

CS415/515 Parallel Programming

Jingke Li

Portland State University

Basic Information

- ▶ Prerequisites
 - Competent with Linux/Unix systems
 - Adequate programming skills in high-level languages
 - No prior parallel programming background is necessary
- ▶ Course Structure
 - 3 hours of lectures, 1 hour lab (optional to grad students)
 - No textbook, notes will be available on D2L
- ▶ Hardware and Software
 - CS Linux Lab machines (linuxlab.cs.pdx.edu)
 - a multicore Linux server (babbage.cs.pdx.edu)
 - Pthreads, OpenMP, Open MPI, Chapel
- ▶ Instructor Info
 - *Office Hours:* MW 13:00-13:55 & by appt @ FAB 120-06
 - *Email:* li@cs.pdx.edu

Grading

- ▶ Class participation (5%)
 - Weekly sign-up sheets
- ▶ Programming Assignments (35%)
 - Hands-on programming exercises using various languages and tools
 - All assignments need to be validated on the CS Linux/Unix systems
- ▶ Exams (60%)
 - Midterm 25%, final 35%
 - Both will be open-book and open-notes (but no internet)
- ▶ Requirements for CS515 students will be higher
 - Details will be specified in individual assignments and in exams

Course Description

An introduction to parallel programming concepts and techniques. Topics include: parallel programming models and languages, share-memory programming, message-passing programming, performance models and analysis techniques, domain-specific parallel algorithms.

Course Goals

Build up a foundation in parallel programming to meet the programming challenges of the future.

- ▶ *Understand the challenges:*
 - computer architectures and systems complexity
 - programming languages complexity
 - applications complexity
- ▶ *Study the principles:*
 - data-parallel vs. task-parallel
 - data races, cache coherence, memory consistency, C synchronization
 - locality, speed-up, performance analysis
- ▶ *Gain hands-on experiences:*
 - shared-memory, message-passing, partitioned global address space
 - Pthreads, OpenMP, MPI, Chapel, OpenCL, ...

Introduction

- ▶ Motivations for parallelism
- ▶ Parallel computing requirements
- ▶ Current status
- ▶ Programming challenges

Motivations for Parallelism

power consumption bottleneck

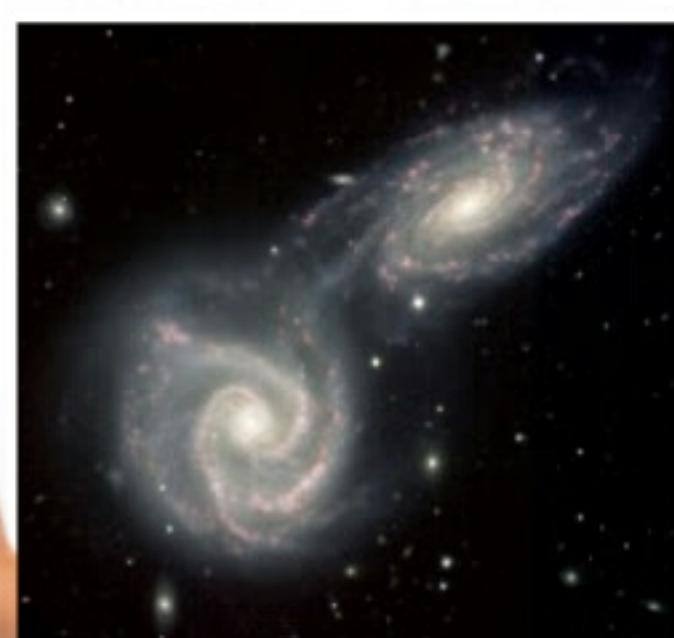
The demand for more computing power never ends:

- Computing has become ubiquitous.
- People always want to solve problems *faster*, and to solve *bigger* and *more challenging* problems. *AlphaGo* -

Super
Computing →



Data Mining
Forensic
Entertainment



(Picture credits: nas.nasa.gov, www.wikipedia.org, zentut.com, topsan.org)

Jingke Li (Portland State University)

CS 415/515 Introduction

7 / 44

How Much Computing Power Is Needed?

- ▶ *Sample 1: Real-time processing of 3D graphics VR, Mouse, etc.*
 - *data elements: 10^9 (1,024 in each dimension)*
 - *operations/element: 200 OPS - operations per second*
 - *update rate: 30/sec TIOPS - integer*

Total requirement: 6×10^{12} IOPS = 6 TIOPS TeraIOPS

- ▶ *Sample 2: Simulation of the earth's climate*
 - *resolution: 100 meters FLOPS - Floating Point OPS*
 - *period: 1 years*
 - *ocean and biosphere models: simple ODEs - ignore*

Total requirement: 10^{20} FLOPS = 100 EFLOPS Exaflops
Etc?

Performance Terms:

IOPS = Integer Operations Per Second

FLOPS = Floating-point Operations Per Second

M(10^6), G(10^9), T(10^{12}), P(10^{15}), E(10^{18}), ...

Jingke Li (Portland State University)

CS 415/515 Introduction

8 / 44

Hardware Challenge and Solution

- ▶ Single CPU performance bottleneck: Main issue: power consumption.
Clock \approx 3 GHz \Rightarrow Performance \approx 10 GFLOPS \rightarrow Slow in Modern world
- ▶ A simple solution — *parallelism!*

100 CPUs @ 10 GFLOPS each	= 1 TFLOPS
1,000 CPUs @ 10 GFLOPS each	= 10 TFLOPS
10,000 CPUs @ 10 GFLOPS each	= 100 TFLOPS
100,000 CPUs @ 10 GFLOPS each	= 1,000 TFLOPS

Theoretically, the power of parallelism is unlimited. With big BUT:
Not every task can be parallelized!

Parallel Computing Requirements

We need three ingredients to succeed: (for successful parallel computation)

- 1 ▶ Parallel algorithms Not all applications can be parallelized
 - Providing source of parallelism
- 2 ▶ Parallel architectures – Multicore . . .
 - Implementing parallelism
- ▶ Parallel languages and tools Bridge between hardware and software
 - Bridging the gap between applications and hardware bridge between 1 and 2.

The Ideal Scenario:

- ▶ The user describes a solution for an application in a high-level (parallel) language. (Not All languages Can do that)
- ▶ The compiler compiles the program onto the target parallel machine.
- ▶ The program runs full speed on the machine.

→ Almost like sequential

But not always possible

Challenges in Reality

Some sequential code is sometimes slower in parallel

- ▶ It is not always simple to parallelize an application.

– Some applications are inherently sequential; others contain inherent sequential components.

- Usually it's a mix of sequential and parallel
- ▶ It is not always easy to write parallel programs. Even if the solution

– Require paradigm shifts. exists → how to connect it to code?

– Need to deal with non-algorithmic issues, such as data partitioning and synchronization. Also tedious and architecture specific

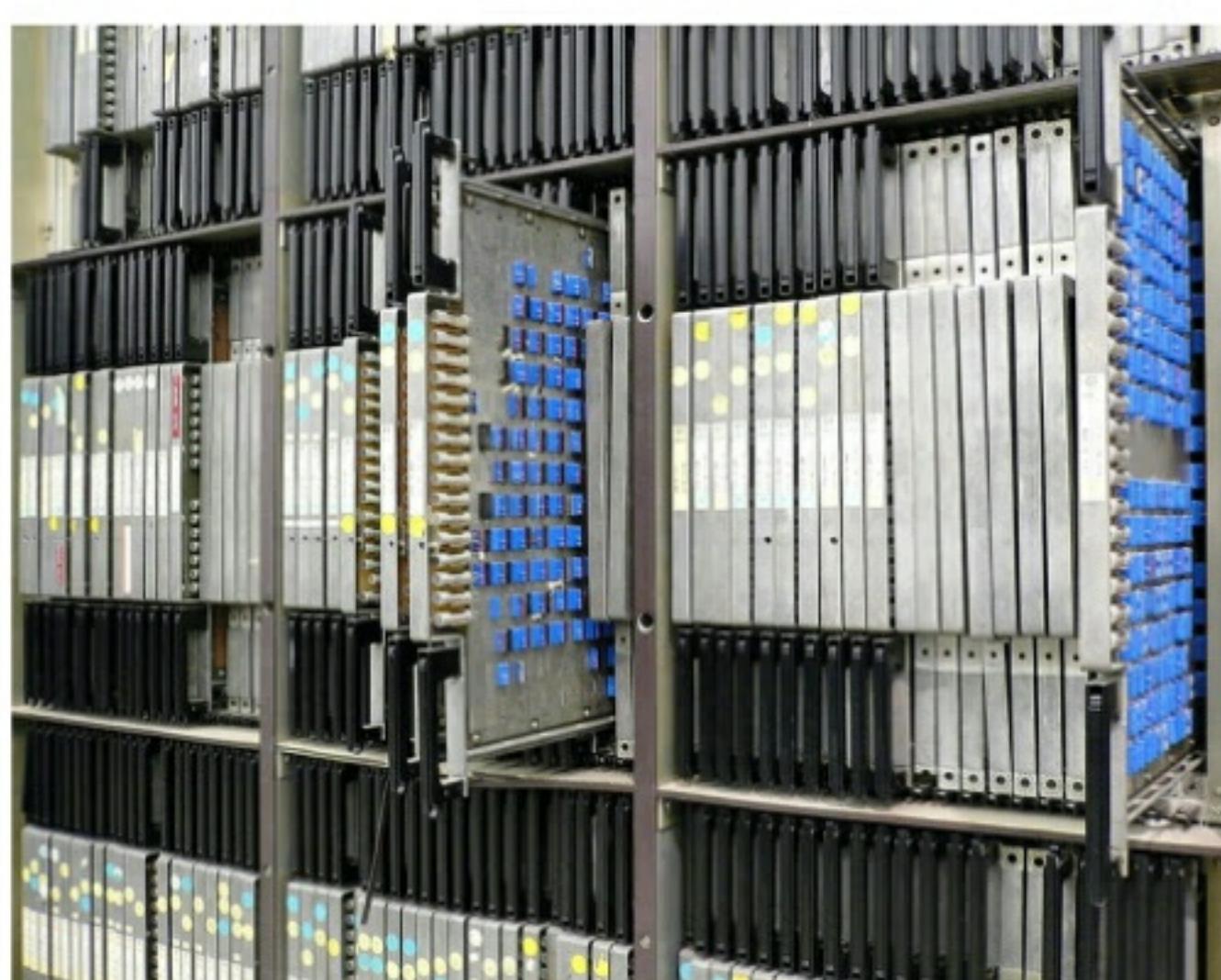
– Languages and tools are often full of systems details.

