CS3101b – Theory of High-performance Computing

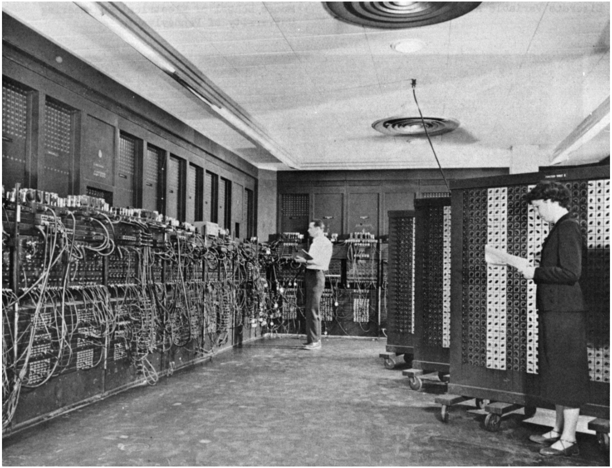
Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS3101

Plan

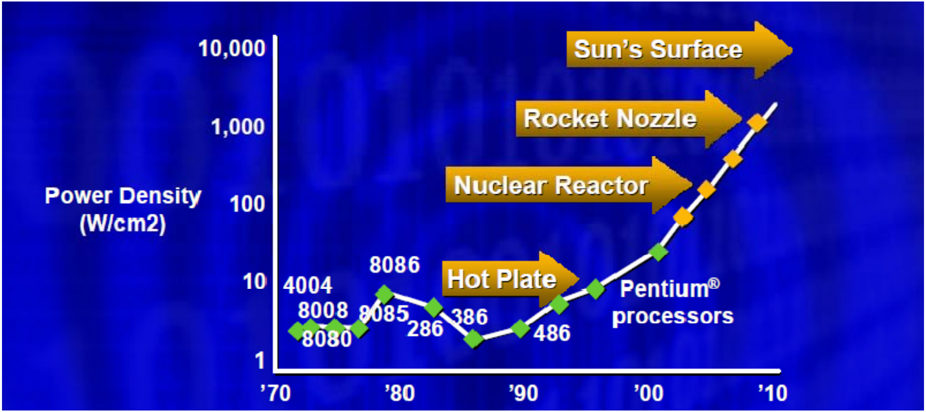
Plan



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.

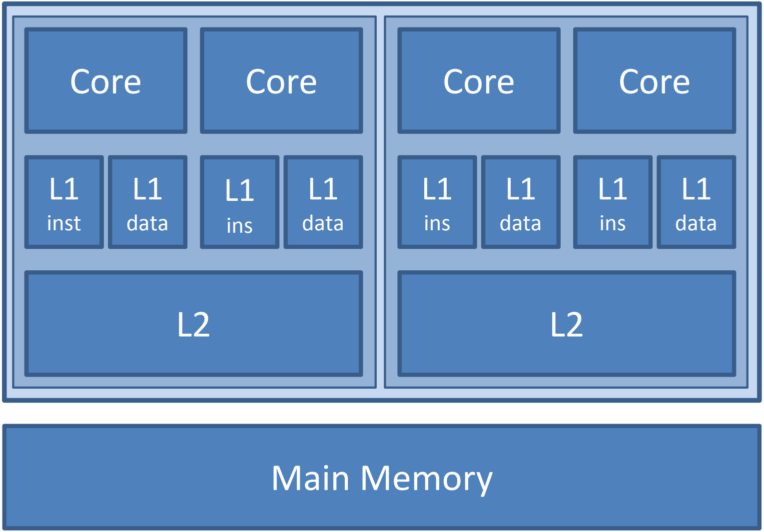
The IBM Personal Computer, commonly known as the IBM PC
(Introduced on August 12, 1981).

The Pentium Family.

