Lecture 4: Gentle Introduction to Parallel Programming

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Models ... Models

Programmers

Programmer’s view

Cost model

- Programming Model
- Computational Model
- Architecture Model
- Machine Model

Interconnection
Mem hierarchy
Execution mode
...

Hardware Description
Let's Start With A Simple Example

• Compute n values and add them together.

• Serial solution:

```python
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.
Example (cont.)

```
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!
• 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!
Another Quick 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
Let's See A Quick Example

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: }

Sequential Version

Speed on Quad Core: 10.36 seconds

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
Let's See A Quick Example

We need to parallelize this.

[source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html]
Let’s See A Quick Example

The above code does not work!! Why?

```c
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:    }
```
Let’s See A Quick Example

```c
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
Let’s See A Quick Example

```c
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](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){
    int num_threads = omp_get_max_threads();  
    #pragma omp parallel
    {
    #pragma omp for
    for(int i = 0; i < page_size; i++){
    char read_character = page[i];
    local_histogram[tid][read_character]++;
    }
    #pragma omp barrier
    #pragma omp single
    for(int t = 0; t < num_threads; t++)
    for(int i = 0; i < num_buckets; i++)
    histogram[i] += local_histogram[t][i];
    }
17: }

Speed is 4.42 seconds. Slower than the previous version.
Let's See A Quick Example

```c
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:      }
12:      #pragma omp for
13:      for(int i = 0; i < num_buckets; i++){
14:        for(int t = 0; t < num_threads; t++)
15:          histogram[i] += local_histogram[t][i];
16:      }
17: }
```

Speed is 3.60 seconds.
What Can We Learn from the Previous Examples?

• Parallel programming is not only about finding a lot of parallelism.

• Critical section and atomic operations
  – Race condition
  – Again: correctness vs performance loss

• Know your tools: language, compiler and hardware
What Can We Learn from the Previous Examples?

• 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
Sources of Performance Loss in Parallel Programs

- Extra overhead
  - load
  - synchronization
  - communication
- Bigger code than sequential version
- Contention due to hardware resources
- Coherence
- Load imbalance
- Artificial dependence
Artificial Dependencies

int result;  //Global variable
for (...) // The OUTER loop
    modify_result(...);
    if(result > threshold)
        break;

void modify_result(...)
    ...
result = ...
Coherence

- Extra bandwidth (scarce resource)
- Latency due to the protocol
- False sharing
Load Balancing

Time

- Synchronization
- Work
- Idle

- Synchronization
Load Balancing

• Assignment of work not data is the key
• If you cannot eliminate it, at least reduce it.
• Static assignment
  – Has its overhead
So …

• We have a serial program.

• How to parallelize it?

• We know that we need to divide work, ensure load balancing, manage synchronization, and reduce communication! \( \rightarrow \) Nice! How to do that?

• Unfortunately: there is no mechanical process.

• Ian Foster has some nice framework.
1. **Partitioning**: divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. 

Communication: determine what communication needs to be carried out among the tasks identified in the previous step.
Foster’s methodology

3. **Agglomeration or aggregation**: combine tasks and communications identified in the first step into larger tasks.

For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
Foster’s methodology

4. **Mapping**: assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.
Example - histogram

- 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9
Serial program - input

1. The number of measurements: \texttt{data\_count}
2. An array of \texttt{data\_count} floats: \texttt{data}
3. The minimum value for the bin containing the smallest values: \texttt{min\_meas}
4. The maximum value for the bin containing the largest values: \texttt{max\_meas}
5. The number of bins: \texttt{bin\_count}
• Data[0] = 1.3  
• Data[1] = 2.9  
• Data[2] = 0.4  
• Data[3] = 0.3  
• Data[4] = 1.3  
• Data[5] = 4.4  
• Data[6] = 1.7  
• Data[7] = 0.4  
• Data[8] = 3.2  
• Data[9] = 0.3  
• Data[10] = 4.9  
• Data[11] = 2.4  
• Data[12] = 3.1  
• Data[13] = 4.4  
• Data[14] = 3.9  
• Data[15] = 0.4  
• Data[16] = 4.2  
• Data[17] = 4.5  
• Data[18] = 4.9  
• Data[19] = 0.9

\[
data\text{\_count} = 20\]
Data[0] = 1.3
Data[1] = 2.9
Data[2] = 0.4
Data[3] = 0.3
Data[4] = 1.3
Data[5] = 4.4
Data[6] = 1.7
Data[7] = 0.4
Data[8] = 3.2
Data[9] = 0.3
Data[10] = 4.9
Data[11] = 2.4
Data[12] = 3.1
Data[13] = 4.4
Data[14] = 3.9,
Data[15] = 0.4
Data[16] = 4.2
Data[17] = 4.5
Data[18] = 4.9
Data[19] = 0.9

data_count = 20
min_meas = 0.3
max_meas = 4.9
bin_count = 5
Serial program - output

1. \textbf{bin\_maxes} : an array of \texttt{bin\_count} floats $\rightarrow$ store the upper bound of each bin