- ▶ There is a wide spectrum of parallel architectures. The area is advancing fast.
 - No uniform programming model to use. However there is no single standard
 - Parallel programs need to be sensitive to target architecture features, or performance will suffer.
 - No uniform programming model to use and often specific to a particular architecture.
 - Big problem! Area of research.

Historical Lessons — ILLIAC IV

What has been tried?
Approach was around for a very long time (idea is not new)

The first parallel computer, ILLIAC IV, was built in 1967:



(Picture credit: www.wikipedia.org)

▷ Aggressive design at the time

▶ Built with (pre-VLSI era) discrete components

▶ 64 PEs in a 2D mesh working in the lock-stepped manner (a separate CU issues instructions)

▶ Each PE has its own 2048 word 64-bit memory — giving the system a total of 1MB(!) memory

▶ Peak performance 128 MFLOPS vs 10 GFLOPS today.

Was
very large
scale hardware

Main reason for its failure:

- ▶ Hardware reliability — technology was not advanced enough.

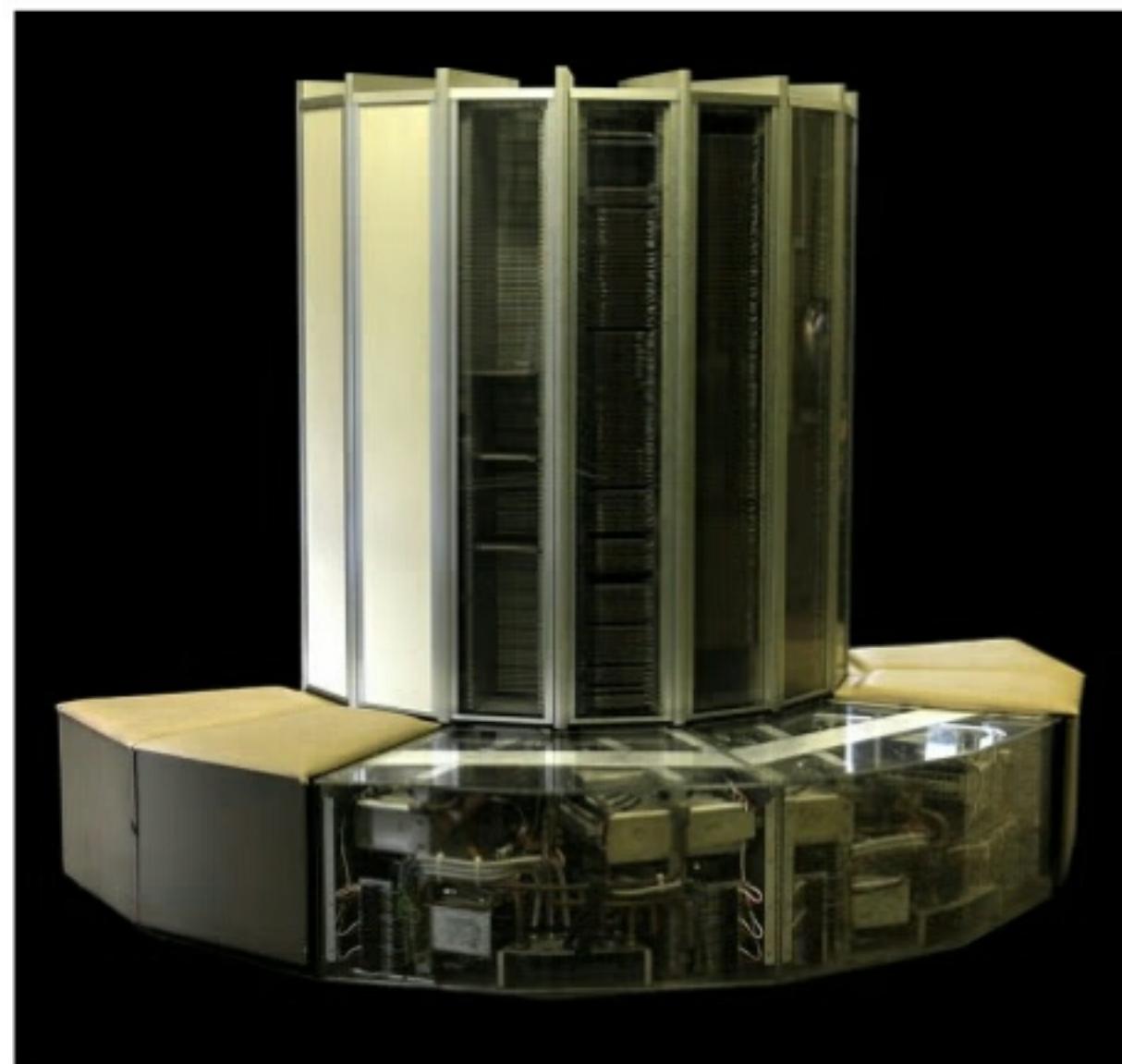
→ Run for a year → retired due to unreliable hardware, use of vacuum tubes.

Next step: Cray

1 CPU but at each step multiple Instructions can be performed.

Historical Lessons — CRAY 1

In the 60s and 70s, vector computers dominated supercomputing field. Among them, Cray-1 is a best representative.



(Picture credit: www.wikipedia.org)

- ▶ Built in 1975, weighed 5.5 tons
- ▶ 2MB of RAM
- ▶ Fetch one instruction per cycle
- ▶ Operate on multiple instructions in parallel and retire up to two every cycle
- ▶ Peak performance: 250 MFLOPS (Today's smartphone has equal or more computing power.)

Refrigerator
in the
middle
was successful
on the market

Main issue:

- ▶ Not scalable

Next milestone.

Historical Lessons — The Second Wave VLSI

New research: Using multiple Chips to achieve better performance

In the late 80s and early 90s, there was a big wave of parallel computer development, with more than two dozens of companies participating:

- ▶ IBM, BBN, Cray, HP, Sun, SGI, Thinking Machine, MasPar, Kendall Square Research, ... *Lots of companies tried to do it*
- ▶ Portland had a large concentration: Intel SSD - scalable system division or Intel (SSD), NCUBE, Sequent, Floating Point Systems, Cogent, ... *supercomputing*

The majority of them did not survive. *small scale (32 CPUs) too advanced*

Main reasons: *Cogent - personal supercomputing*. *Few of them survived thanks to government support.*

- 1 ▶ Language and software developments were lagging behind.
- 2 ▶ Sequential processor was advancing at the Moore's Law speed.

Reasons for failure - no software support

- upgrading cycle is too slow comparing to sequential CPUs.

Where Are We Now?

We now have mature hardware technology:

- ▶ Today's supercomputers are all parallel systems
- ▶ Today's servers are all parallel systems
- ▶ Today's PCs all have multicore chips

Quiz: What do you think the maximum number of CPU cores on an existing computer system are today? 10K? 100K? 1M?

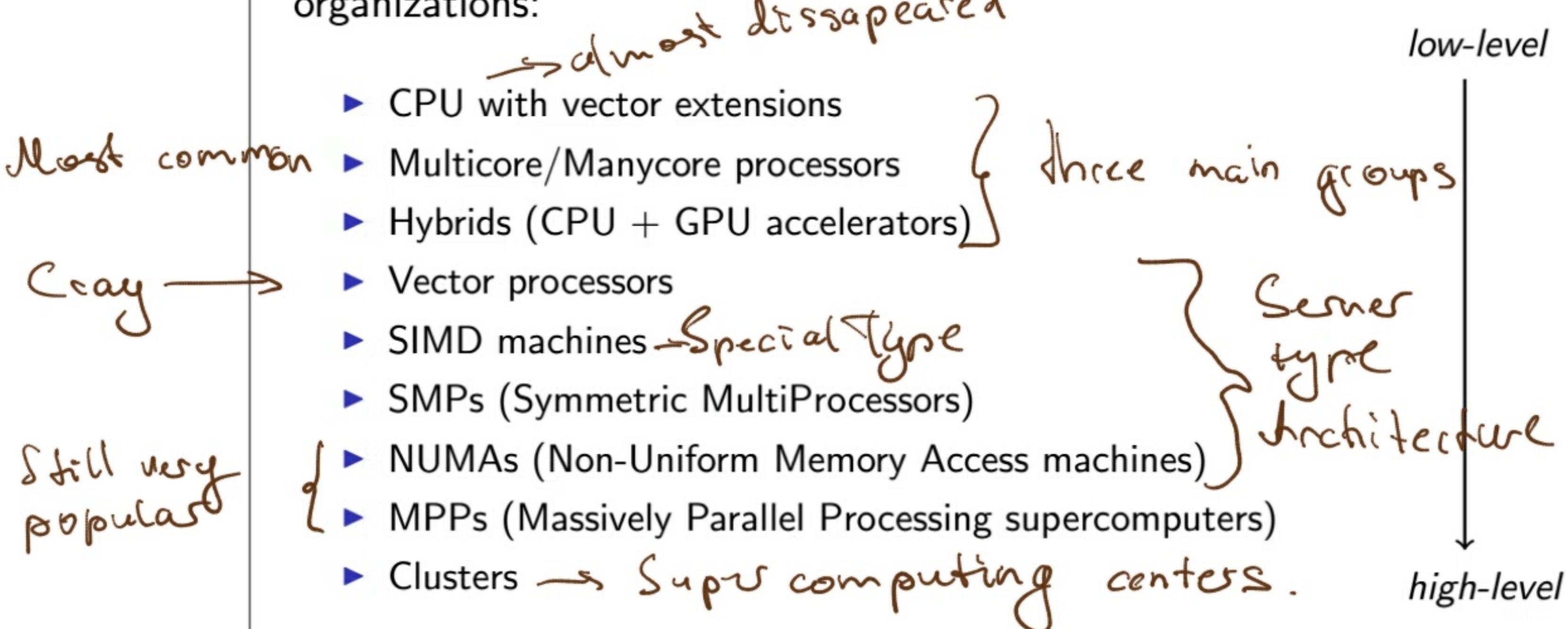
Answer: 3.12 million (Current #1 supercomputer, Milkyway-2)

How to program for such a computer?