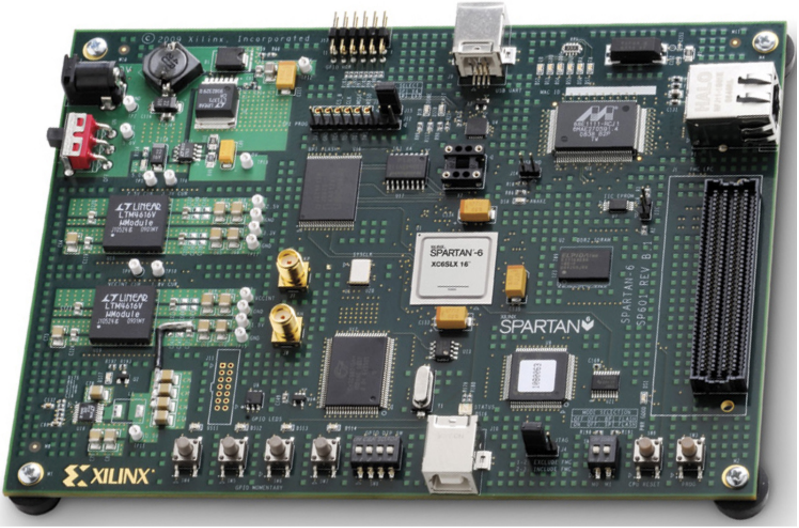


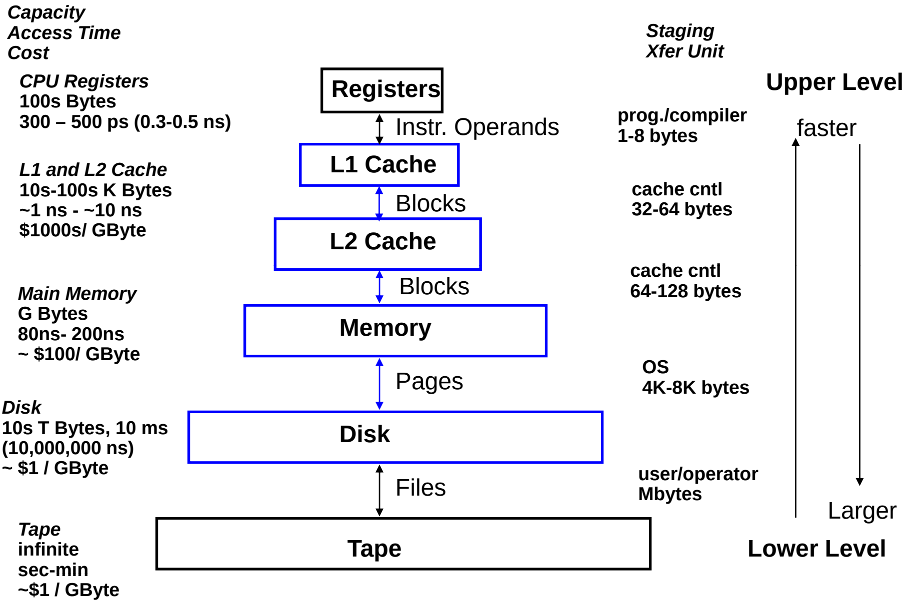
Sun's Surface
Rocket Nozzle
Nuclear Reactor
Hot Plate
Power Density
(W/cm2)
10,000
1,000
100
10
1
4004
8008
8080
8085
8086
286
386
486
Pentium®
processors
'70
'80
'90
'00
'10


INTEL Ⓜ © '05 Q6600
INTEL® CORE™2 QUAD
SLACR MALAY
2.40GHZ/8M/1066/05A
L738B526

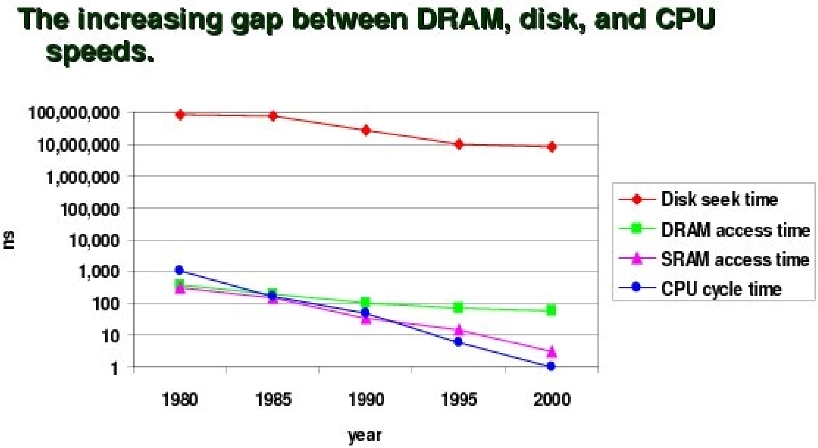

Core
Core
Core
Core
L1 inst
L1 data
L1 ins
L1 data
L1 ins
L1 data
L1 ins
L1 data
L2
L2
Main Memory


NVIDIA
94V-0
MADE IN CHINA
RoHS
COMPLIANT


XILINX
SPARTAN
SPARTAN-6
XC6SLX 16
SPARTAN-6
SP601 REV B.1
© 2009 Xilinx, Incorporated
LTM4616V
µModule


Capacity
Access Time
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
Registers
Instr. Operands
L1 Cache
Blocks
L2 Cache
Blocks
Memory
Pages
Disk
Files
Tape
Staging
Xfer Unit
prog./compiler
1-8 bytes
cache cntl
32-64 bytes
cache cntl
64-128 bytes
OS
4K-8K bytes
user/operator
Mbytes
Upper Level
faster
Larger
Lower Level

The CPU-Memory Gap



Once uopn a time, every thing was slow in a computer . . .

