I-0 — Introduction

Objectives:
- Explain necessity of parallel/multithreaded algorithms.
- Describe different forms of parallel processing.
- Present commonly used architectures.
- Introduce a few basic terms.

Comments:
- Try to relate to the students’ environment:
  - Student laptop most likely has multi-core CPU.
  - Most gaming consoles, smartphones, tablets have at least a dual-core CPU.
  - Lab: machines could work together in a distributed manner.

Outcomes:
  2.1 Explain why learning about multithreaded algorithms is important.

I-1 — Introduction

Quote:
"The major cause [of the software crisis] is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them, it has created the problem of using its products."

E. Dijkstra, 1972 Turing Award Lecture

The Humble Programmer, 1972 Turing Award Lecture,
http://www.cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF

I-2 — Introduction

Moore’s Law: The number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years.

Source: http://en.wikipedia.org/wiki/Moore’s_law

M-0 — Multithreaded Algorithms

Objectives:
- Develop simple parallel algorithm out of a sequential one.
- Introduce the notion of barriers. Need to synchronize threads.
- Apply Amdahl’s Law.
- Use mutual exclusion to access shared resources (safety/critical sections/locks).
- Explain data races and dead locks / granularity and load balancing.

Comments:
- Skip M-3 – M-10 if you did Module 1.

Outcomes:
  2.2 Develop a parallel algorithm based on an easy to parallelize sequential algorithm.
  2.3 Determine the speedup of a parallel algorithm.
  2.4 Detect potential pitfalls of parallel algorithms.
**M-1 — Multithreaded Algorithms — Vector Operations**

**Vector operations:**
- Addition (subtraction) of two $n$-dimensional vectors $a$ and $b$.
  \[
  \begin{pmatrix}
  c_1 \\
  \vdots \\
  c_n
  \end{pmatrix}
  =
  \begin{pmatrix}
  a_1 \\
  \vdots \\
  a_n
  \end{pmatrix}
  +
  \begin{pmatrix}
  b_1 \\
  \vdots \\
  b_n
  \end{pmatrix}
  =
  \begin{pmatrix}
  a_1 + b_1 \\
  \vdots \\
  a_n + b_n
  \end{pmatrix}
  \]
- Scalar product of two $n$-dimensional vectors.
  \[
  c = \begin{pmatrix}
  a_1 \\
  \vdots \\
  a_n
  \end{pmatrix}
  \times
  \begin{pmatrix}
  b_1 \\
  \vdots \\
  b_n
  \end{pmatrix}
  = a_1 \cdot b_1 + \cdots + a_n \cdot b_n = \sum_{i=1}^{n} a_i \cdot b_i
  \]

**Question:** Can these sequential vector operations be transformed into parallel vector operations?

**Question:** What idea might help us to parallelize these vector operations?

---

**M-2 — Multithreaded Algorithms — Vector Addition**

**Vector operations:** parallel for $m$ threads.
- Addition (subtraction) of two $n$-dimensional vectors $a$ and $b$.
  \[
  \begin{pmatrix}
  c_1 \\
  \vdots \\
  c_n
  \end{pmatrix}
  =
  \begin{pmatrix}
  a_1 \\
  \vdots \\
  a_n
  \end{pmatrix}
  +
  \begin{pmatrix}
  b_1 \\
  \vdots \\
  b_n
  \end{pmatrix}
  =
  \begin{pmatrix}
  a_1 + b_1 \\
  \vdots \\
  a_n + b_n
  \end{pmatrix}
  \]
- We divide the vector into $m$ parts, $(a_{s_i+1}, \ldots, a_{s_i+1})^T$ and $(b_{s_i+1}, \ldots, b_{s_i+1})^T$ with $s_i = \left[\frac{(i-1)n}{m}\right]$ and $1 \leq i \leq m$.
- Add (subtract) the partial vectors in parallel.

**Algorithm (VECTOR-ADD(a,b))**

1. /* thread 1 */
2. for ($1 \leq i \leq \left\lfloor \frac{n}{m} \right\rfloor$) do
3. \[c_i = a_i + b_i\]
4. barrier
5. return $c$

**Question:** What idea might help us to parallelize these vector operations?

---

**M-3 — Multithreaded Algorithms**

**Synchronize threads:**
- We use a **barrier** to synchronize a given number of threads.
- A barrier that synchronizes $n$ threads is denoted with **barrier**$^n$.
- The instruction **barrier**$^n$.wait lets a thread wait until the given number of $n$ threads has reached the "waiting point".
- All threads continue as soon as the $n$-th thread reached this "waiting point".
- Different barriers are denoted by **barrier$_i$**.

**Performance analysis:**
- We divided the vector operations into 2 parts ⇒ $2 \times$ speedup?
- Our algorithm could have a sequential and a parallel part.
- Parallel parts might have different running times.
- Performance (speedup) depends on the parallel and the sequential part.

---

**M-8 — Multithreaded Algorithms — Amdahl’s Law**

**Example:** investigating potential maximum speedup.
- **Case 3:** Compare two programs each using 4 threads where one program has 50% parallel code, the other 66.6% parallel code.