→ no magic, or compiler can automate it at this point.

The Landscape of Today's Parallel Architectures

Parallel architectures expand a broad range of computer and system organizations:



CPU with Vector Extensions Was ad hoc extension.

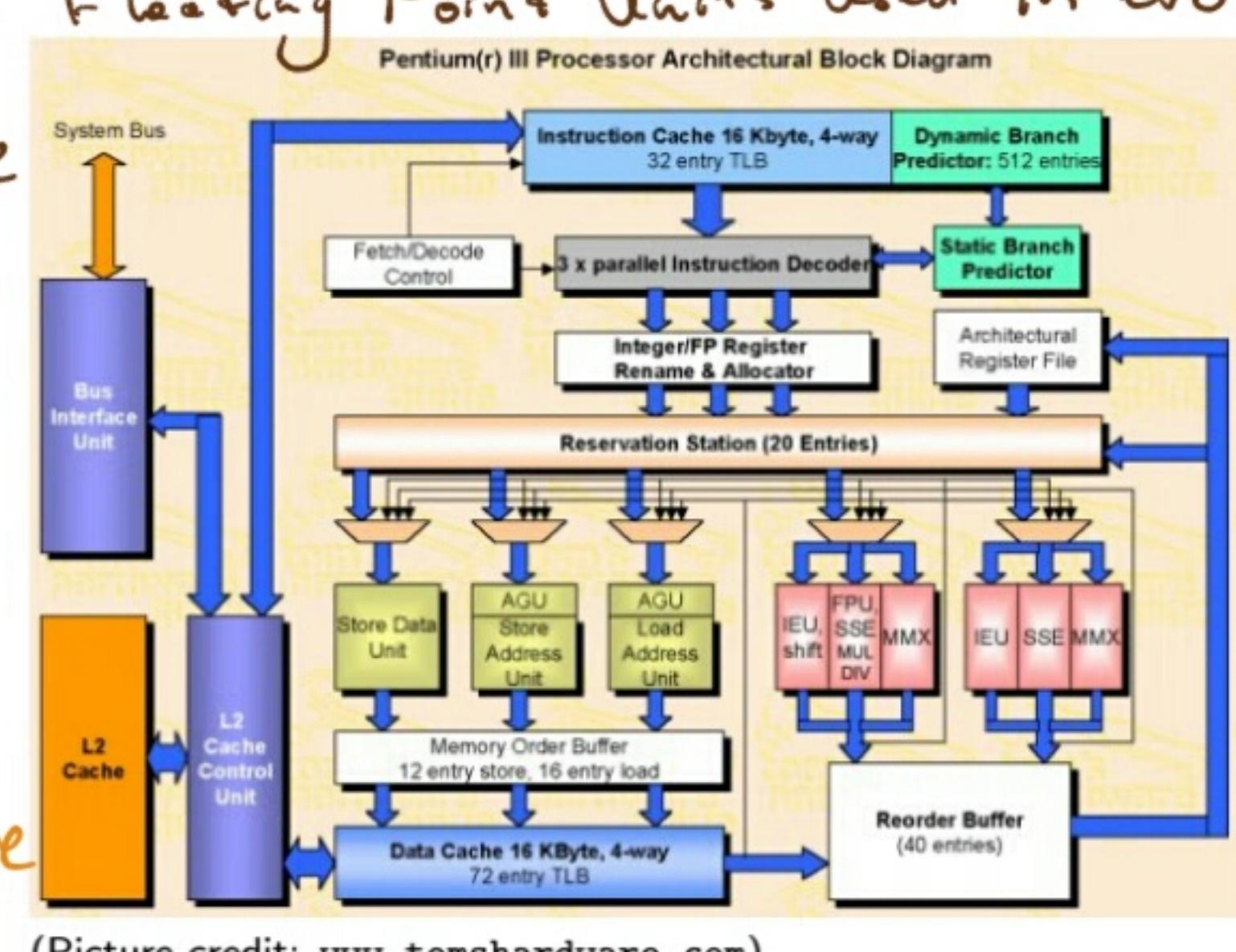
This category of architecture was introduced to support a limited set of parallel operations for graphics applications with minimal hardware changes to a sequential processor — The redundant functional units are typically *overloaded* on existing hardware components.

Example: Intel Pentium III

It was noticed that not all floating point units were used in every application, so it was decided to use them in parallel for something else as vector units.

- ▶ The FPUs are overloaded as vector units.
- ▶ A full set of vector instructions (MMX/SSE) are added for operating the vector units.

Not very effective anymore
too specific



Jingke Li (Portland State University)

CS 415/515 Introduction

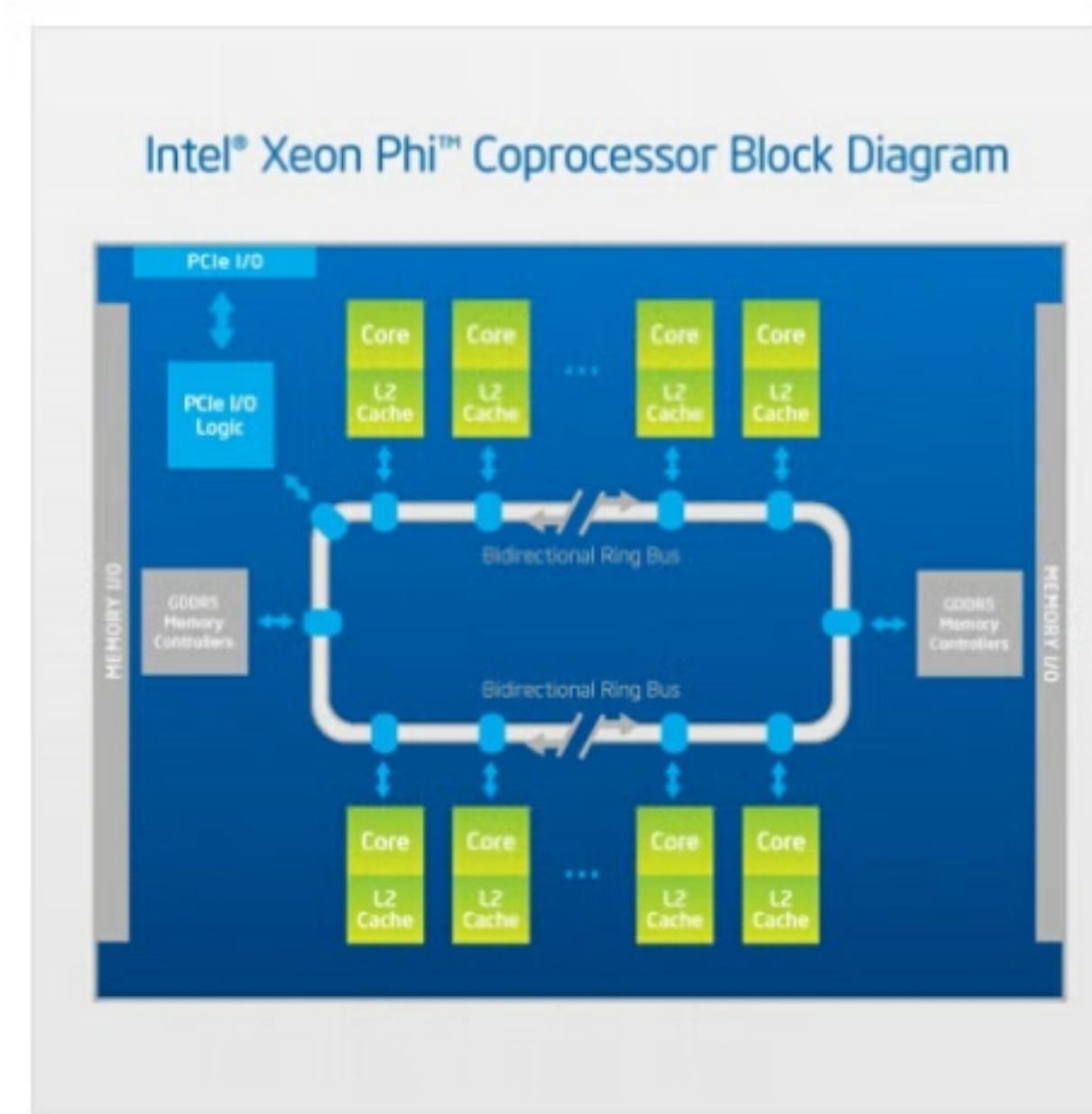
17 / 44

Today's most common approach (Truly Parallel) Multicore/Manycore Processors

Two or more processors on the same chip (*chip-level multiprocessing*).
The processors have individual L1 caches, but share a common L2 cache.

Example: Intel Many Integrated Core Architecture (MIC)

- ▶ Several generations of prototype:
 - Taraflops Chip
 - Larabee
 - Knights Ferry
- ▶ Current brand name: Xeon Phi
 - 60 cores/240 threads
 - 512-bit SIMD instructions
 - Performance: 1.2 TFlop/s



Jingke Li (Portland State University)

CS 415/515 Introduction

18 / 44

Smaller Scale, but needs to be supported by OS
and architecture! (Unlike Vector approach)

Shared L2 cache

Using multicore - automatically reduces L1 cache size!

Multicore/Manycore Processors (cont.)

