Parallel Computing with OpenMP
Work Sharing
April 26, 2012

“In theory, there is no difference between theory and practice. In practice there is.”
Yogi Berra
Before We Get Started…

- Last lecture
  - Wrap up, MPI Derived Types (Handling complex data)
  - Start OpenMP

- Today
  - Work sharing under OpenMP

- Other issues
  - Assignment 12 due Sunday, April 29 at 11:59 pm
    - One problem, compute integral using OpenMP
  - Doodle pool available soon - select time slot for your Final Project presentation
  - Hope to do a first pass through your Midterm Projects this weekend
Work Plan

● What is OpenMP?
  Parallel regions
  Work sharing – Parallel Sections
  Data environment
  Synchronization

● Advanced topics
Function Level Parallelism

a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n", bigboss(s,c));
omp sections

- `#pragma omp sections`
- Must be inside a parallel region
- Precedes a code block containing \(N\) sub-blocks of code that may be executed concurrently by \(N\) threads
- Encompasses each omp section

- `#pragma omp section`
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads
#pragma omp parallel sections
{
    #pragma omp section
        double a = alice();
    #pragma omp section
        double b = bob();
    #pragma omp section
        double c = cy();
}

double s = boss(a, b);
printf ("%6.2f\n", bigboss(s,c));
Advantage of Parallel Sections

- Independent sections of code can execute concurrently – reduce execution time

```c
#pragma omp parallel sections
{
#pragma omp section
    phase1();
#pragma omp section
    phase2();
#pragma omp section
    phase3();
}
```
```c
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 2 procs\n");
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
```
sections, Example: 2 threads
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 4 procs\n");
    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 6.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
sections, Example: 4 threads
Work Plan

● What is OpenMP?
  Parallel regions
  Work sharing – Tasks
  Data environment
  Synchronization

● Advanced topics
OpenMP Tasks

- **Task** – Most important feature added in latest 3.0 version of OpenMP

- Allows parallelization of irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer
Tasks: What Are They?

- Tasks are independent units of work
- A thread is assigned to perform a task
- Tasks might be executed immediately or might be deferred
  - The runtime system decides which of the above
- Tasks are composed of
  - code to execute
  - data environment
  - internal control variables (ICV)
Tasks: What Are They?

[More specifics…]

- **Code to execute**
  - The literal code in your program enclosed by the task directive

- **Data environment**
  - The shared & private data manipulated by the task

- **Internal control variables**
  - Thread scheduling and environment variable type controls

- A task is a specific instance of executable code and its data environment, generated when a thread encounters a `task` construct

- **Two activities: packaging and execution**
  - A thread packages new instances of a task (code and data)
  - Some thread in the team executes the task at some later time
using namespace std;

typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itrtr) {
    *itrtr *= 2.
}

int main() {
    LISTDBL test; // default constructor
    LISTDBL::iterator it;

    for( int i=0; i<4; ++i )
        for( int j=0; j<8; ++j ) test.insert(test.end(), pow(10.0, i+1)+j);
    for( it = test.begin(); it != test.end(); it++ ) cout << *it << endl;

    it = test.begin();
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single private(it)
        {
            while( it != test.end() ) {
                #pragma omp task
                {
                    doSomething(it);
                }
                it++;
            }
        }
    }
    for( it = test.begin(); it != test.end(); it++ ) cout << *it << endl;
    return 0;
}
#include <omp.h>
#include <list>
#include <iostream>
#include <math.h>

using namespace std;

typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itr){
    *itr = -2.0;
}

int main(){
    LISTDBL test;          // default constructor
    LISTDBL::iterator it;
    for( int i=0;i<4;++i)
        for( int j=0;j<8;++j)
            test.insert(test.end(), pow(10.0,i+1)+j);
    it = test.begin();
    #pragma omp parallel num_threads(8)
    #pragma omp single private(it)
    {
        while( it != test.end() ) {
            #pragma omp task
            {
                doSomething(it);
                it++;
            }
        }
    }
    for( it = test.begin();it != test.end();it++) cout << *it << endl;
    return 0;
}
Task Construct – Explicit Task View

- A team of threads is created at the `omp parallel` construct
- A single thread is chosen to execute the while loop – call this thread “L”
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the `omp task` construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region’s construct
- Each task has its own stack space that will be destroyed when the task is completed

```c
#pragma omp parallel
{
    #pragma omp single
    {
        // block 1
        node *p = head_of_list;
        while (p) {
            // block 2
            #pragma omp task private(p)
            process(p);
            p = p->next;  // block 3
        }
    }
}
```
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
{
    #pragma omp single
    {
        // block 1
        node *p = head_of_list;
        while (p) {
            // block 2
            #pragma omp task private(p)
            process(p);
            p = p->next;
            // block 3
        }
    }
}
```
Tasks: Synchronization Issues

- **Setup:**
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guarantee completion of Task A before invoking the execution of Task B

- **Tasks are guaranteed to be complete at thread or task barriers:**
  - At the directive: `#pragma omp barrier`
  - At the directive: `#pragma omp taskwait`
Task Completion Example

```c
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple foo tasks created here – one for each thread
- All foo tasks guaranteed to be completed here
- One bar task created here
- bar task guaranteed to be completed here
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
- Data scoping
- Synchronization
- Advanced topics
Data Scoping – What’s shared

- OpenMP uses a shared-memory programming model

- **Shared variable** - a variable that can be read or written by multiple threads

- Shared clause can be used to make items explicitly shared
  - Global variables are shared by default among tasks
  - Other examples of variables being shared among threads
    - File scope variables
    - Namespace scope variables
    - Variables with const-qualified type having no mutable member
    - Static variables which are declared in a scope inside the construct
Data Scoping – What’s Private

- Not everything is shared...

