CS3101b – Theory of High-performance Computing

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS3101
Plan

1. Hardware Acceleration Technologies
2. Distributed computing with Julia
3. Optimizing Code for Data Locality: A Case Study
4. Multicore Programming with CilkPlus
5. CS3101 Course Outline
Plan

1. Hardware Acceleration Technologies
2. Distributed computing with Julia
3. Optimizing Code for Data Locality: A Case Study
4. Multicore Programming with CilkPlus
5. 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

Power Density (W/cm²)

- Sun's Surface
- Rocket Nozzle
- Nuclear Reactor
- Hot Plate
- Pentium® processors

Timeline from '70 to '10.
Hardware Acceleration Technologies

Core

L1 inst
L1 data
L1 ins
L1 data

L2

Core

L1 ins
L1 data
L1 ins
L1 data

L2

Main Memory
Hardware Acceleration Technologies
### Hardware Acceleration Technologies

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

#### Access Time
- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - Approx. 1 ns - ~10 ns

#### Cost
- **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
- **Instr. Operands**
  - prog./compiler
  - 1-8 bytes

- **Blocks**
  - Cache cntl
  - 32-64 bytes

- **Pages**
  - OS
  - 4K-8K bytes

- **Files**
  - user/operator
  - Mbytes

#### Upper Level
- **L1 Cache**
- **L2 Cache**
- **Memory**
- **Disk**

#### Lower Level
- **Tape**
Once upon a time, everything was slow in a computer...
Plan

1. Hardware Acceleration Technologies
2. Distributed computing with Julia
3. Optimizing Code for Data Locality: A Case Study
4. Multicore Programming with CilkPlus
5. CS3101 Course Outline
Tasks (aka Coroutines)

Tasks

- Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner.
- This feature is sometimes called by other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations.
- When a piece of computing work (in practice, executing a particular function) is designated as a Task, it becomes possible to interrupt it by switching to another Task.
- The original Task can later be resumed, at which point it will pick up right where it left off.
Producer-consumer scheme example

```julia
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
julia> p = Task(producer)
Task

julia> consume(p)
"start"

julia> consume(p)
2

julia> consume(p)
4

julia> consume(p)
"stop"
```
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: example

moreno@gorgosaurus:~$ julia -p 4

julia> r = remotecall(2, rand, 2, 2)
RemoteRef(2,1,6)

julia> fetch(r)
2x2 Array{Float64,2}:
  0.675311  0.735236
  0.682474  0.569424

julia> s = @spawnat 2 1+fetch(r)
RemoteRef(2,1,8)

julia> fetch(s)
2x2 Array{Float64,2}:
  1.67531  1.73524
  1.68247  1.56942

Commnets on the example

- Starting with julia -p n provides n processors on the local machine.
- The first argument to remotecall is the index of the processor that will do the work.
- The first line we asked processor 2 to construct a 2-by-2 random matrix, and in the third line we asked it to add 1 to it.
- The @spawnat macro evaluates the expression in the second argument on the processor specified by the first argument.
Distributed arrays and parallel reduction (1/4)

[moreno@compute-0-3 ~]$ julia -p 5

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>

julia> da.indexes
4-element Array{((Range1{Int64},),1),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
Distributed arrays and parallel reduction (3/4)

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> function Base.minimum(x::Int64, y::Int64)
    min(x,y)
end
minimum (generic function with 10 methods)

julia> preduce(minimum, da)
2
Plan

1. Hardware Acceleration Technologies
2. Distributed computing with Julia
3. Optimizing Code for Data Locality: A Case Study
4. Multicore Programming with CilkPlus
5. CS3101 Course Outline
A typical matrix multiplication C code

```c
#define IND(A, x, y, d) A[(x)*(d)+(y)]

uint64_t testMM(const int x, const int y, const int z) {
    double *A; *B; *C;
    long started, ended;
    float timeTaken;
    int i, j, k;
    srand(getSeed());
    A = (double *)malloc(sizeof(double)*x*y);
    B = (double *)malloc(sizeof(double)*x*z);
    C = (double *)malloc(sizeof(double)*y*z);
    for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
    for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
    for (i = 0; i < x*y; i++) A[i] = 0 ;
    started = example_get_time();
    for (i = 0; i < x; i++)
        for (j = 0; j < y; j++)
            for (k = 0; k < z; k++)
                // A[i][j] += B[i][k] + C[k][j];
                IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,y);
    ended = example_get_time();
    timeTaken = (ended - started)/1.f;
    return timeTaken;
}
```
Issues with matrix representation

- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte per cache line)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix C should reduce L1 cache misses!
float testMM(const int x, const int y, const int z)
{
    double *A; double *B; double *C; double *Cx;
    long started, ended; float timeTaken; int i, j, k;
    A = (double *)malloc(sizeof(double)*x*y);
    B = (double *)malloc(sizeof(double)*x*z);
    C = (double *)malloc(sizeof(double)*y*z);
    Cx = (double *)malloc(sizeof(double)*y*z);
    srand(getSeed());
    for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
    for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
    for (i = 0; i < x*y; i++) A[i] = 0 ;
    started = example_get_time();
    for(j =0; j < y; j++)
        for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C,k,j,y);
    for (i = 0; i < x; i++)
        for (j = 0; j < y; j++)
            for (k = 0; k < z; k++)
                IND(A, i, j, y) += IND(B, i, k, z)*IND(Cx, j, k, z);
    ended = example_get_time();
    timeTaken = (ended - started)/1.f;
    return timeTaken;
}
Issues with data reuse


- Computing a $32 \times 32$-block of A, so computing again 1024 coefficients: 1024 accesses in A, $384 \times 32$ in B and $32 \times 384$ in C. Total $= 25,600$.

- The iteration space is traversed so as to reduce memory accesses.
float testMM(const int x, const int y, const int z)
{
    double *A; double *B; double *C;
    long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
    A = (double *)malloc(sizeof(double)*x*y);
    B = (double *)malloc(sizeof(double)*x*z);
    C = (double *)malloc(sizeof(double)*y*z);
    srand(getSeed());
    for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
    for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
    for (i = 0; i < x*y; i++) A[i] = 0 ;
    started = example_get_time();
    for (i = 0; i < x; i += BLOCK_X)
        for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
                for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                    for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                        for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                            IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,k0,j0,y);
    ended = example_get_time();
    timeTaken = (ended - started)/1.f;
    return timeTaken;
}
Transposing and blocking for optimizing data locality

```c
float testMM(const int x, const int y, const int z)
{
    double *A; double *B; double *C, double *Cx;
    long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
    A = (double *)malloc(sizeof(double)*x*y);
    B = (double *)malloc(sizeof(double)*x*z);
    C = (double *)malloc(sizeof(double)*y*z);
    srand(getSeed());
    for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
    for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
    for (i = 0; i < x*y; i++) A[i] = 0 ;
    started = example_get_time();
    for(j =0; j < y; j++)
        for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C,k,j,y);
    for (i = 0; i < x; i += BLOCK_X)
        for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
                for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                    for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                        for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                            IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(Cx,j0,k0,z);
    ended = example_get_time();
    timeTaken = (ended - started)/1.f;
    return timeTaken;
}
```
Experimental results

Computing the product of two $n \times n$ matrices on my laptop (Quad-core Intel i7-3630QM CPU @ 2.40GHz L2 cache 6144 KB, 8 GBytes of RAM)

<table>
<thead>
<tr>
<th>$n$</th>
<th>naive</th>
<th>transposed</th>
<th>$8 \times 8$-tiled</th>
<th>t. &amp; t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>7854</td>
<td>1086</td>
<td>1105</td>
<td>999</td>
</tr>
<tr>
<td>2048</td>
<td>8335</td>
<td>8646</td>
<td>10166</td>
<td>7990</td>
</tr>
<tr>
<td>4096</td>
<td>747100</td>
<td>69149</td>
<td>100538</td>
<td>69745</td>
</tr>
<tr>
<td>8192</td>
<td>6914349</td>
<td>546585</td>
<td>823525</td>
<td>562433</td>
</tr>
</tbody>
</table>

Timings are in milliseconds.

