CS3101b – Theory of High-performance Computing

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS3101
Plan

1. Hardware Acceleration Technologies
2. Multicore Programming: Code Examples
3. Distributed computing with Julia
4. CS3101 Course Outline
Plan

1. Hardware Acceleration Technologies
2. Multicore Programming: Code Examples
3. Distributed computing with Julia
4. CS3101 Course Outline
Electronic Numerical Integrator And Computer (ENIAC). The first general-purpose, electronic computer. It was a Turing-complete, digital computer capable of being reprogrammed and was running at 5,000 cycles per second for operations on the 10-digit numbers.
The IBM Personal Computer, commonly known as the IBM PC (Introduced on August 12, 1981).
The Pentium Family.
Hardware Acceleration Technologies
Hardware Acceleration Technologies

![Diagram of a computer architecture with multiple cores and levels of cache](image-url)
Hardware Acceleration Technologies

**Capacity**

- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)

- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte

- **Main Memory**
  - G Bytes
  - 80ns- 200ns
  - ~ $100/ GByte

- **Disk**
  - 10s T Bytes, 10 ms (10,000,000 ns)
  - ~ $1 / GByte

- **Tape**
  - infinite sec-min
  - ~$1 / GByte

**Access Time**

- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)

- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte

- **Main Memory**
  - G Bytes
  - 80ns- 200ns
  - ~ $100/ GByte

- **Disk**
  - 10s T Bytes, 10 ms (10,000,000 ns)
  - ~ $1 / GByte

- **Tape**
  - infinite sec-min
  - ~$1 / GByte

**Cost**

- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)

- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte

- **Main Memory**
  - G Bytes
  - 80ns- 200ns
  - ~ $100/ GByte

- **Disk**
  - 10s T Bytes, 10 ms (10,000,000 ns)
  - ~ $1 / GByte

- **Tape**
  - infinite sec-min
  - ~$1 / GByte

**Staging Xfer Unit**

- **Upper Level**
  - faster
  - faster

- **Lower Level**
  - larger
  - larger

**Registers**

- Instr. Operands
- Blocks
- Pages
- Files

**Memory**

- Blocks
- Pages

**Disk**

- Blocks
- Pages
- Files
The CPU-Memory Gap

The increasing gap between DRAM, disk, and CPU speeds.

Once upon a time, every thing was slow in a computer . . .
Plan

1. Hardware Acceleration Technologies
2. Multicore Programming: Code Examples
3. Distributed computing with Julia
4. CS3101 Course Outline
Cilk and CilkPlus

- Cilk has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- Cilk has been integrated into Intel C compiler under the name CilkPlus, see http://www.cilk.com/
- CilkPlus (resp. Cilk) is a small set of linguistic extensions to C++ (resp. C) supporting fork-join parallelism
- Both Cilk and CilkPlus feature a provably efficient work-stealing scheduler.
- CilkPlus provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation.
- CilkPlus includes the Cilkscreen race detector and the Cilkview performance analyzer.
Nested Parallelism in CilkPlus

```c
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x+y;
}
```

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent.
- CilkPlus keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.
A scheduler’s job is to map a computation to particular processors. Such a mapping is called a schedule.

- If decisions are made at runtime, the scheduler is online, otherwise, it is offline.
- Cilk++’s scheduler maps strands onto processors dynamically at runtime.
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x+y);
    }
}
Benchmarks for the parallel version of the divide-n-conquer mm

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

<table>
<thead>
<tr>
<th>#core</th>
<th>Elision (s)</th>
<th>Parallel (s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>420.906</td>
<td>51.365</td>
<td>8.19</td>
</tr>
<tr>
<td>16</td>
<td>432.419</td>
<td>25.845</td>
<td>16.73</td>
</tr>
<tr>
<td>24</td>
<td>413.681</td>
<td>17.361</td>
<td>23.83</td>
</tr>
<tr>
<td>32</td>
<td>389.300</td>
<td>13.051</td>
<td>29.83</td>
</tr>
</tbody>
</table>
Benchmarks using Cilkview

Speedup for 'multiply 5000x10000 matrix by 10000x5000 matrix'

- parallelism
- burdened speedup
- trials
Plan

1. Hardware Acceleration Technologies
2. Multicore Programming: Code Examples
3. Distributed computing with Julia
4. CS3101 Course Outline
Julia’s message passing principle

Julia’s message passing

- Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processors in shared or distributed memory.

- Julia’s implementation of message passing is one-sided:
  - the programmer needs to explicitly manage only one processor in a two-processor operation
  - these operations typically do not look like message send and message receive but rather resemble higher-level operations like calls to user functions.
Remote references and remote calls

Two key notions: remote references and remote calls

- A **remote reference** is an object that can be used from any processor to refer to an object stored on a particular processor.
- A **remote call** is a request by one processor to call a certain function on certain arguments on another (possibly the same) processor. A remote call returns a remote reference.

