Lecture 10: Advanced Topics
Software
Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Now it’s up to the programmers

• Adding more processors doesn’t help much if programmers aren’t aware of them...

• ... or don’t know how to use them.

• Serial programs don’t benefit from this approach (in most cases).
The Need for Parallel Programming

Parallel computing: using multiple processors in parallel to solve problems more quickly than with a single processor.

Examples of parallel machines:
- A cluster computer that contains multiple PCs combined together with a high speed network.
- A shared memory multiprocessor (SMP) by connecting multiple processors to a single memory system.
- A Chip Multi-Processor (i.e. multicore) (CMP) contains multiple processors (called cores) on a single chip.
Attempts to Make Multicore Programming Easy

• 1st idea: The right computer language would make parallel programming straightforward
  – Result so far: Some languages made parallel programming easier, but none has made it as fast, efficient, and flexible as traditional sequential programming.
Attempts to Make Multicore Programming Easy

• 2\textsuperscript{nd} idea: If you just design the hardware properly, parallel programming would become easy.
  – Result so far: no one has yet succeeded!
Attempts to Make Multicore Programming Easy

- 3rd idea: Write software that will automatically parallelize existing sequential programs.
  - Result so far: Success here is inversely proportional to the number of cores!
Parallelizing a sequential program is not very easy!

- It is not about parallelizing every step of the sequential program.
- Maybe we need a totally new algorithm.
- Our parallelization strategy also depends on the software!
Example

• Compute n values and add them together.

• Serial solution:

```c
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(...);
    sum += x;
}
```
Example (cont.)

- We have p cores, p much smaller than n.
- Each core performs a partial sum of approximately n/p values.

```c
my_sum = 0;
my_first_i = ... ;
my_last_i = ... ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
    my_x = Compute_next_value(...) ;
    my_sum += my_x;
}
```

Each core uses its own private variables and executes this block of code independently of the other cores.
Example (cont.)

• Once all the cores are done computing their private `my_sum`, they form a global sum by sending results to a designated “master” core which adds the final result.
if (I'm the master core) {
    sum = my_x;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_x to the master;
}
But wait!

There’s a much better way to compute the global sum.
Better parallel algorithm

• Don’t make the master core do all the work.
• Share it among the other cores.
• Pair the cores so that core 0 adds its result with core 1’s result.
• Core 2 adds its result with core 3’s result, etc.
• Work with odd and even numbered pairs of cores.
Better parallel algorithm (cont.)

• Repeat the process now with only the evenly ranked cores.
  • Core 0 adds result from core 2.
  • Core 4 adds the result from core 6, etc.

• Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.
Multiple cores forming a global sum
Analysis

• In the first example, the master core performs 7 receives and 7 additions.

• In the second example, the master core performs 3 receives and 3 additions.

• The improvement is more than a factor of 2!
Analysis (cont.)

• The difference is more dramatic with a larger number of cores.

• If we have 1000 cores:
  – The first example would require the master to perform 999 receives and 999 additions.
  – The second example would only require 10 receives and 10 additions.

• That’s an improvement of almost a factor of 100!
Two Ways Of Thinking ... And one Strategy!

- **Strategy:** Partitioning!
- **Two ways of thinking:**
  - Task-parallelism
  - Data-parallelism
- **Some constraints:**
  - communication
  - load balancing
  - synchronization
Cost and Challenges of Parallel Execution

• Communication cost
• Synchronization cost
• Not all problems are amenable to parallelization
• Hard to think in parallel
• Hard to debug
The Multicore Software Triad

- Application Performance
- Software Reliability
- Development Time
Example 1 of Parallel Programming:

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

int main() {

    // Do this part in parallel

    printf( "Hello, World!\n" );

    return 0;
}

Small and Easy Motivation

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

int main() {
    omp_set_num_threads(16);

    // Do this part in parallel
    #pragma omp parallel
    {
        printf( "Hello, World!\n" );
    }

    return 0;
}
OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence.

OpenMP is a small API that hides cumbersome threading calls with simpler directives.
OpenMP

• An **API** for shared-memory parallel programming.

• Designed for systems in which each **thread** can potentially have access to all available memory.