In comparison to multi-chip SMP designs,

Pros:

- ▶ *Faster cache snoop operations* — signals don't have to travel off-chip
- ▶ *Smaller physical package* — smaller circuitry space is needed
- ▶ *Less power consumption* — since signals are on-chip, the cores can operate at lower voltages

Cons:

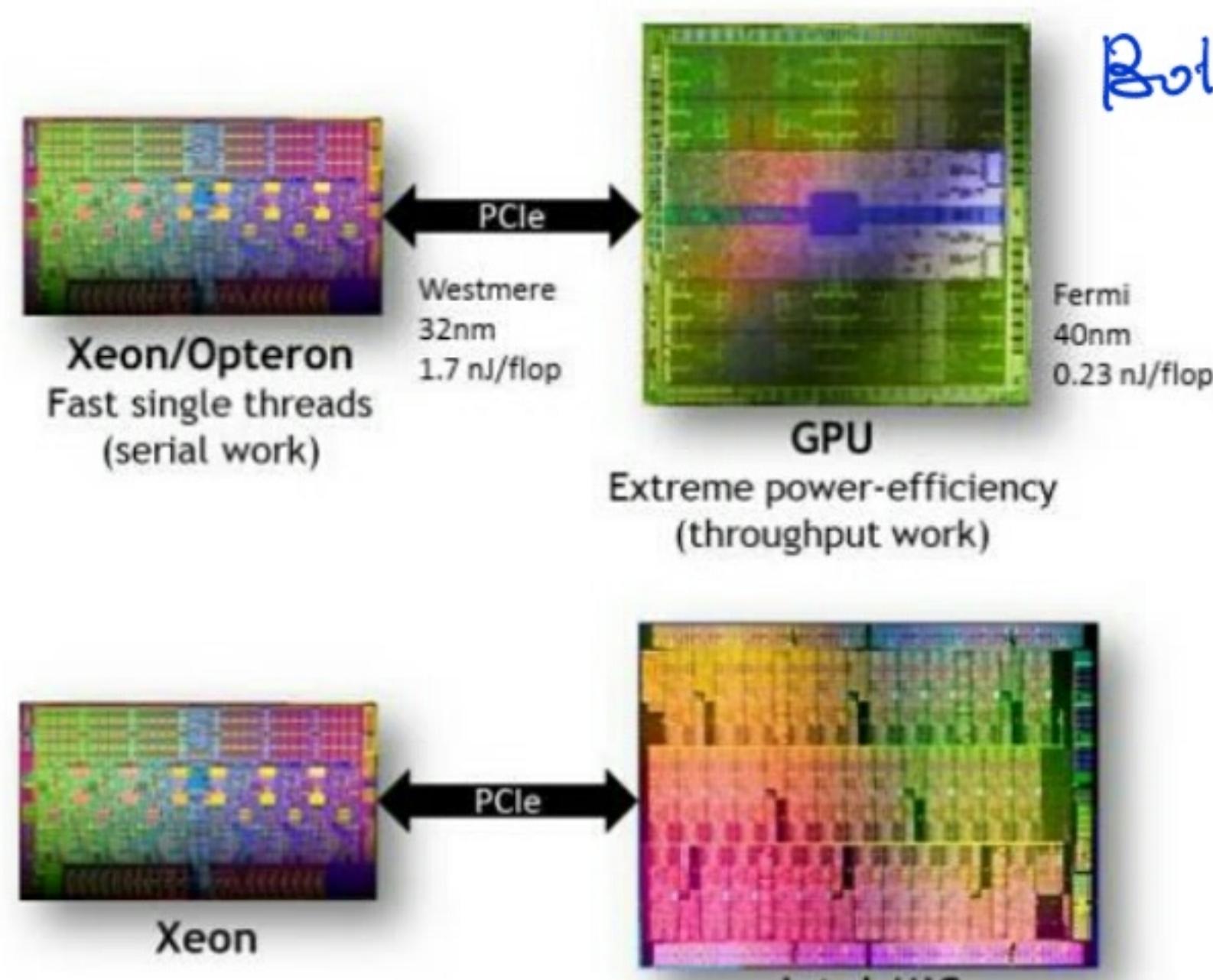
- ▶ Require OS and application software support *hardware can't detect threads so software support is must have*
- ▶ More difficult to manage thermally than single-CPU chips (due to the higher integration)
- ▶ CPU power may be underutilized for some applications, since scaling efficiency is largely dependent on the application or problem set.

Hybrids (CPU + GPU Accelerators) *Winning Combination in many cases.*

Package a CPU and a GPU together.

Examples:

CPU - IO, scheduling etc.
GPU → actual computation.



(Picture credit: www.hpcwire.com)

Every operation can be performed in parallel.

Used to be dominating type due to simplicity (Fortran)
Vector Processors can be automated by a computer easily.

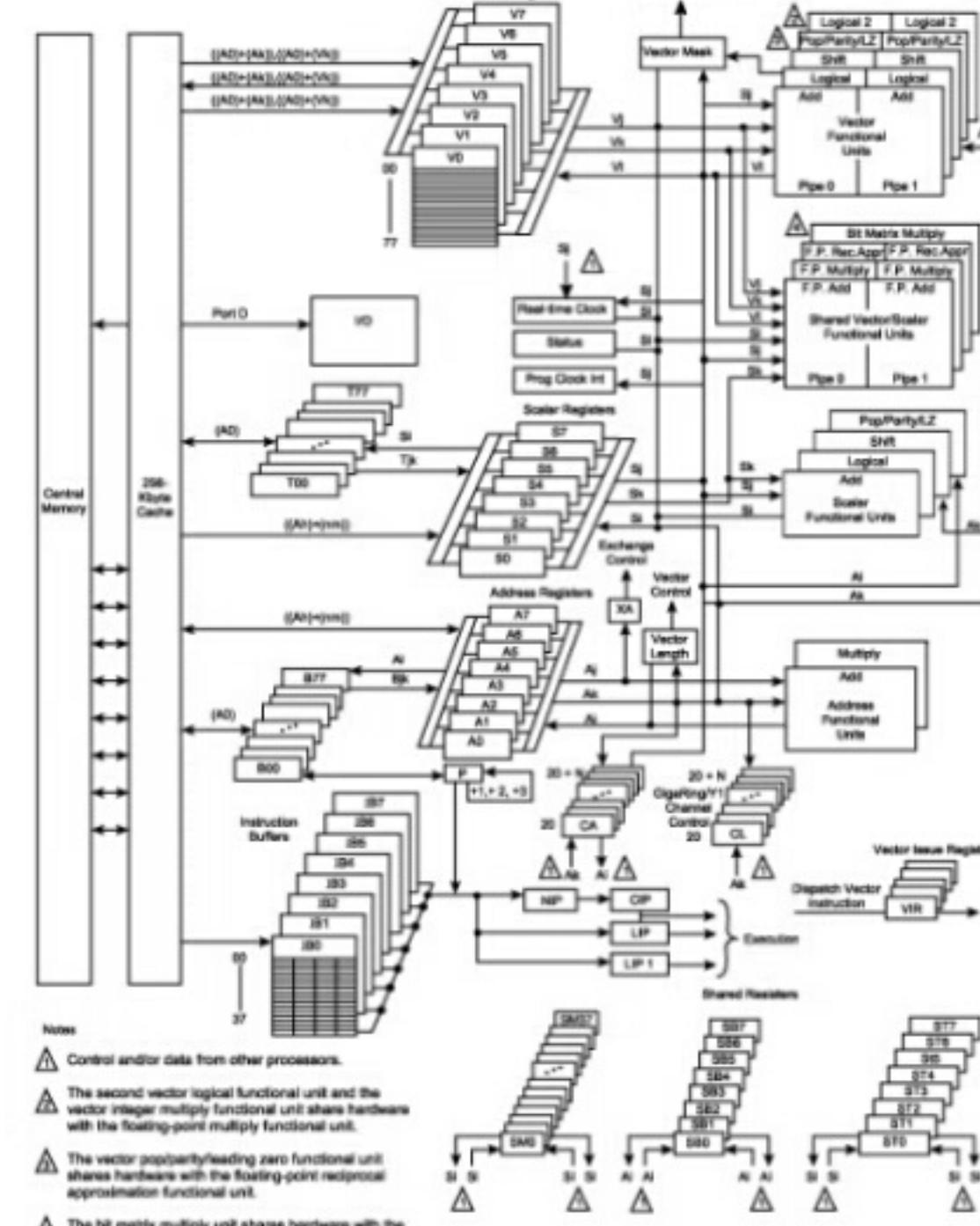
Vector processors derive their performance from a heavily pipelined architecture which can execute special vector instructions very efficiently. The following are the key components:

- ▶ A set of pipelined functional units
- ▶ Special vector registers
- ▶ Special vector instructions
- ▶ Interleaved memory — multiple "banks" allow wide access bandwidth

Facts:

Bad for graphics.

- ▶ Speed up floating-point operations very well, especially on inner loops.
- ▶ Vectorizing compilers are good at identifying code to exploit.
- ▶ Handle irregular data structures poorly; and poor scalability



(Picture credit: docs.cray.com)

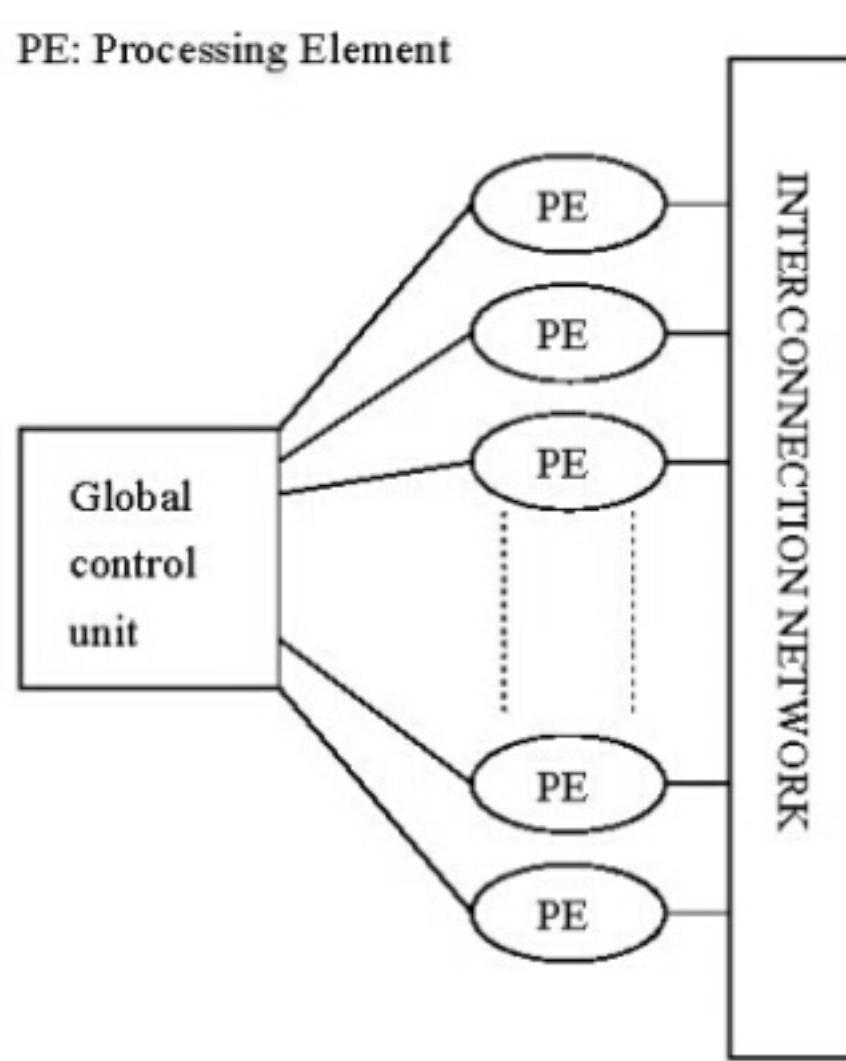
SIMD Systems were popular for long time
SIMD more generalized than Vectors (generalized version of Vector Machines).