How remote calls are handled in the program flow

- Remote calls return immediately: the processor that made the call can then proceeds to its next operation while the remote call happens somewhere else.
- You can **wait** for a remote call to finish by calling wait on its remote reference, and you can obtain the full value of the result using **fetch**.
A first example of parallel reduction

```
@everywhere function count_heads(n)
    c::Int = 0
    for i=1:n
        c += randbool()
    end
    c
end

a = @spawn count_heads(100000000)
RemoteRef(7,1,31)

b = @spawn count_heads(100000000)
RemoteRef(2,1,32)

fetch(a)+fetch(b)
99993168
```

- This simple example demonstrates a powerful and often-used parallel programming pattern: reduction.
- Many iterations run independently over several processors, and then their results are combined using some function.
Distributed arrays and parallel reduction (1/4)

```
 julia> da = @parallel [2i for i = 1:10]
 10-element DArray{Int64,1,Array{Int64,1}}:
   2
   4
   6
   8
  10
  12
  14
  16
  18
  20
```
Distributed arrays and parallel reduction (2/4)

julia> procs(da)
4-element Array{Int64,1}:
  2
  3
  4
  5

julia> da.chunks
4-element Array{RemoteRef,1}:
  RemoteRef(2,1,1)
  RemoteRef(3,1,2)
  RemoteRef(4,1,3)
  RemoteRef(5,1,4)

julia> da.indexes
4-element Array{((Range1{Int64},),),1}:
  (1:3,)
  (4:5,)
  (6:8,)
  (9:10,)

julia> da[3]
6

julia> da[3:5]
3-element SubArray{Int64,1,DArray{Int64,1,Array{Int64,1}},(Range1{Int64},)}:
  6
  8
  10
julia> fetch(@spawnat 2 da[3])
6

julia>

julia> { (@spawnat p sum(localpart(da))) for p=procs(da) }
4-element Array{Any,1}:
  RemoteRef(2,1,71)
  RemoteRef(3,1,72)
  RemoteRef(4,1,73)
  RemoteRef(5,1,74)

julia>

julia> map(fetch, { (@spawnat p sum(localpart(da))) for p=procs(da) })
4-element Array{Any,1}:
  12
  18
  42
  38

julia>

julia> sum(da)
110
julia> reduce(+, map(fetch,
            { (@spawnat p sum(localpart(da))) for p=procs(da) })))
110

julia>

julia> preduce(f,d) = reduce(f,
            map(fetch,
            { (@spawnat p f(localpart(d))) for p=procs(d) })))
# methods for generic function preduce
preduce(f,d) at none:1

julia>

julia> preduce(min, da)
2

julia>

julia> preduce(max, da)
20
function producer()
    produce("start")
    for n=1:2
        produce(2n)
    end
    produce("stop")
end

To consume values, first the producer is wrapped in a Task, then consume is called repeatedly on that object:

julia> p = Task(producer)
Task

julia> consume(p)
"start"

julia> consume(p)
2

julia> consume(p)
4

julia> consume(p)
"stop"
Plan

1. Hardware Acceleration Technologies
2. Multicore Programming: Code Examples
3. Distributed computing with Julia
4. CS3101 Course Outline
Course Topics

Week 1: Course presentation and orientation
Week 2-3: Distributed and parallel computing with the Julia interactive system
Week 4-5: Multicore architectures and the fork-join multithreaded parallelism
Week 6: Analyzing the cache complexity of algorithms
Weeks 7-8: Cache memories and their impact on the performance of computer programs
Week 9-10: Fundamental models of concurrent computations (PRAM and its variants)
Week 11: Highly data parallel architecture models (pipeline, stream, vector, etc.)
Weeks 12: Many-core processors (GPGPUs) with an overview of many-core programming
Weeks 13: Multi-processed parallelism, message passing: an overview
About this course

- **Prerequisites**: Computer Science 2101A/B or 2211A/B.
- **Objectives**: introducing students to the necessary theoretical background (architectures, models of computations, algorithms) in order to understand and practice high-performance computing.
- **This course can be seen as extension of other CS courses such as 3331A - Foundations of Computer Science I 3305B - Operating Systems 3340B - Analysis of Algorithms I 3350B - Computer Architecture, providing the parallel dimension of Today’s Computer Science.**
- **It will become next year a preliminary requirement to 4402B - Distributed and Parallel Systems.**
- **We will cover a large of materials and we will have tutorial every week.**