• System is viewed as a collection of cores or CPU's, all of which have access to main memory → shared memory architecture
A shared memory system
Pragmas

• Special preprocessor instructions.
• Typically added to a system to allow behaviors that aren’t part of the basic C specification.
• Compilers that don’t support the pragmas ignore them.

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

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

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

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

    return 0;
} /* main */

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

running with 4 threads

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

possible outcomes
OpenMp pragmas

• `# pragma omp parallel`

– Most basic parallel directive.
– The number of threads that run the following structured block of code is determined by the run-time system.
A process forking and joining two threads
Some terminology

• In OpenMP parlance 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, and the additional threads are called slaves.
The trapezoidal rule
Serial algorithm

/* Input: a, b, n */
h = (b–a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n–1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
A First OpenMP Version

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.
A First OpenMP Version

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.
Assignment of trapezoids to threads

Thread 0

Thread 1

Thread 2

Thread 3
Unpredictable results when two (or more) threads attempt to simultaneously execute:

```c
global_result += my_result;
```
Mutual exclusion

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

only one thread can execute
the following structured block at
a time
```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 */
Another Example

• **Problem:** Count the number of times each ASCII character occurs on a page of text.

• **Input:** ASCII text stored as an array of characters.

• **Output:** A histogram with 128 buckets – one for each ASCII character

**source:** http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Another Example

Sequential Version

```
1: void compute_histogram_st(char *page, int page_size, int *histogram){
2:     for(int i = 0; i < page_size; i++){
3:         char read_character = page[i];
4:         histogram[read_character]++;
5:     }
6: }
```

Speed on Quad Core: 10.36 seconds

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Another Example

We need to parallelize this.

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Another Example

1: void compute_histogram_st(char *page, int page_size, int *histogram){
2:     #pragma omp parallel for
3:     for(int i = 0; i < page_size; i++){
4:         char read_character = page[i];
5:         histogram[read_character]++;
6:     }

The above code does not work!! Why?

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Another Example

1: void compute_histogram_mt2(char *page, int page_size, int *histogram) {
2:    #pragma omp parallel for
3:    for (int i = 0; i < page_size; i++) {
4:        char read_character = page[i];
5:        #pragma omp atomic
6:        histogram[read_character]++;
7:    }
8: }

Speed on Quad Core:
114.89 seconds
> 10x slower than the single thread version!!

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Another Example

1: void compute_histogram_mt3(char *page,
   int page_size,
   int *histogram, int num_buckets){

2: 
   #pragma omp parallel
3: {
4:     int local_histogram[111][num_buckets];
5:     int tid = omp_get_thread_num();
6:     #pragma omp for nowait
7:       for(int i = 0; i < page_size; i++){
8:           char read_character = page[i];
9:           local_histogram[tid][read_character]++;
10:       }
11:     for(int i = 0; i < num_buckets; i++){
12:         #pragma omp atomic
13:           histogram[i] += local_histogram[tid][i];
14:     }
15: }
16: }

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html

Runs in 3.8 secs
Why speedup is not 4 yet?
void compute_histogram_mt4(char *page, int page_size,  
    int *histogram, int num_buckets){
1:       int num_threads = omp_get_max_threads();
2:       #pragma omp parallel
3:       {
4:           __declspec (align(64)) int local_histogram[num_threads+1][num_buckets];
5:           int tid = omp_get_thread_num();
6:           #pragma omp for
7:           for(int i = 0; i < page_size; i++){
8:               char read_character = page[i];
9:               local_histogram[tid][read_character]++;
10:          }
11:       #pragma omp barrier
12:       #pragma omp single
13:       for(int t = 0; t < num_threads; t++){
14:           for(int i = 0; i < num_buckets; i++)
15:               histogram[i] += local_histogram[t][i];
16:       }
17:   }

Speed is 4.42 seconds. Slower than the previous version.
void compute_histogram_mt4(char *page, int page_size,
    int *histogram, int num_buckets){
    int num_threads = omp_get_max_threads();
    #pragma omp parallel
    {
        __declspec (align(64)) int local_histogram[num_threads+1][num_buckets];
        int tid = omp_get_thread_num();
        #pragma omp for
        for(int i = 0; i < page_size; i++){
            char read_character = page[i];
            local_histogram[tid][read_character]++;
        }
        #pragma omp for
        for(int i = 0; i < num_buckets; i++){
            for(int t = 0; t < num_threads; t++)
                histogram[i] += local_histogram[t][i];
        }
    }
    Speed is 3.60 seconds.
}
What Can We Learn from the Previous Example?

• Atomic operations
  – They are expensive
  – Yet, they are fundamental building blocks.

• Synchronization:
  – correctness vs performance loss
  – Rich interaction of hardware-software tradeoffs
  – Must evaluate hardware primitives and software algorithms together
OpenMP Parallel Programming

1. Start with a parallelizable algorithm
   • loop-level parallelism is necessary
2. Implement serially
3. Test and Debug
4. Annotate the code with parallelization (and synchronization) directives
   • Hope for linear speedup
5. Test and Debug
All OpenMP programs begin with a single thread: **master thread** (ID = 0)

**FORK:** the master thread then creates a team of parallel **threads**.

**JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate.

OpenMP uses the fork-join model of parallel execution.
Example 2 of Parallel Programming:

MPI
We will talk about processes
MPI processes

• Identify processes by nonnegative integer ranks.

• $p$ processes are numbered $0, 1, 2, .. p-1$
Compilation

MPICC tool to create this executable file name (as opposed to default a.out) 

MPI is NOT a language. Just libraries called from C/C++, ...

Compile MPI source file:

```
mpicc -g -Wall -o mpi_hello mpi_hello.c
```

Wrapper script to compile

produce debugging information

turns on all warnings
Execution

mpiexec -n <number of processes> <executable>

miexec -n 1 ./mpi_hello

run with 1 process

mpiexec -n 4 ./mpi_hello

run with 4 processes
```c
#include <stdio.h>
#include <string.h>  /* For strlen */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
    char greeting[MAX_STRING];
    int comm_sz;  /* Number of processes */
    int my_rank;  /* My process rank */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!",
                my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
                 MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
                      0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```
Our first MPI program

```c
#include <stdio.h>
#include <string.h>  /* For strlen */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
    char greeting[MAX_STRING];
    int comm_sz;    /* Number of processes */
    int my_rank;    /* My process rank */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!\n", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```
Execution

mpiexec -n 1 ./mpi_hello
Greetings from process 0 of 1!

mpiexec -n 4 ./mpi_hello
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
Trends

• Toward more application specific languages
• The power consumption problem must be solved over all the stack (from algorithms to transistors).
• Still looking for an efficient parallel programming model
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
</table>
| Given a problem, how to write a program to solve it?                    | • Fundamental Algorithms (CSCI-GA.1170)  
• Software Engineering (CSCI-GA.2440)  
• Programming Languages (CSCI-GA.2110)  
• Logic In Computer Science  
• Heuristic Problem Solving                                                |
| What happens under the hood between you and the hardware?              | • Compiler construction (CSCI-GA.2130)  
• OS (CSCI-GA.2250)                                                                                                               |
| Domains you may be interested in                                         | • Bioinformatics  
• Machine learning  
• Graphics  
• Scientific computing  
• Web search engines  
• Financial computing  
• NLP  
• DB  
• Numerical methods  
• Artificial Intelligence                                                                                                       |
| Parallelism                                                              | • Multicore processors  
• GPUs  
• Cloud computing  
• Distributed systems                                                                                                           |
| The age of big data                                                     | • Big Data: Large Scale Machine Learning  
• Realtime and Big Data Analytics  
• Data Science                                                                                                                     |
| How are machines connected?                                             | • Networks and Distributed Systems                                                                                                 |
| The web                                                                 | • Social Networks  
• Building Responsive Websites  
• Applied Cryptography & Network Security                                                                                             |
Conclusions

• Due to technology constraints, we moved to multicore processors.
• Parallel programming is now a must → The free lunch is over!
• There are different flavors of parallel hardware that we will discuss and also many flavors of parallel programming languages that we will deal with.