SIMD = Single thread of Instructions; Multiple Data items

Packs multiple CPUs

A SIMD system consists of an array of worker processors and a distinguished control processor.

Don't support Floating Point.



- ▶ On each clock cycle, the control processor issues an instruction to all processors using the control bus.
- ▶ Each processor performs that instruction and (optionally) returns a result to the memory via the data bus.
- ▶ The individual processors may have their own memory, or the whole system may share a single main memory.

No support for 32/64.

Only support for 1 bit width

for 32 bit \rightarrow 32 CPUs per bit.

Tradco

Some of them use 4 bit.

- must synchronize all cpus at every tick.
- still manageable by programming languages

But very limited \rightarrow dying.

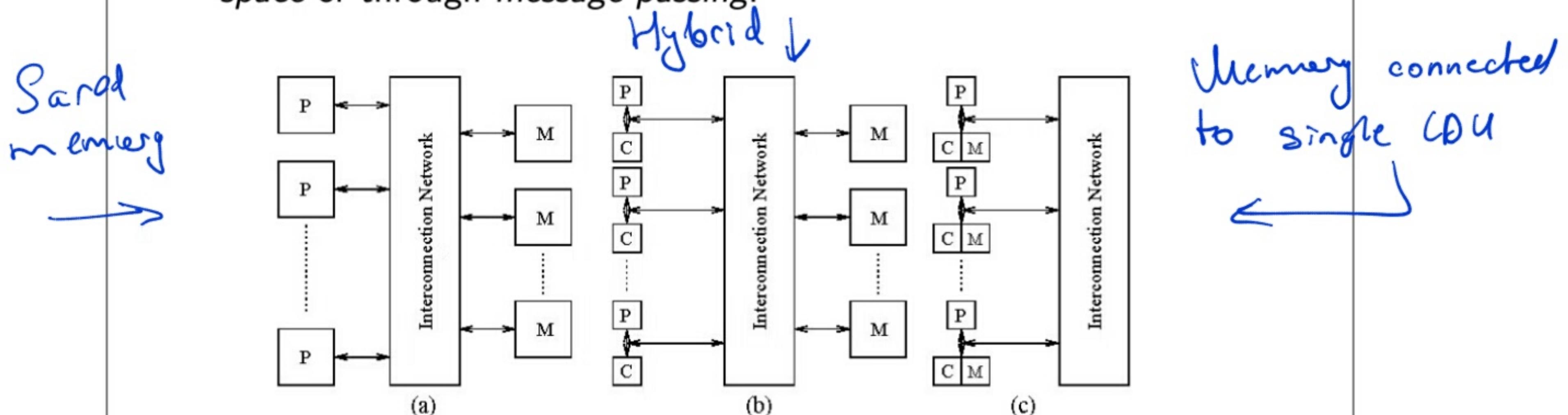
MIMD - all modern machines are MIMD, however there are lots of different specifications
 Shared Memory vs Message Passing.

MIMD Systems

MIMD = Multiple threads of Instructions; Multiple Data items

An MIMD system consists of a collection of processors:

- ▶ Each processor is capable of running a distinct thread of computation.
- ▶ The processors coordinate on a joint program via a *shared address space* or through *message passing*.



Has many subcategories.

Jingke Li (Portland State University)

CS 415/515 Introduction

23 / 44

Main Distinction → Shared Memory or Not (Could be from User point of view, i.e. single address space).

The processors share a single address space.

- ▶ No need to partition or duplicate data
- ▶ Programming style close to sequential programming style — OS can hide the fact of the multiprocessors from applications.
- ▶ Compared to message-passing, less communication overhead

The shared address space can be realized in two ways.

- ▶ Through a single physical memory accessible to all processors. These systems are called *symmetric multiprocessors* (SMPs). Simplified but not scalable
- ▶ Through a set of *distributed memory modules* attached to the processors. These systems are called *non-uniform memory access machines* (NUMAs). not all CPUs have the same speed of accessing memory

Main Issues:

- ▶ Scalability of the interconnection network
- ▶ Memory-cache consistency → memory hierarchy L1, L2, L3..

Jingke Li (Portland State University)

CS 415/515 Introduction

24 / 44

Important to know which one is used due to locality issues.

Look in L1 → L2 → L3 and fetch (single CPU)
 VS same item can be accessed by multiple CPU therefore each CPU has a copy - what happens is value is updated → update must be propagated to all CPUs (expensive)

SMP Systems

SMP = Symmetric MultiProcessors

Commodity microprocessors connected to a *single* shared memory through a high-speed interconnect, typically a bus or a crossbar.

- ▶ *Symmetric* — each processor has exactly the same abilities, any processor can do anything
- ▶ *Single physical address space* — other than processors, there is one copy of everything else (memory, I/O system, OS, etc)
- ▶ *Hardware-supported cache coherence* — typically via snoopy protocols
Typically small scale *Can be done*

SMPs are heavily favored to run commercial applications, e.g. as database or Internet servers.

All major vendors of computer systems are producing and selling these types of machines: Sun, SGI, HP/Compaq, IBM, Intel, ...

Doing very well ← Physical limit 256 CPUs want more?
Jingke Li (Portland State University) CS 415/515 Introduction 25 / 44

use NUMA

NUMA Systems

Main issue \rightarrow how to handle Cache coherence

1. Don't support it

2. Support it (very complex).

Most NUMA systems designed and built are cache-coherent NUMAs (cc-NUMAs). They are distributed shared-memory machines in that

- ▶ The memory is physically distributed among different processors.
- ▶ The system hardware and software maintains coherent caches, and create an illusion of a *single* address space to application users.

Advantages over SMPs: more scalable and increased availability

But the overhead of maintaining cache-coherence can be very high.

Again, all major vendors of computer systems are producing and selling these types of machines.

Cray tried to implement no CC (cache coherence) \rightarrow caching only works for local memory \rightarrow use Shared Memory as Cache

Cache coherence is hard in NUMA \rightarrow need to track/find all the places where data is used (possibly usually expensive)
Much more scalable.

No-Remote-Cache NUMA Systems

On these systems (nrc-NUMA), non-local memory accesses are not cached. As a result, cache coherence is not an issue.

- ▶ The benefits are clear — Zero cache-coherence overhead and high scalability.
- ▶ But so are the downside — Non-local memory accesses are much more expensive.

Cray tried to implement it, but it didn't work very well, the CC is better!

Very General Marketing term referring to any MPP Systems Supercomputing architecture usually refers to systems with 1k CPUs.
MPP = Massive Parallel Processing

An MPP system consists of a large number of *nodes* tightly-integrated by a custom interconnection network,

- ▶ Each node consists of a processor and a memory module
- ▶ Nodes share data by explicitly passing messages
- ▶ The interconnection can be of various forms
 - e.g. a single topology or a hierarchy of structures

Many single Computers interconnected by specially designed bus. Each pc has its own address space

Main Issues:

- ▶ Programming is more challenging
- ▶ Non-local data access is much more expensive than on shared-memory systems

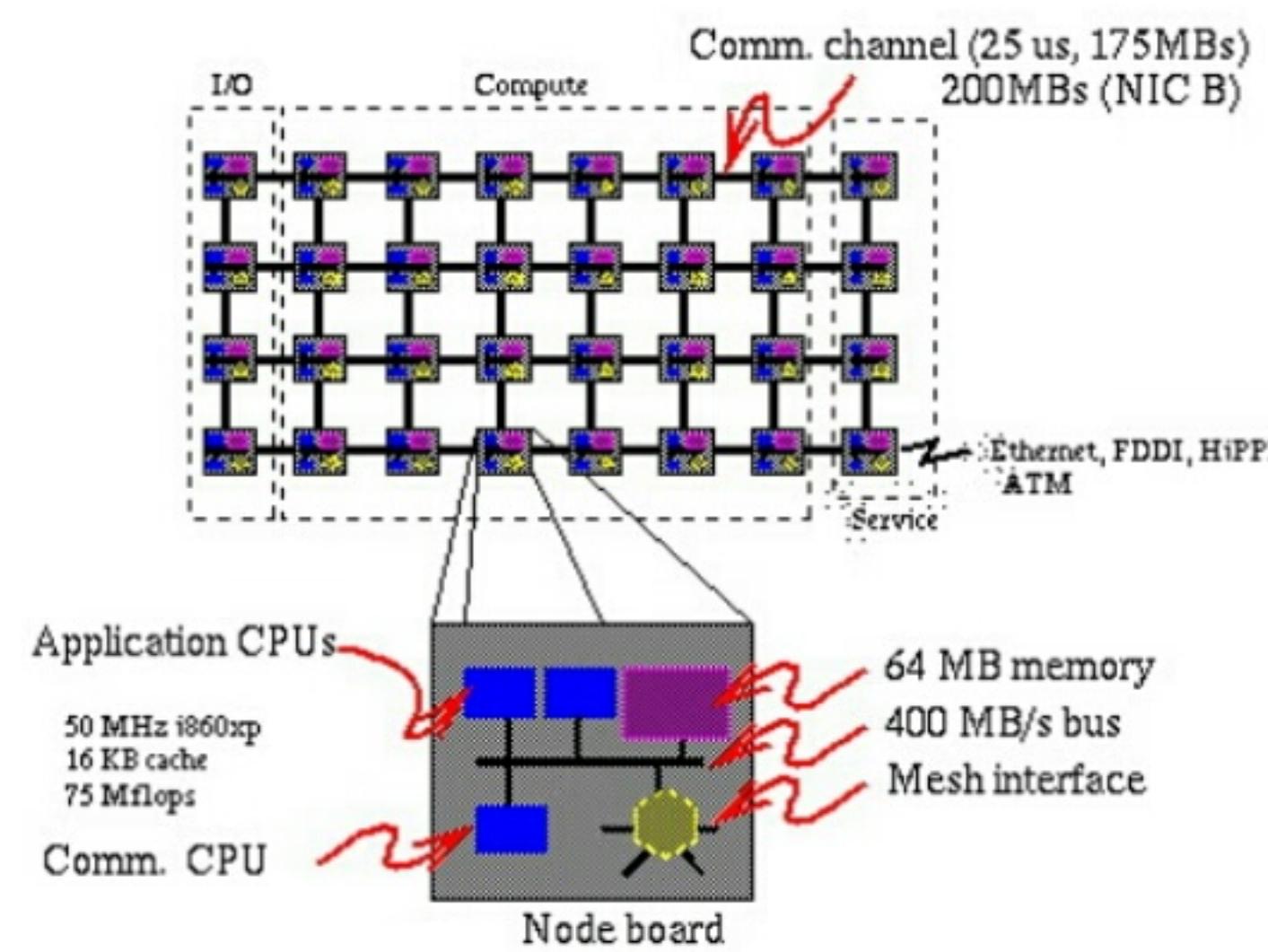
Hard to program because application must be partitioned.
MPP - highly scalable → broadcast → sharing memory is very expensive.