  - Examples of implicitly determined PRIVATE variables:
    - Stack (local) variables in functions called from parallel regions
    - Automatic variables within a statement block
    - Loop iteration variables
    - Implicitly declared private variables within tasks will be treated as firstprivate

- **firstprivate**
  - Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct
Data Scoping – The Golden Rule

- When in doubt, explicitly indicate who’s what
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);

    #pragma omp sections nowait
    {
        #pragma omp section
        {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++)
            {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
        }
        #pragma omp section
        {
            printf("Thread %d doing section 2\n",tid);
            for (i=0; i<N; i++)
            {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
            }
        }
    } /* end of sections */
    printf("Thread %d done.\n",tid);
} /* end of parallel section */
A Data Environment Example

float A[10];
main () {
    int index[10];
    #pragma omp parallel
    {
        Work (index);
    }
    printf ("%d\n", index[1]);
}

extern float A[10];
void Work (int *index)
{
    float temp[10];
    static integer count;
    <...>
}

A, index, and count are shared by all threads, but temp is local to each thread

Includes material from IOMPP
Data Scoping Issue:

**fib Example**

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

**Values of the private variables not available outside of tasks**

- n is private in both tasks
- x is a private variable
- y is a private variable
- This is very important here
- What’s wrong here?

Credit: IOMPP
### Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    {
        x = fib(n-1);
    }
    #pragma omp task
    {
        y = fib(n-2);
    }
    #pragma omp taskwait
    return x+y
}
```

**Values of the private variables not available outside of tasks**

- x is a private variable
- y is a private variable

Credit: IOMPP
Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait

    return x+y;
}
```

- `n` is private in both tasks
- `x` & `y` are now shared
- We need both values to compute the sum

The values of the `x` & `y` variables will be available outside each task construct – after the taskwait.
Work Plan

What is OpenMP?
- Parallel regions
- Work sharing
- Data environment

Synchronization

● Advanced topics
Implicit Barriers

- Several OpenMP constructs have implicit barriers
  - `parallel` – necessary barrier – cannot be removed
  - `for`
  - `single`

- Unnecessary barriers hurt performance and can be removed with the `nowait` clause
  - The `nowait` clause is applicable to:
    - `for` clause
    - `single` clause

Credit: IOMPP
Nowait Clause

- Use when threads unnecessarily wait between independent computations

```c
#pragma omp for nowait
for(...)
{
...;
```
```
#pragma omp single nowait
{
[...]  
```
```
#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
    a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```
Barrier Construct

- Explicit barrier synchronization
- Each thread waits until all threads arrive

```c
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A, B); // Processed A into B
#pragma omp barrier

    DoSomeWork(B, C); // Processed B into C
}
```

Credit: IOMPP
Atomic Construct

- Applies only to simple update of memory location
- Special case of a critical section, to be discussed shortly
  - Atomic introduces less overhead than critical

```c
#pragma omp parallel for shared(x, y, index, n)
  for (i = 0; i < n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
  }
```

index[0] = 2;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 5;
index[7] = 1;
Example: Dot Product

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

What is Wrong?
Race Condition

- A *race condition* is nondeterministic behavior produced when two or more threads access a shared variable at the same time.

- For example, suppose both Thread A and Thread B are executing the statement:
  
  ```
  area += 4.0 / (1.0 + x*x);
  ```
Two Possible Scenarios

Order of thread execution causes non-determinant behavior in a data race

Credit: IOMPP
Protect Shared Data

- The **critical** construct: protects access to shared, modifiable data
- The critical section allows only one thread to enter it at a given time

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Credit: IOMPP
OpenMP Critical Construct

```
#pragma omp critical [(lock_name)]

- Defines a critical region on a structured block
```

Threads wait their turn – only one at a time calls `consum()` thereby protecting `RES` from race conditions.

Naming the critical construct `RES_lock` is optional but highly recommended.

```
float RES;
#pragma omp parallel
{
#pragma omp for
    for(int i=0; i<niters; i++){
        float B = big_job(i);
        #pragma omp critical (RES_lock)
            consum(B, RES);
    }
}
```

Good Practice – Name all critical sections

Includes material from IOMPP
OpenMP Reduction Clause

reduction (op : list)

- The variables in “list” must be shared in the enclosing parallel region

- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable

Credit: IOMPP
Reduction Example

```c
#pragma omp parallel for reduction(+:sum)
for(i=0; i<N; i++) {
    sum += a[i] * b[i];
}
```

- Local copy of `sum` for each thread
- All local copies of `sum` added together and stored in “global” variable
OpenMP Reduction Example: Numerical Integration

\[ \int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi \]

```c
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Credit: IOMPP
OpenMP Reduction Example: Numerical Integration

```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1.0/(double(num_steps));
    double sum;

    #pragma omp parallel for reduction(+:sum)
    {
        for(int i=0; i<num_steps; i++) {
            double x = (i + .5)*step;
            sum += 4.0/(1.+ x*x);
        }
    }

    double my_pi = sum*step;
    printf("Value of integral is: \%f\n", my_pi);

    return 0;
}
```
OpenMP Reduction Example:

Output

- gcc didn’t cut it for me...
  - Ended up using g++

```
[negrut@euler24 CodeBits]$ g++ testOMP.cpp -o me964.exe
[negrut@euler24 CodeBits]$ ./me964.exe 100000
Value of integral is: 3.141593
```
C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Credit: IOMPP