<table>
<thead>
<tr>
<th>Sequential Time</th>
<th>Parallel Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>50% parallel code.</td>
<td>50 seconds (total)</td>
</tr>
<tr>
<td>66.6% parallel code.</td>
<td>30 seconds (total)</td>
</tr>
</tbody>
</table>
Example: investigating potential maximum speedup.

- **Maximum:** Compare two programs each using \( \infty \) threads where one program has 50%, the other 66.6% parallel code.

<table>
<thead>
<tr>
<th>50% parallel code.</th>
<th>66.6% parallel code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential (20 sec)</td>
<td>parallel (0 sec)</td>
</tr>
<tr>
<td>sequential (10 sec)</td>
<td>parallel (0 sec)</td>
</tr>
<tr>
<td>sequential (20 sec)</td>
<td>sequential (10 sec)</td>
</tr>
</tbody>
</table>

Running time:

<table>
<thead>
<tr>
<th>40 seconds (total)</th>
<th>20 seconds (total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Rightarrow ) maximum speedup of 2</td>
<td>( \Rightarrow ) maximum speedup of 3</td>
</tr>
</tbody>
</table>

Is the parallel algorithm correct?

Algorithm (SCALARPRODUCT\((a,b)\))

1. \( c = 0 \)
2. /* Compute in parallel */
3. /* Thread 1 */
4. for \( (1 \leq i \leq \lceil \frac{n}{m} \rceil) \) do
5. \( c = c + a_i \cdot b_i \)
6. barrier\(^m\).wait

Is the parallel algorithm correct?

Problem:

- The subproblems are not really independent of each other.
- The shared variable \( c \) is modified by all \( m \) threads in line (5).
- The read and write to \( c \) could be interleaved in an unfortunate order that most likely would lead to an incorrect scalar product computation.

Safety:

- We have multiple threads that access a shared resource or variable simultaneously.
- Access is safe if:
  - All access operations have no effect on the shared resource, e.g., all threads perform read only access to a shared variable.
  - All accesses are idempotent, e.g., thread one queries \( \text{if } (a > 0) \{ \ldots \} \) and thread two modifies with \( a = 3 \cdot a \).
  - We allow only one access at a time: mutual exclusion.

Example: interleaving as potential problem

```c
int x; int y;
x = shared; y = shared;
shared = x + 1
shared = y + 1
```

Interleaving OK: does not cause unexpected behavior.

```c
int x; int y;
y = shared;
shared = y + 1;
x = shared;
shared = x + 1;
```

Interleaving problematic: causes unexpected behavior.

```c
int x; int y;
x = shared;
shared = x + 1
shared = y + 1
```
Potential problem: data races
- We have a data race, if two threads access a variable simultaneously and at least one of them is a write access.
- The possible interleaving of threads may lead to undesired or to incorrect results.
- Data races are non-deterministic and depend on the interleaving of the threads ⇒ the results might change in each run.
- Data races often lead to data corruption, which could be a cause of crashes.
- Data races are often difficult to detect, reproduce, and eliminate.
- We use **mutual exclusion** to avoid races, by ensuring **serialized access** to all shared objects.
- It is necessary to observe the **locking discipline**.

---

Programming multithreaded algorithms:

- **OpenMP**: API that supports multi-platform shared memory multiprocessing programming. Consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
- **Pthreads**: POSIX threads define an API for creating and manipulating threads.
- **Boost**: free peer-reviewed portable C++ source libraries. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications.
- **Cilk**: is a linguistic and runtime technology for algorithmic multithreaded programming developed at MIT. Programmer should concentrate on structuring her or his program to expose parallelism and exploit locality.

⇒ we are going to use the **Boost** library.

---

Boost Library: our first multithreaded program.

```cpp
#include <boost/thread/thread.hpp>
#include <iostream>
using namespace std;

void hello()
{
    cout << "Hello world, I'm a thread!" << endl;
}

int main(int argc, char* argv[])
{
    boost::thread thrd(&hello);
    thrd.join();
    return 0;
}
```

Compile: `g++ -O3 -pthread hello.cc -o hello -lBoost`

Output: “Hello world, I’m a thread!”

---

Boost Library: using multiple threads.

```cpp
#include <boost/thread/thread.hpp>
#include <iostream>
#include <cstdio>
using namespace std;

void hello(int t)
{
    printf("Hello world, I’m a thread with number %d\n", t);
}

int main(int argc, char* argv[])
{
    boost::thread* thrds[10];
    // create threads
    for (int i = 0; i < 10; i++)
        thrds[i] = new boost::thread(boost::bind(&hello, i));
    // Join threads
    for (int i = 0; i < 10; i++)
        thrds[i]->join();
    // Free allocated threads
    for (int i = 0; i < 10; i++)
        delete thrds[i];
    return 0;
}
```