Many tried to implement MPP
 Hypercube was considered most successful but had exponential explosion of messages.
 Nowdays 2D is more popular due to simplicity (Mesh)

MPPs with a Flat Interconnection

Example: ASCI Red Supercomputer (Intel Paragon) ['90s]

- ▶ Compute nodes — 4,640 (9,536 PII Xeon cores)
- ▶ Other nodes — 112
- ▶ Topology — Mesh (38×32)
- ▶ Footprint — 2,500sf (104 cabs)
- ▶ Total memory — 606GB
- ▶ Total storage — 12.5TB
- ▶ Peak performance — 3.2 TFlop/s



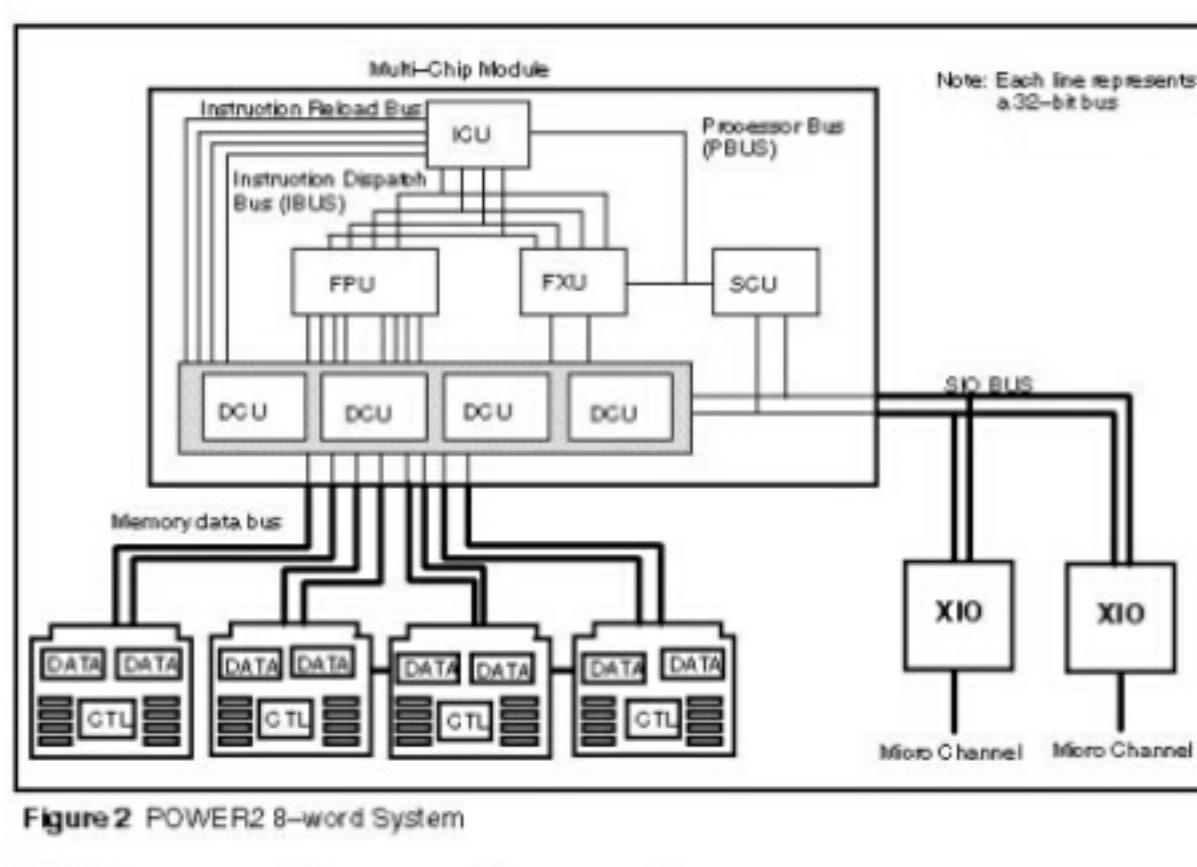
(Picture credit: www.csm.ornl.gov)

All aspects of this system architecture are scalable: communication bandwidth, main memory, internal disk storage capacity, and I/O.

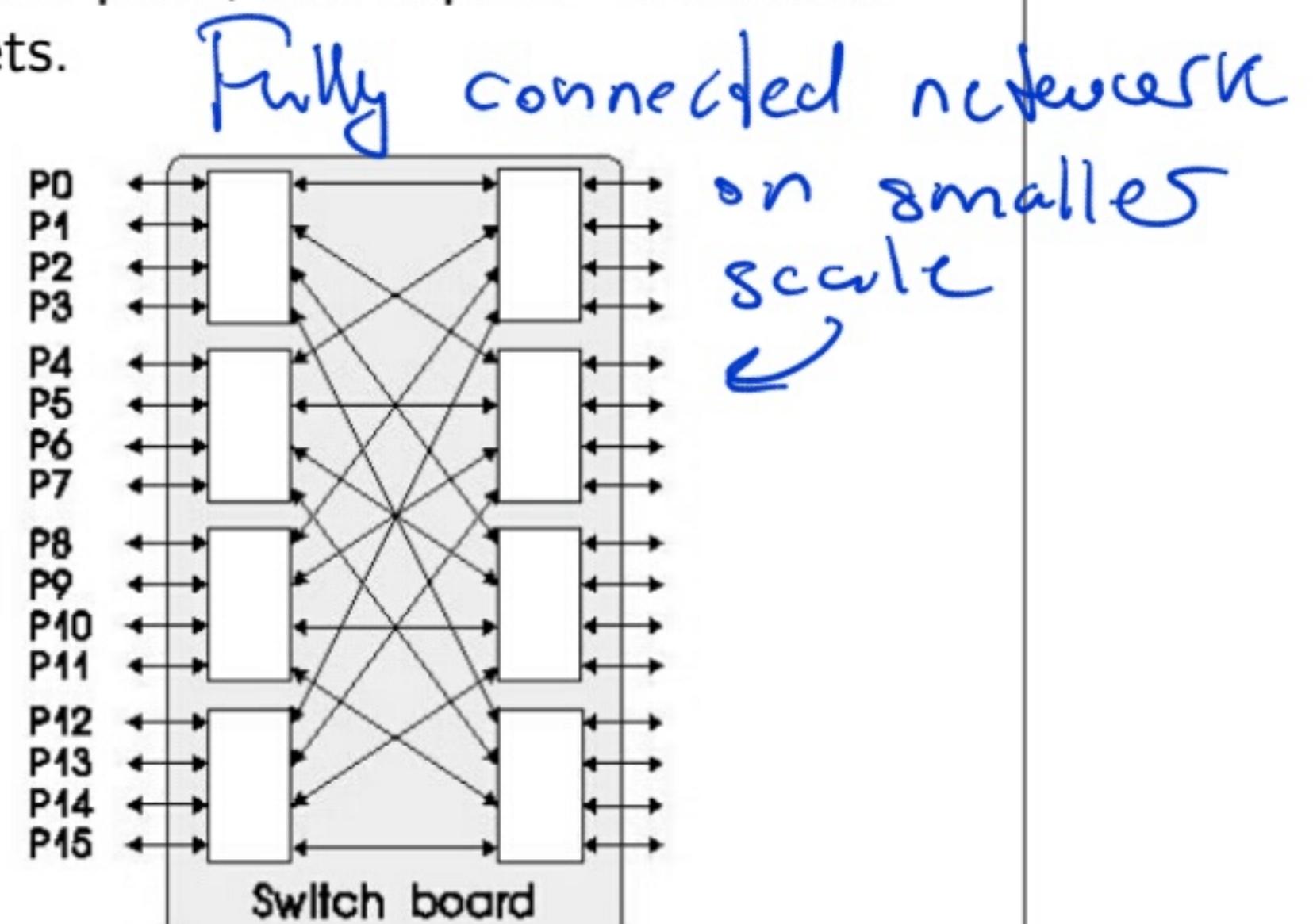
MPPs with a Hierarchical Interconnection

Example: ASCI Blue/White Supercomputers (IBM SP2) ['90s]

- ▶ IBM SP2 Node and Frame:
 - Each cabinet (system frame) holds sixteen nodes, communicating through a SP Switch at 110MB/second peak, full duplex. To make a 128-processor setup, use eight cabinets.