Plan

Cilk and CilkPlus

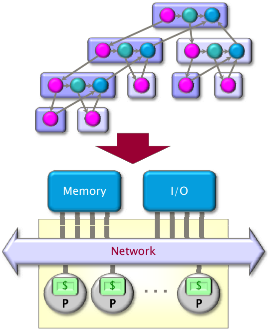
- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- `Cilk` has been integrated into Intel C compiler under the name CilkPlus, see `http://www.cilk.com/`
- `CilkPlus` (resp. `Cilk`) is a small set 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

```
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.

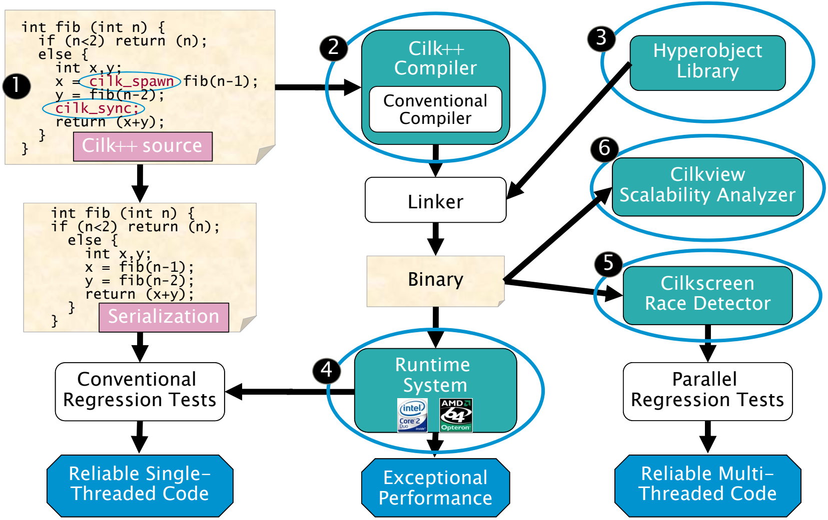
Scheduling



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



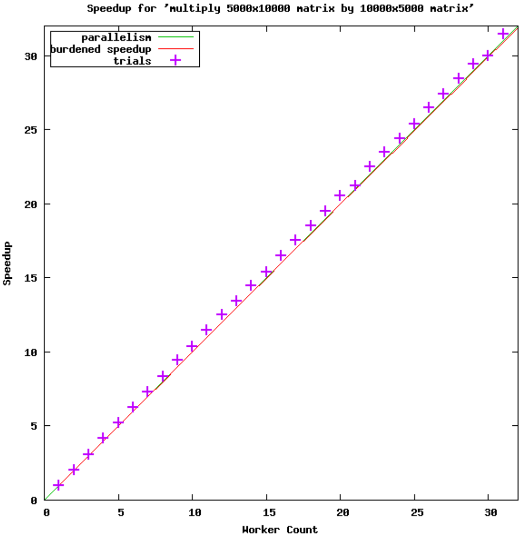
Benchmarks for the parallel version of the divide-n-conquer mm

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.

#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83

Benchmarks using Cilkview



Plan

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.
- Julias 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

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

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

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

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

- This simple example demonstrates a powerful and often-used parallel programming pattern: *reductuon*.
- Many iterations run independently over several processors, and then their results are combined using some function.

Distributed arrays and parallel reduction (1/4)

```
[moreno@compute-0-3 ~]$ julia -p 5
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "help()" to list help topics
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.2.0-prerelease+3622
 _/ |\__'_|_|_|\__'_|  |  Commit c9bb96c 2013-09-04 15:34:41 UTC
|__/                   |  x86_64-redhat-linux

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}:
 (1:3,)
 (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
```

Distributed arrays and parallel reduction (4/4)

```
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
```

Producer-consumer scheme example

```
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:

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

julia> consume(p)
"start"

julia> consume(p)
2

julia> consume(p)
4

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

Plan

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.