
<tr>
<td>threadprivate</td>
<td>Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.</td>
</tr>
</tbody>
</table>
A process forking and joining two threads
Forking and Joining Threads
Some OpenMP Comments & Observations

- In OpenMP terminology, the collection of threads executing the parallel block, the original thread and the new threads, is called a team.
- The original thread is called the master.
- Additional threads are called slaves.
- The master starts p-1 new threads.
- Implicit barrier: formed after the hello thread – all threads must return to this point in the code.
- All threads share STDIO.
- There may be system-defined limitations on the number of threads that a program can start.
- OpenMP standard does not guarantee that the directive will actually start the number of requested threads.
- Modern systems can start hundreds or thousands of threads.
Binding OpenMP Thread to a Processor

[mthomas@tuckoo ch5] mpicc -g -Wall -fopenmp -o omp_hello omp_hello.c
[mthomas@tuckoo ch5]$ cat omp_hello.c

/* File: omp_hello.c

* Purpose: A parallel hello, world program that uses OpenMP
*
* Compile: gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
* Run: ./omp_hello <number of threads>
*
* Input: none
* Output: A message from each thread
*
* IPP: Section 5.1 (pp. 211 and ff.)
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/unistd.h>
#include <omp.h>
#include <sched.h>

void Hello(void); /* Thread function */

/*-------------------------------------------------------------*/
int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */
/*******************************************************
 * Function: Hello
 * Purpose: Thread function that prints message
 */
void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    int cpu, np;
    char hostname[1024];
    hostname[1023] = '\0';
    gethostname(hostname, 1023);
    cpu=sched_getcpu();
    np = omp_get_num_procs();

    printf("Hello from Thread[%d] of [%d] on host: %s, cpu=[%d] of np=[%d]\n",
            my_rank, thread_count,hostname,cpu,np);
}
} /* Hello */
Here is the batch script:

Notes: I asked for all the processors on the node, and varied the number of threads.

[mthomas@tuckoo ch5]$ cat batch.omphello
#!/bin/sh
# this example batch script requests omp_hello processors...
# for more info on requesting specific nodes see
# "man pbs_resources"
# Run with:
# qsub -v NTHDS=6 batch.omphello
# qsub -v NTHDS=64 batch.omphello
#PBS -V
#PBS -l nodes=1:ppn=6:node9
#PBS -N omphello
#PBS -j oe
#PBS -q batch

cd $PBS_O_WORKDIR

# npte: both of these lines do the same thing:
mpirun -np 1 -hostfile $PBS_NODEFILE ./omp_hello $NTHDS
### ./omp_hello $NTHDS
Here are some outputs:

[mthomas@tuckoo ch5]$ qsub -v NTHDS=3 batch.omphello
71360.tuckoo.sdsu.edu
[mthomas@tuckoo ch5]$ cat omphello.o71360 | sort

[mthomas@tuckoo ch5]$ qsub -v NTHDS=12 batch.omphello
71360.tuckoo.sdsu.edu
[mthomas@tuckoo ch5]$ cat omphello.o71360 | sort
The Trapezoid Rule for Numerical Integration

**Solve the Integral:** \( \int_a^b F(x) \, dx \)

**The Trapezoidal Rule**

Where \( F(x) \) can be any function of \( x \): \( f(x^2), f(x^3) \)

See Pacheco IPP (2011), Ch3.
Trapezoid Equations

Integral: \[ \int_{a}^{b} f(x) \, dx \]

Area of 1 trapezoid: \[ = \frac{h}{2} \left[ f(x_i) + f(x_{i+1}) \right] \]

Base: \[ h = \frac{b-a}{n} \]

Endpoints: \[ x_0 = a, \quad x_1 = a + h, \quad x_2 = a + 2h, \ldots, \quad x_{n-1} = a + (n-1)h, \quad x_c = b \]

Sum of Areas: \[ \text{Area} = h \left[ \frac{f(x_0)}{2} + f(x_{i+1}) + f(x_{i+1}) + \ldots + f(x_{n-1}) \frac{f(x_n)}{2} \right] \]
Trapezoid Problem: Serial Algorithm

/* Input: a ,b, n */
h = (b-a)/n;
approx = (F(a) + F(b))/2.0
for (i=0; i<= n-1; i++) {
    x_i = a + i*H;
    approx += f(x_i);
}
approx = h* approx
Parallelizing the Trapezoidal Rule

**PCAM Approach**
- Partition problem solution into tasks.
- Identify communication channels between tasks.
- Aggregate tasks into composite tasks.
- Map composite tasks to cores.
Two types of tasks:
Compute area of 1 trapezoid
Compute area sums
First OpenMP Version of the Trap Alg.