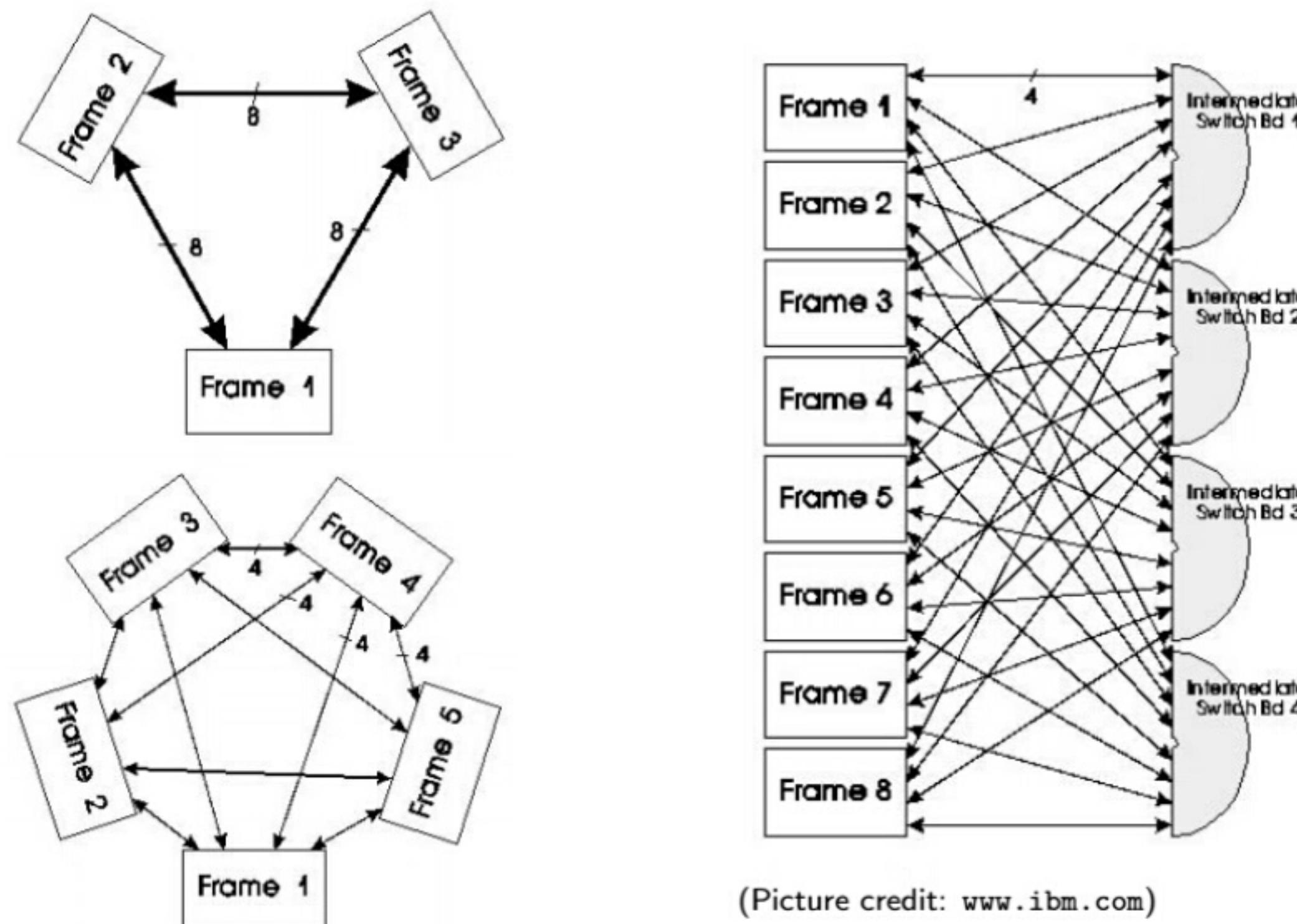
(Picture credit: www.ibm.com)



less connected on larger scale
 still used.

MPPs with a Hierarchical Interconnection (cont.)

- ▶ IBM SP2 Communication System:



(Picture credit: www.ibm.com)

Anything Beyond MPP – IS cluster Cluster Systems

A cluster is a parallel computer system comprising an integrated collection of independent ("off-the-shelf") nodes, each of which is a system in its own right, capable of independent operation.

Clusters offer an attractive alternative to MPPs for supercomputing:

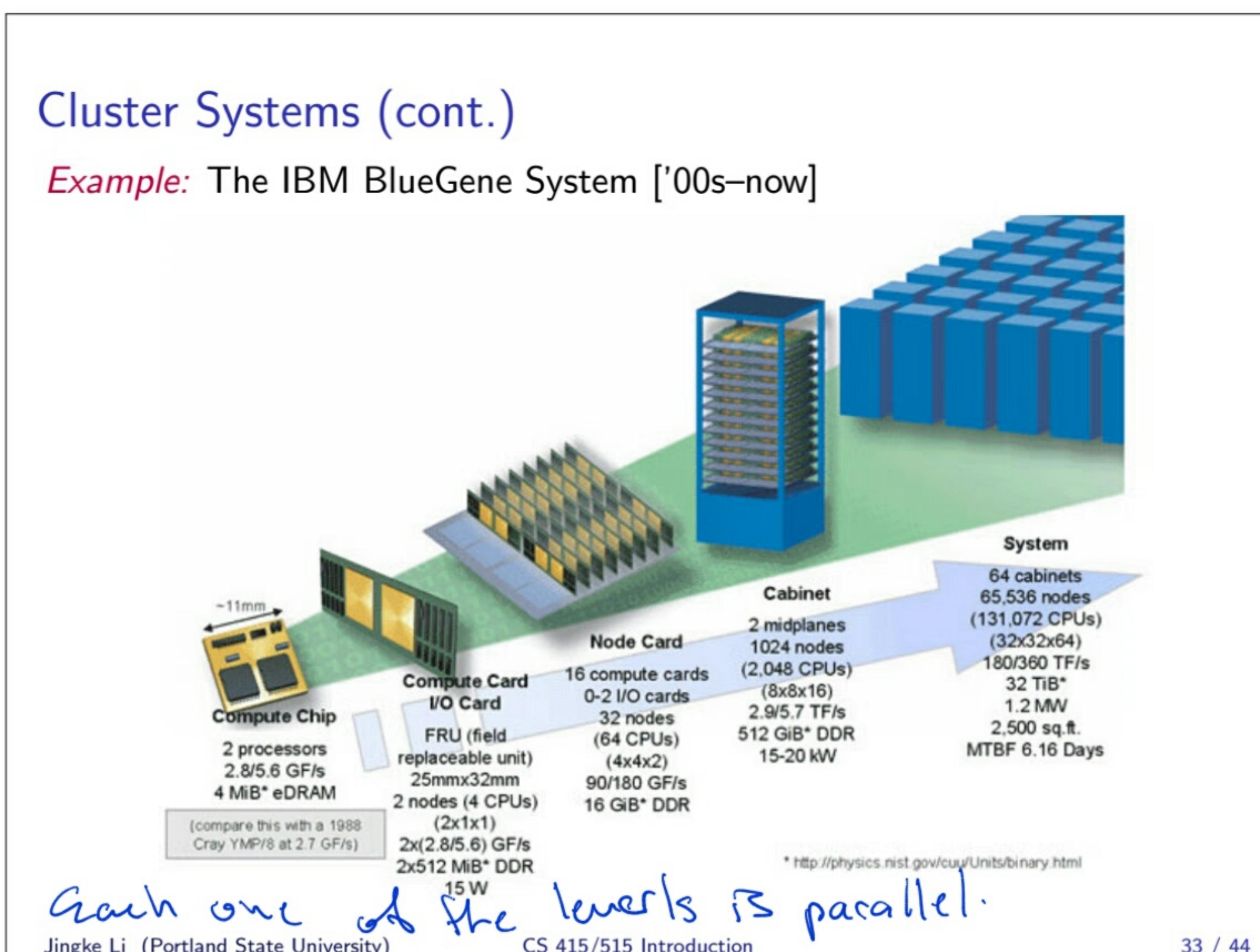
- ▶ The latest processors can easily be incorporated into the system as they become available.
- ▶ They tend to be more scalable.

Example: IBM Roadrunner System ['00s]

- An Opteron cluster with Cell accelerators — Each node consists of a Cell attached to an Opteron core, and the Opterons are connected to each other.
- Total of 6,948 dual-core Opterons and 12,960 Cell chips in 294 racks.
- The final cluster is made up of 18 connected units, which are connected via eight additional (second-stage) Infiniband ISR2012 switches.

Cluster Systems (cont.)

Example: The IBM BlueGene System ['00s–now]



We are more concerned with programming.

How Are Today's Parallel Machines Programmed? No single answer

easy

to handle

- ▶ Vector/SIMD Systems: *High level / simple / easy to use / elegant*
 - A language that supports vector operations (e.g. Fortran 90)
 - A sequential language + a vectorizing compiler (e.g. Fortran + PGI compiler) *// C doesn't work very well due to pointer semantic issues*
- ▶ CPU/GPU Hybrid Systems: *Much of the code is about data movement!*
 - A specialized GPU language (e.g. CUDA or OpenCL)
- ▶ Shared-Memory Systems: *Very common → broad of many developers.*
 - A sequential language + an explicit thread library (e.g. C + Pthreads)
 - A sequential language + meta directives (e.g. OpenMP) *→ compiler decides how to do it.*
- ▶ Message-Passing Systems:
 - A sequential language + an explicit message-passing library (e.g. C + MPI)
- ▶ Supercomputers with a Hierarchical Structure:
 - A combination of the above (e.g. C + MPI + OpenMP) *Pthreads usually not used here due to being expensive.*

Jingke Li (Portland State University)

CS 415/515 Introduction

34 / 44

Usually written by specialized programming
messy code / hard to understand!

No Silver bullet!!!
Academia tried to find it for many years unsuccessfully

How Are Today's Parallel Machines Programmed? (cont.)

The state of art of parallel programming shown above is the result of practical compromises — between performance and elegance, performance has won.

Due to the vast diversity in parallel architectures, it is a great challenge to strive a balance between performance and elegance:

Performance:

- ▶ sensitive to target architecture
- ▶ sensitive to data locality → is the issue!

Elegance (i.e. Ease-of-use, High productivity):

- ▶ architecture independence
- ▶ portability across hardware configurations how many threads available
- ▶ global address space

Question: Is there any hope for a unified parallel programming language? /No answer
but people are trying hard.

Jingke Li (Portland State University)

CS 415/515 Introduction

/No answer
but people are trying hard.

One of the most promising approaches today.

PGAS — A New Programming Model for Message-Passing
Attack the MPP first since it's harder (Attack the hard part first).