The cache-oblivious multiplication (more on this later) and the titled multiplication have similar performance.
Other performance counters

Hardware count events

- **CPI**  **Clock cycles Per Instruction**: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism.

- **L1 and L2 Cache Miss Rate**.

- **Instructions Retired**: In the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled.

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE Instructions</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>0.02</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
<tr>
<td>Tiled</td>
<td>0.49</td>
<td>0.02</td>
<td>0.00</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
</tbody>
</table>
Optimizing Code for Data Locality: A Case Study

Analyzing cache misses in the naive and transposed multiplication

Let $A$, $B$ and $C$ have format $(m, n)$, $(m, p)$ and $(p, n)$ respectively.

- $A$ is scanned once, so $mn/L$ cache misses if $L$ is the number of coefficients per cache line.
- $B$ is scanned $n$ times, so $mnp/L$ cache misses if the cache cannot hold a row.
- $C$ is accessed “nearly randomly” (for $m$ large enough) leading to $mnp$ cache misses.
- Since $2mnp$ arithmetic operations are performed, this means roughly **one cache miss per flop!**
- If $C$ is transposed, then the ratio improves to 1 for $L$. 
Analyzing cache misses in the tiled multiplication

- Let $A$, $B$ and $C$ have format $(m, n)$, $(m, p)$ and $(p, n)$ respectively.
- Assume all tiles are square of order $b$ and three fit in cache.
- If $C$ is transposed, then loading three blocks in cache cost $3b^2/L$.
- This process happens $n^3/b^3$ times, leading to $3n^3/(bL)$ cache misses.
- Three blocks fit in cache for $3b^2 < Z$, if $Z$ is the cache size.
- So $O(n^3/(\sqrt{Z}L))$ cache misses, if $b$ is well chosen, which is optimal.
Plan

1. Hardware Acceleration Technologies
2. Distributed computing with Julia
3. Optimizing Code for Data Locality: A Case Study
4. Multicore Programming with CilkPlus
5. 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 Cilk-screen race detector and the Cilkview performance analyzer.
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.
Scheduling

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.
The CilkPlus Platform

```c
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);
    }
}
```

1. Cilk++ source
2. Cilk++ Compiler
3. Hyperobject Library
4. Conventional Compiler
5. Linker
6. Cilkview Scalability Analyzer
7. Cilkview Serialization
8. Cilkview Race Detector
9. Binary
10. Conventional Regression Tests
11. Linker
12. Parallel Regression Tests
13. Runtime System
14. Exceptional Performance
15. Reliable Single-Threaded Code
16. Reliable Multi-Threaded Code
Benchmarks for the parallel version of the divide-n-conquer mm

Multiplying a 4000×8000 matrix by a 8000×4000 matrix

- on 32 cores = 8 sockets × 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. Distributed computing with Julia
3. Optimizing Code for Data Locality: A Case Study
4. Multicore Programming with CilkPlus
5. 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
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.
- In the future, it should become a preliminary requirement to 4402B - Distributed and Parallel Systems.
- We will cover a large variety of materials and we will have tutorial every week.
High-performance computing and symbolic computation

www.bpaslib.org

www.metafork.org

www.cumodp.org

www.regularchains.org
Acknowledgments and references

Acknowledgments.

- Charles E. Leiserson (MIT), Matteo Frigo (Axis Semiconductor) Saman P. Amarasinghe (MIT) and Cyril Zeller (NVIDIA) for sharing with me the sources of their course notes and other documents.
- My past and current graduate students, in particular: Changbo Chen (Chinese Academy of Science) Xiaohui Chen (UWO), Svyatoslav Covano (UWO & École Polytechnique) Anisul Sardar Haque (Mississauga), Xin Li (U. Carlos III), Farnam Mansouri (Microsoft), Wei Pan (Intel Corp.) and Ning Xie (UWO) for their contribution to the materials presented in this tutorial.

References.

- The Implementation of the Cilk-5 Multithreaded Language by Matteo Frigo Charles E. Leiserson Keith H. Randall.
- The Cache Complexity of Multithreaded Cache Oblivious Algorithms by Matteo Frigo and Volker Strumpen.
- http://www.csd.uwo.ca/~moreno/HPC-Resources.html