1 We identified two types of tasks:
   a computation of the areas of individual trapezoids, and
   b adding the areas of trapezoids.

2 There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.

3 We assumed that there would be many more trapezoids than cores. So we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).
<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>global_result = 0 to register</td>
<td>finish my_result</td>
</tr>
<tr>
<td>1</td>
<td>my_result = 1 to register</td>
<td>global_result = 0 to register</td>
</tr>
<tr>
<td>2</td>
<td>add my_result to global_result</td>
<td>my_result = 2 to register</td>
</tr>
<tr>
<td>3</td>
<td>store global_result = 1</td>
<td>add my_result to global_result</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>store global_result = 2</td>
</tr>
</tbody>
</table>

Unpredictable results when two (or more) threads attempt to simultaneously execute:

```c
global_result += my_result;
```

Results in a race condition
Mutual exclusion

```c
#pragma omp critical
global_result += my_result;
```

only one thread can execute
the following structured block at a time

critical directive tells compiler that system needs to provide
mutually exclusive access control for the block of code.
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double global_result = 0.0; /* Store result in global_result */
    double a, b; /* Left and right endpoints */
    int n; /* Total number of trapezoids */
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    #pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
    return 0;
} /* main */
```
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    # pragma omp critical
    *global_result_p += my_result;
}

/* Trap */
Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.
Scope in OpenMP

- A variable that can be accessed by all the threads in the team has **shared** scope.

- A variable that can only be accessed by a single thread has **private** scope.

- The default scope for variables declared before a parallel block is **shared**.
for C, variables defined in `main` have global; variables defined in a `function` have function scope.

for OpenMP: the scope of a variable is associated with the set of threads that can access the variable in a parallel block.

- **shared scope:**
  - the default scope for variables defined outside a parallel block
  - e.g. `global_results` was declared in `main`, so it is shared by all threads

- **private scope:**
  - a variable that can only be accessed by a single thread
  - The default scope for variables declared inside a parallel block is private (e.g. all vars in defined in Trap).
int main(int argc, char* argv[]) {
    /* Store result in global_result */
    double global_result = 0.0;
    /* Left and right endpoints */
    double a, b;
    int n; /* Total number of trapezoids*/
    int thread_count;
    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    if (n % thread_count != 0) Usage(argv[0]);
    # pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", a, b, global_result);
    return 0;
} /* main */

* Function: Trap
* Purpose: Use trapezoidal rule to estimate definite integral
* Input args:
* a: left endpoint
* b: right endpoint
* n: number of trapezoids
* global_result_p: pointer to global trap sum
* Output arg:
* integral: estimate of integral from a to b of f(x)
*/

void Trap(double a, double b, int n,
           double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    # pragma omp critical
    *global_result_p += my_result;
} /* Trap */
OpenMP: Reduction Clause

We need this more complex version to add each thread’s local calculation to get `global_result`.

```c
void Trap(double a, double b, int n, double* global_result_p);
```

Although we’d prefer this.

```c
double Trap(double a, double b, int n);
```

```
global_result = Trap(a, b, n);
```
If we use this, there’s no critical section!

```c
double Local_trap(double a, double b, int n);
```

If we fix it like this...

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
#pragma omp critical
    global_result += Local_trap(double a, double b, int n);
}
```

...we force the threads to execute sequentially.

**Local Trap does not have reference to the global variable global_result**
We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0; /**< private */
    my_result += Local_trap(double a, double b, int n);
#pragma omp critical
    global_result += my_result;
}
```

Notes: the call to Local_Trap is inside the parallel block, but outside critical section; my_result is private to each thread
Reduction operators

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.
A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)

+ , *, -, &, |, ^, &&, ||
```

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
  reduction(+: global_result)
  global_result += Local_trap(double a, double b, int n);
```
A few comments

- OpenMP (1) creates private thread variable, (2) stores result for thread, and (3) creates critical section block.

- Subtraction ops are not guaranteed (not associative or commutative):

  \[
  \text{result} = 0; \\
  \text{for} \ (i = 1; \ i \leq 4; \ i++) \\
  \text{result} -= i;
  \]

- Floating point arithmetic is not associative, so results are not guaranteed:

  \[ a + (b + c) \text{ may not equal } (a + b) + c \]
Next class: 04/14/15: Continue OpenMP
Reach Pacheco, IPP, Ch. 5.
HW #6 (Pthreads & Open MP): Due 04/23/15