2. \textbf{bin\_counts} : an array of \texttt{bin\_count} ints $\rightarrow$ stores the number of elements in each bin
\[
\text{Data}[0] = 1.3 \\
\text{Data}[1] = 2.9 \\
\text{Data}[2] = 0.4 \\
\text{Data}[3] = 0.3 \\
\text{Data}[4] = 1.3 \\
\text{Data}[5] = 4.4 \\
\text{Data}[6] = 1.7 \\
\text{Data}[7] = 0.4 \\
\text{Data}[8] = 3.2 \\
\text{Data}[9] = 0.3 \\
\text{Data}[10] = 4.9 \\
\text{Data}[11] = 2.4 \\
\text{Data}[12] = 3.1 \\
\text{Data}[13] = 4.4 \\
\text{Data}[14] = 3.9, \\
\text{Data}[15] = 0.4 \\
\text{Data}[16] = 4.2 \\
\text{Data}[17] = 4.5 \\
\text{Data}[18] = 4.9 \\
\text{Data}[19] = 0.9
\]

\[
\text{bin\_maxes}[0] = 0.9 \\
\text{bin\_maxes}[1] = 1.7 \\
\text{bin\_maxes}[2] = 2.9 \\
\text{bin\_maxes}[3] = 3.9 \\
\text{bin\_maxes}[4] = 4.9
\]

\[
\text{bin\_counts}[0] = 6 \\
\text{bin\_counts}[1] = 3 \\
\text{bin\_counts}[2] = 2 \\
\text{bin\_counts}[3] = 3 \\
\text{bin\_counts}[4] = 6
\]
int bin = 0;
for( i = 0; i < data_count; i++){
    bin = find_bin(data[i], ...);
    bin_counts[bin]++;
}
First two stages of Foster's Methodology

Find_bin ...
\[ \text{data}[i-1] \quad \text{data}[i] \quad \text{data}[i+1] \]

Increment bin_counts ...
\[ \text{bin_counts}[b-1]++ \quad \text{bin_counts}[b]++ \]

Find_bin returns the bin that data[i] belongs to.
Alternative definition of tasks and communication

```
Find_bin ... data[i-1]  data[i]  data[i+1]  data[i+2] ...

• loc_bin_cts[b-1]++  loc_bin_cts[b]++

• loc_bin_cts[b-1]++  loc_bin_cts[b]++

• bin_counts[b-1]++  bin_counts[b]++  ...```

Copyright © 2010, Elsevier Inc.
All rights Reserved
Adding the local arrays
Patterns in Parallelism

- Task-level (e.g. Embarrassingly parallel)
- Divide and conquer
- Pipeline
- Iterations (loops)
- Client-server
- Geometric (usually domain dependent)
- Hybrid (different program phases)
Task Level

Independent Tasks

A  B  C  D  E

A
B
C
E
D
Client-Server/Repository

- Compute A
- Compute B
- Compute C
- Compute D
- Compute E

Asynchronous Function calls

Repository
Example

Assume we have a large array and we want to compute its minimum (T1), average (T2), and maximum (T3).

```
#define maxN 1000000000
int m[maxN];
int i;
int min = m[0];
int max = m[0];
double avrg = m[0];
for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
    avrg = avrg + m[i];
    if(m[i] > max)
        max = m[i];
}
avrg = avrg / maxN;
```
Divide-And-Conquer

problem

split

subproblem

Compute subproblem

subproblem

merge

subproblem

Compute subproblem

merge

subproblem

Compute subproblem

split

subproblem

Compute subproblem

merge

solution

split

subproblem

Compute subproblem

merge

subproblem

Compute subproblem
A series of **ordered** but **independent** computation stages need to be applied on data.
Pipeline

• Useful for
  – streaming workloads
  – Loops that are hard to parallelize
    • due inter-loop dependence
• Usage for loops: split each loop into stages so that multiple iterations run in parallel.
• Advantages
  – Expose intra-loop parallelism
  – Locality increases for variables uses across stages
• How shall we divide an iteration into stages?
  – number of stages
  – inter-loop vs intra-loop dependence
The Big Picture of Parallel Programming

Decomposition
- Task Decomposition
- Data Decomposition

Dependence Analysis
- Group Tasks
- Order Tasks
- Data Sharing

Design Evaluation

Source: David Kirk/NVIDIA and Wen-mei W. Hwu /UIUC
BUGS

- Sequential programming bugs + more
- Hard to find
- Even harder to resolve 😞
- Due to many reasons:
  - example: race condition
Example of Race Condition

1. Process A reads in
2. Process B reads in
3. Process B writes file name in slot 7
4. Process A writes file name in slot 7
5. Process A makes in = 8

In this context, process and thread can be used interchangeably.
How to Avoid Race Condition?

• Prohibit more than one process from reading and writing the shared data at the same time -> mutual exclusion

• The part of the program where the shared memory is accessed is called the critical region

source: http://www.futurechips.org/wp-content/uploads/2011/06/Screenshot20110618at12.11.05AM.png
Conditions of Good Solutions to Race Condition

1. No two threads may be simultaneously inside their critical region
2. No assumptions may be made about speeds or the number of CPUs/Cores
3. No thread running outside its critical region may block other processes
4. No thread has to wait forever to enter its critical region
Traditional Way of Parallelization

Strategy 1: Automatic Parallelization

Existing Source Code → Minor Code Modification → Automatic Parallelization → Parallel Application

Do We Have To Start With Sequential Code?

Strategy 3: Major Recoding

Existing Source Code → Major Recoding → Compiler Assisted Parallelization → Parallel Application
About Threads

• Thread vs Process
  – Process can consists of several threads
  – Each thread has its own stack

• Once created a thread can be in one of 4 states: ready, running, waiting (blocked), or terminated.

• User level threads vs kernel level threads
Multithreaded Programs

• Using established APIs at the application program
  – Example: Pthreads and OpenMP

  OpenMP:
  – developer-friendly
  – Requires compiler supporting OpenMP API

• Pthreads
  – More lower-level
  – More control and richer constructs

• Higher-level languages exist, but they tend to sacrifice performance to make program-development easier.
  – Example: Haskell
Conclusions

• Pick your programming model
• Task decomposition
• Data decomposition
• Refine based on:
  – What compiler can do
  – What runtime can do
  – What the hardware provides