“A programming language is low level when its programs require attention to the irrelevant.”

-- Allan Perlis
Before We Get Started…

- Last time
  - Parallel computing on the CPU
  - Got started with OpenMP for parallel computing on multicore CPUs
- Today:
  - Continue OpenMP discussion:
    - sections
    - tasks
    - data scoping
    - synchronization
- Miscellaneous
  - Exam on Monday, November 25 at 7:15 PM.
    - Room 1163ME
    - Review session held during regular class hour that Monday
Function Level Parallelism

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

Alice, Bob, and Cy can be computed in parallel.
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`, see below

- `#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

[IOMPP]
#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 → reduces execution time

```c
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```

Time (Execution Flow)

Serial

Parallel

The pink and green tasks are executed at no additional time-penalty in the shadow of the purple task
#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

```
\'\\teddy\users\negrut\Academic\Classes\ME964\Spring2012\CodingSandBox\Simple\'
CMD.EXE was started with the above path as the current directory.
UNC paths are not supported. Defaulting to Windows directory.
Start with 2 procs
Start work 1
Start work 2
Finish work 1
Start work 3
Finish work 2
Finish work 3
Press any key to continue . . .
```
```
#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 the 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 OS & runtime decide 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**: (1) packaging, and (2) 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
    {
      while( it != test.end() ) {
        #pragma omp task private(it)
        {
          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>
void doSomething(LISTDBL::iterator & litr) {
    "litr" = 2;
}

int main() {
    LISTDBL test; // default constructor
    LISTDBL::iterator it;
    for (int i = 0; i < 10; i++) test.insert(test.end(), pow(i, 1.0 + i));
    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()) {
                doSomething(it);
                it++;
            }
        }
    }
    for (it = test.begin(); it != test.end(); it++) cout << "*it << endl;
    return 0;
}

Compile like:
$ g++ -o testOMP.exe testOMP.cpp

Initial values...

Final values...
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 runs 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
  - See an example in a bit

```c
#pragma omp parallel
// threads are ready to go now
{
    #pragma omp single
    {
        // block 1
        node *p = head_of_list;
        while (p!=listEnd) {
            // block 2
            #pragma omp task private(p)
            process(p);
            p = p->next;  // block 3
        }
    }
}
```

[IOMPP]→
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
// threads are ready to go now
{
    #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
        }
    }
}
```

How about synchronization issues?
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`
#pragma omp parallel
{
  #pragma omp task
  foo();
  #pragma omp barrier
  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
Comments: sections vs. tasks

- sections have a “static” attribute: things are mostly settled at compile time

- The tasks construct is more recent and more sophisticated
  - They have a “dynamic” attribute: things are figured out at run time and the construct counts under the hood on the presence of a scheduling agent
  - They can encapsulate any block of code
    - Can handle nested loops and scenarios when the number of jobs is not clear
  - The run time system generates and executes the tasks, either at implicit synchronization points in the program or under explicit control of the programmer

- NOTE: It’s the developer responsibility to ensure that different tasks can be executed concurrently
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 Basic Rule

- When in doubt, explicitly indicate who’s what
  - Data scoping: one of the most common sources of errors in OpenMP
#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]);
}

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

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

Assumed to be in another translation unit

A, index, count

temp temp temp

A, index, count

Includes material from IOMPP
Data Scoping Issue: fib Example

Assume that the parallel region exists outside of fib and that fib and the tasks inside it are in the dynamic extent of a parallel region.

```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;
}
```

- 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?

Values of the private variables not available outside of tasks

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
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

Credit: IOMPP
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
Nowait Clause

- Use when threads unnecessarily wait between independent computations

```
#pragma omp for nowait
for(...)
{
...;
}
```

```
#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);
```

Credit: IOMPP
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
index[0] = 2;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 5;
index[7] = 1;

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

Credit: IOMPP
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;
}
```

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

For example, suppose that `area` is shared and both Thread A and Thread B are executing the statement:

\[
\text{area} += 4.0 / (1.0 + \text{x}^2);
\]
Two Possible Scenarios

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

Value of area

Thread A | Thread B
---|---
11.667 | +3.765
15.432 | 15.432
18.995 | +3.563

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