Basics of Parallel Computing

General-purpose Programming of Massively Parallel Graphics Processors

Shiraz University, Spring 2010
Instructor: Reza Azimi

Some of the slides are based on Kathy Yelick’s course at Berkeley:
http://www.eecs.berkeley.edu/~yelick/cs267_sp07/

The Need For Parallel Computing

- Solving complex scientific and engineering problems require huge computational powers

- Hardware has hit single-CPU performance barrier
  - Memory Wall
  - Power Wall
  - Design Complexity
  - Etc.
Some Challenging Computations

- **Science**
  - Global climate modeling
  - Biology: genomics; protein folding; drug design
  - Astrophysical modeling
  - Computational Chemistry
  - Computational Material Sciences and Nanosciences

- **Engineering**
  - Semiconductor design
  - Earthquake and structural modeling
  - Computation fluid dynamics (airplane design)
  - Combustion (engine design)
  - Crash simulation

- **Business**
  - Financial and economic modeling
  - Transaction processing, web services and search engines

Can you add to this list?

Units of High Performance Computing

- **High Performance Computing (HPC) units are:**
  - Flop: floating point operation
  - Flops/s: floating point operations per second
  - Bytes: size of data (a double precision floating point number is 8)

- **Typical sizes are millions, billions, trillions...**
  - Meg Mflop/s = 10^6 flop/sec
  - Giga Gflop/s = 10^9 flop/sec
  - Tera Tflop/s = 10^12 flop/sec
  - Peta Pflop/s = 10^15 flop/sec
  - Exa Eflop/s = 10^18 flop/sec
  - Mbyte = 2^{20} = 1048576 \sim 10^6 bytes
  - Gbyte = 2^{30} \sim 10^9 bytes
  - Tbyte = 2^{40} \sim 10^{12} bytes
  - Pbyte = 2^{50} \sim 10^{15} bytes
  - Ebyte = 2^{60} \sim 10^{18} bytes

- **See [www.top500.org](http://www.top500.org) for current list of fastest machines**
Principles of Parallel Computing

- Finding enough parallelism (Amdahl’s Law)
- Granularity
- Locality
- Load balance
- Coordination and synchronization
- Performance modeling

Amdahl’s Law

- Let $f$ be the fraction of work that must be done sequentially, so $(1-f)$ is fraction parallelizable
- Let $P$ be the number of processors

\[
\text{Speedup} (P) = \frac{\text{Time}(1)}{\text{Time}(P)} \leq \frac{1}{(1 - f)}
\]

Example:
- Let $f$ be 80% (0.8)
- Speed up cannot be more than 5 even if you have hundreds of processors
**Granularity**

- **Overhead of Parallelism**
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation

- **Tradeoff**
  - Large units of work reduces overhead of parallelisms
  - Large units of work reduces available parallelism

---

**Locality**

- Large memories are slow
- Fast memories are small
- Algorithms should do most of the work on local data
Load Balance

- Load imbalance is the time that some processors in the system are idle due to
  - Insufficient parallelism (during that phase)
  - Unequal size tasks
    - tree-structured computations
    - fundamentally unstructured problems

- Algorithm needs to balance load

Parallel Computing Platforms

- Shared Memory vs. Distributed Memory
- SIMD vs. MIMD
- UMA vs. NUMA
- ILP vs. TLP
**Shared vs. Distributed Memory**

- **Shared Memory:**
  - Single physical address space
  - All processors have access to a pool of shared memory
  - Example: most existing multiprocessors

- **Distributed Memory:**
  - Each processor has its own memory
  - No direct access to remote memory (message passing is required)
  - Example: clusters

---

**UMA vs. NUMA**

- **Uniform Memory Access (UMA)**
  - Each processor has access to all memory modules with equal latency

- **Non-Uniform Memory Access (NUMA)**
  - Access to different memory modules incur different latencies
**UMA Example: Front Side Bus**

![Diagram of Front Side Bus](http://img.hexus.net/v2/cpu/intel/Tigerton/E8501block-big.png)

**NUMA Example: Hypertransport**

![Diagram of Hypertransport](source: AMD)
**SIMD vs. MIMD**

- **Single Instruction Multiple Data (SIMD) model:**
  - A large number of (usually) small processors.
  - A single "control processor" issues each instruction.
  - Each processor executes the same instruction.
  - Some processors may be turned off on some instructions.

- **Multiple Instruction Multiple Data (MIMD) model:**
  - Each processor executes its own sequence of instructions independently.

**SIMD Example: SSE**

- Stands for: Streaming SIMD Extensions

Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

- Available in most x86 (and x86_64) CPUs today
  - Support for 128-bit size vectors (16 x 8-bit integers or 4 x 32-bit integers)
ILP vs. TLP

- Instruction-Level Parallelism (ILP)
  - Parallelism among individual instructions
  - Implicit
  - Automatically extracted by the microprocessor
  - Limited by
    - Pipeline width
    - Instruction dependencies

- Thread-Level Parallelism (TLP)
  - Multiple concurrent tasks
  - Explicit
  - Requires programmer’s involvement

Threads

- A Stream of Instructions with Execution Context
  - PC, Stack Pointer, Registers

- Several threads can live an address space
  => Communication among threads is through shared memory

- Threads is the unit of CPU scheduling
- Supported by most modern systems
Example: POSIX Thread Interface

/* Creates a new thread, which begins executing at start_procedure */

    t = pthread_create(thread, attributes,
                      start_procedure, args)

/* Suspends the execution of the calling thread until the thread identified by thread_id terminates. */

    pthread_join(thread_id, status)

/* Terminates the calling thread */

    pthread_exit(status)

#include <pthread.h>
#define NUM_THREADS     5

void *Hello(void *threadid) {
    while (1) {
        sleep(1);
        printf("%d: running!
", threadid);
    }
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS]; int rc, t;
    for(t = 0; t < NUM_THREADS; t++){
        printf("Creating thread %d
", t);
        rc = pthread_create(&threads[t], NULL, Hello, (void *)&t);
        // checking for errors here
    }
    pthread_exit(NULL);
}
Thread Categories

- **Coarse-grain**
  - Each thread runs on the processor for thousands (or millions) of instructions before it’s switched out
  - Reduces total context-switch overhead
  - Example: pthreads

- **Fine-grain**
  - Switches between threads on each instruction
  - Useful when thread execution is delayed often
    - Memory latencies, other dependencies
  - Requires special hardware support
  - Example: GPUs

- **Simultaneous Multi-Threading (SMT)**
  - Several threads execute simultaneously on a processor
  - Example: Hyperthreading

**Thread Categories Diagram**

```
<table>
<thead>
<tr>
<th>Time (processor cycle)</th>
<th>Superscalar</th>
<th>Fine-Grained</th>
<th>Coarse-Grained</th>
<th>Multiprocessing</th>
<th>Simultaneous Multithreading</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread 1  Thread 2  Thread 3  Thread 4  Thread 5  Idle slot
```

*SOURCE: David Patterson*