Compile: `g++ -O3 -pthread hello.cc -o hello -lBoost`

Output: ”Hello world, I’m a thread!”
**P-4 — Programming — Multiple Threads**

**Output: using printf**

```
Hello world, I'm a thread with number 0!
Hello world, I'm a thread with number 1!
Hello world, I'm a thread with number 3!
Hello world, I'm a thread with number 2!
Hello world, I'm a thread with number 5!
Hello world, I'm a thread with number 6!
Hello world, I'm a thread with number 7!
Hello world, I'm a thread with number 8!
Hello world, I'm a thread with number 9!
Hello world, I'm a thread with number 4!
```

**Output: using cout**

```
Hello world, I'm a thread with number 0!
Hello world, I'm a thread with number 1!
Hello world, I'm a thread with number 3!
Hello world, I'm a thread with number 2!
Hello world, I'm a thread with number 5!
Hello world, I'm a thread with number 6!
Hello world, I'm a thread with number 7!
Hello world, I'm a thread with number 8!
Hello world, I'm a thread with number 9!
Hello world, I'm a thread with number 4!
```

**P-5 — Programming — Boost**

**Threads:**

- `boost::thread` enables the use of multiple threads of execution with shared data.
- We need to add `#include <Boost/thread.hpp>` in order to use class `boost::thread`.
- `boost::thread thrd(&hello)` creates and executes a thread that is bound to the function `hello`.
- `new boost::thread(boost::bind(&hello, i))` is another way to bind a function (including parameters for this function) to a thread.
- We call `thrd.join()` or `thrs[i]->join()` to wait for the thread to finish.
- We use a loop with individual join instruction to wait for a number of threads to finish.

**P-6 — Programming — Boost**

The header file `<Boost/thread.hpp>` includes the classes `boost::barrier` for barriers and `boost::mutex` for locks.

**Barrier:**

- The constructor for `boost::barrier` has one integer parameter, the thread count.
- The thread count is the number of threads that wait at the synchronization point defined by the `wait()` member function.
- Barriers reset after the last thread reached the synchronization point.
- The barrier should be accessible by all threads.

**Lock (mutex):**

- The constructor for `boost::mutex` does not have parameters.
- Use `lock()` member function to enter a critical section.
- After the computation, use the member function `unlock()` to leave the critical section.
- The lock should be accessible by all threads.

**O-21 — Multithreaded Algorithms — Matrix Multiplication**

**Parallel matrix multiplication:**

- We have significantly reduced the synchronization overhead.
  - We reduced the number of locks from $m \cdot n^2$ to 0.
  - We reduced the number of barriers from $n^2$ to 1.
- We moved from a fine-grain to a coarse-grain parallelism.
- We have maximized the computation to communication ratio.
- **Static balancing** is sufficient for built-in datatypes
  - Time for a multiplication and addition does not depend on the actual value.
  - The amount of work is evenly distributed as long as the $m$ distinct parts are the same size.
- **Dynamic balancing** might help with other data types, such as long integer or multi-precision floating points.
  - Time for operations depends on the input (on the bit length).
  - Equal sized parts might not result in equal amount of work.
  - Split into more than $m$ tasks and use a work queue for the task assignments to the $m$ threads.
All-Pair Shortest Path: parallel FASTER-APSP.

- Develop parallel EXTEND based on the parallel matrix multiplication.

**Algorithm (EXTEND(L,W,n))**

1. Let \( L' = (l'_{ij}) \) be a new \( n \times n \) matrix
2. /* Compute in parallel */
3. /* thread 1 */
4. for \((1 \leq i \leq \frac{n}{2})\) do
5.   \( l'_{ij} = \infty \)
6. for \((1 \leq j \leq n)\) do
7.   \( l'_{ij} = \infty \)
8. for \((1 \leq k \leq n)\) do
9.   \( l'_{ij} = \min\left(l'_{ij}, l_{ik} + w_{kj}\right) \)
10. barrier\textsuperscript{m}.wait

- We use the parallel EXTEND within the FASTER-APSP algorithm.
- We do not need any locks and only \( \lg n \) barriers.

Algorithm (FLOYD-WARSHALL(W,n))

1. \( D^{(1)} = W \)
2. for \((1 \leq k \leq n)\) do
3.   let \( D^{(k)} = (d^{(k)}_{ij}) \) be a new \( n \times n \) matrix
4.   /* Compute in parallel */
5. /* thread m */
6. for \(((\frac{m \cdot n - n}{m}) + 1 \leq k \leq \frac{m \cdot n}{m})\) do
7.   \( d^{(k)}_{ij} = \min\left(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}\right) \)
8. barrier\textsuperscript{m}.wait
9. barrier\textsuperscript{m}.wait
10. return \( D^{(n)} \)

- We do not need any locks.
- We need to synchronize (with a barrier) the end of the outer loop body.

⇒ we can compute all-pair shortest paths and the transitive closure in parallel.

**Question:** Which of the loops can be split into distinct parts?

**Question:** Can we parallelize the Floyd-Warshall algorithm?

- Analyze the sequential algorithm and identify parts or tasks that can safely computed in parallel.
- Concentrate on the loops and check if they can be split up into distinct parts.

