Lecture 5: Overview of Parallel Programming

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

Programmers

1. Machine Model
2. Architecture Model
3. Computational Model
4. Programming Model

- Programmer’s view
- Cost model
- Interconnection
- Mem hierarchy
- Execution mode
- Hardware Description
Let’s See A 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: }

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

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

#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: }
```

Runs in 3.8 secs
Why speedup is not 4 yet?

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

```c
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 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];
        }
    }
    Speed is 4.42 seconds.
    Slower than the previous version.
}
```

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

```c
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(aligned(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];
        }
    }

    }  // end of parallel block

    // Speed is 3.60 seconds.

source: http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html
```
What Can We Learn from the Previous Example?

- 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 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
Sources of Performance Loss in Parallel Programs

- Extra overhead
  - load
  - synchronization
  - communication
- Artificial dependencies
  - Hard to find
  - May introduce more bugs
  - A lot of effort to get rid of
- Contention due to hardware resources
- Coherence
- Load imbalance
Artificial Dependencies

int result;
// Global variable

for (…) // The OUTER loop
    modify_result(…);
    if(result > threshold)
        break;

void modify_result(…)
    …
    …
    result = …

What is wrong with that program when we try to parallelize it?
Coherence

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

Time

- Synchronization
- Work
- Idle
Load Balancing

• Assignment of work not data is the key
• If you cannot eliminate it, at least reduce it.
• Static assignment
• Dynamic assignment
  – Has its overhead
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
Client-Server/Repository

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

Repository

Asynchronous Function calls
Example

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

```c
#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;
```

```c
int i; int min = m[0];
for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
}
```

```c
int j;
double avrg = m[0];
for(j=1; j < maxN; j++) {
    avrg = avrg + m[j];
    avrg = avrg / maxN;
}
```

```c
int k; int max = m[0];
for(k=1; k < maxN; k++) {
    if(m[k] > max)
        max = m[k];
}
```
Divide-And-And-Conquer

Diagram:
- Problem
  - Subproblem
    - Split
      - Compute subproblem
        - Merge
          - Subproblem
    - Split
      - Compute subproblem
        - Merge
          - Subproblem
  - Split
    - Compute subproblem
      - Merge
        - Subproblem
    - Compute subproblem
      - Merge
        - Subproblem
  - Merge
    - Solution
Pipeline

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

RACE CONDITION!!

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

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

Do We Have To Start With Sequential Code?

Strategy 3: Major Recoding
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