The new *Partitioned Global Address Space* (PGAS) programming model is aimed at providing a high-level message-passing programming model — higher than MPI+OpenMP yet retain as much their performance benefit as possible.

The PGAS model supports the following (Important Goals)

- ▶ a global address space directly accessible by any process (like a share-memory model)
- ▶ a local-view programming style (like a distributed memory model) distinguish local vs Remote
- ▶ explicit separation between local and non-local data (enables better locality control, hence better performance)

It relies on compilers to introduce (one-sided) communication to resolve remote references. No need to do explicit MPI due to global space. Global space → local less expensive
→ remote more expensive

Jingke Li (Portland State University)

CS 415/515 Introduction

36 / 44

Single global space is important — global space + MPI

New PGAS-Based Programming Languages

A new crop of PGAS-based parallel programming languages are being actively developed:

- ▶ Co-Array Fortran (CAF)
 - Explicit PGAS language extensions to Fortran 95.
- ▶ Unified Parallel C (UPC)
 - Explicit PGAS language extensions to ANSI C.
- ▶ X10
 - IBM's Java-flavored PGAS language.
- ▶ Chapel
 - A new parallel programming language being developed by Cray Inc.
(Now an open-source project.)

} University projects.

IBM

Performance is the issue.

– A new parallel programming language being developed by Cray Inc.
(Now an open-source project.)

Most ambitious because
it is trying to implement language from scratch
Clean - most promising - open source.

Jingke Li (Portland State University)

CS 415/515 Introduction

37 / 44

Examples of languages, which approach is the best?

An Illustrative Example

Problem: Compute n values and then add them together.

$$sum = \sum_{i=0}^{n-1} compute(i)$$

Sequential Solution:

```
#define N 100           /* problem domain size */  
int sum = 0;  
  
int compute(int i) {  
    return i*i;  
}  
int main(int argc, char **argv) {  
    int i;  
    for (i = 0; i < N; i++)  
        sum += compute(i);  
    printf("The result sum is %d\n", sum);  
}
```

Jingke Li (Portland State University)

CS 415/515 Introduction

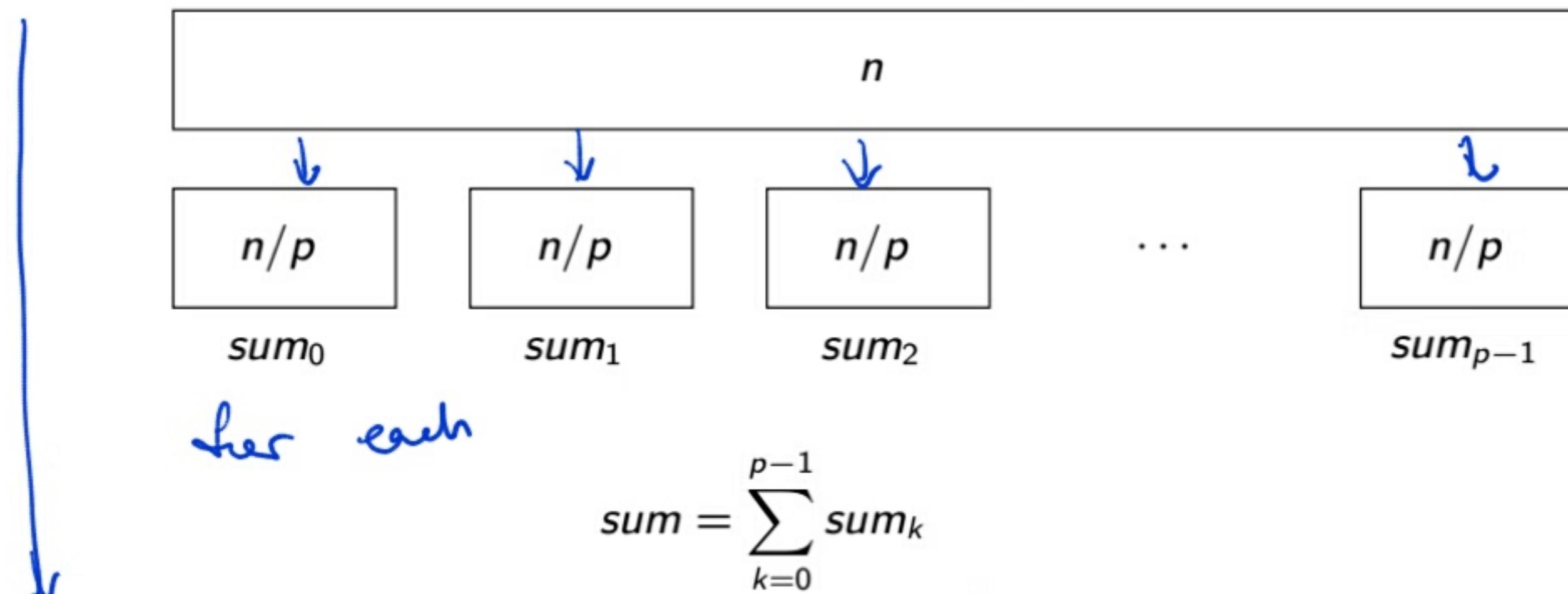
38 / 44

Parallel Solution

p - is the number of resources available (like cpus)

- ▶ Partition n units of work into p ($p \leq n$) groups.
- ▶ Simultaneously perform the computation over the groups, and produce p partial results.
- ▶ Add these results together.

↳ Do it in several different languages.



Tasks sometimes can be complicated and have some internal dependencies

Even from the first glance parallel application looks a lot more complex

Parallel Version 1 (Pthreads Version)

```
#include <pthread.h>
#define N 100           /* problem domain size */
#define P 10            /* number of threads */

pthread_mutex_t sumLock;
int sum = 0;

int compute(int i) {
    return i*i;
}

void worker(long tid) {
    int i, low, high, psum;
    low = (N/P) * tid;    /* a simplistic partition scheme */
    high = low + (N/P);
    psum = 0;
    for (i = low; i < high; i++)
        psum += compute(i);
    pthread_mutex_lock(&sumLock);
    sum += psum;
    pthread_mutex_unlock(&sumLock);
}

int main(int argc, char **argv) {
    pthread_t thread[P];
    long k;
    pthread_mutex_init(&sumLock, NULL); /* initialize mutex */
    for (k=0; k<P; k++)             /* create threads */
        pthread_create(&thread[k], NULL, (void*)worker, (void*)k);
    for (k=0; k<P; k++)             /* join threads */
        pthread_join(thread[k], NULL);
    printf("The result sum is %d\n", sum);
}
```

- ▶ Need to create and manage explicit threads.
- ▶ Need to handle synchronization.
- ▶ Potential performance bottleneck at the global summing statement.

in this version each thread adds results to a global variable hence use of mutex. Another implementation: each thread saves results to its own memory. Then loop over memory to sum up.

Sometimes overhead of using threads negates all the speedup.

Join \Rightarrow bottleneck

OpenMP

Pro Adv: No explicit thread usage - only high level directives.

CBN: Very hard to optimize, not as flexible as pthreads.

OpenMP Directives are complicated on their own (a new language).
What to do when variable name conflict happens (OpenMP share variable name with nos).

Parallel Version 2 (OpenMP Version)

work a lot like sequential version still can be compiled with regular compiler.

```
+ #include <omp.h>
+ #define N 100           /* problem domain size */
+ #define P 10            /* number of threads */

int sum = 0;

int compute(int i) {
    return i*i;
}

int main(int argc, char **argv) {
    int i;
    omp_set_num_threads(P); ← optional, can be moved out
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < N; i++)
        sum += compute(i);
    printf("The result sum is %d\n", sum);
}
```

► Threads and synchronizations are implicit.

► Special reduction directive alleviates performance bottleneck.

► However, directives' semantics can be tricky; and complexity arises quickly when program gets large.

reduction → add multiple numbers into a single sum
if the compiler smart enough it will generate binary tree

Jingke Li (Portland State University)

CS 415/515 Introduction

41 / 44

Message Passing Interface.

It is program's responsibility that each CPU will perform unique work

Parallel Version 3 (MPI Version)

```
#include <mpi.h>
#define N 100           /* problem domain size */

int sum = 0;           /* for holding the result */

int compute(int i) {
    return i*i;         /* some computation on i */
}

int main(int argc, char **argv) {
    int rank, size, dest;
    int i, low, high, psum, sum;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    low = (N/size) * rank; ← standard in all MPI
    high = low + (N/size); ← what if size doesn't divide evenly?
    psum = 0;
    for (i = low; i < high; i++)
        psum += compute(i); ← produce partial sum

    dest = 0;
    MPI_Reduce(&psum, &sum, 1, MPI_INT, MPI_SUM, dest, MPI_COMM_WORLD); ← message passing
    if (rank == dest)      every participant will make the same
        printf("The result sum is %d\n", sum); call.

    MPI_Finalize();
    return 0;
}
```

- Need to partition data.
- Need to manage explicit messages.
- Hard to debug the program.
- Not always easy to tune for performance.

Unique ID → CPUs →

Who is responsible for holding result?
It's usually process 0

Jingke Li (Portland State University)

CS 415/515 Introduction

42 / 44

Cons: Need to partition data and workload (pthread only workload)

MPI uses local copy of data instead of global.

OpenMP is "easy" to debug since it is using the same language

MPI hard to debug since it's executed on multiple machines.

Other Modes used in computation might be involved in some other activities, which creates performance problems.

New version based on Chapel:

- + Can use all forms of parallelization
- ✓ Message Passing
- ✓ Shared memory
- ✓ etc...

Chapel looks similar to sequential C version.
Chapel includes many ideas from many different languages.

Parallel Version 4 (Chapel Version)

Shared-memory version:

```
config const N = 100;
const D = {1..N};
var sum: int;

proc compute(i: int): int {
    return i*i;
}
sum = + reduce [i in D] (compute(i));
writeln("The result sum is " + sum);
```

invented by Fastcom

Message-passing version:

```
use BlockDist; // Partitioning Library.
config const N = 100;
const D = {1..N};
const BlockD = D dmapped Block(boundingBox=D); // new domain
var sum: int; // distributed partitioned domain

proc compute(i: int): int {
    return i*i;
}
sum = + reduce [i in BlockD] (compute(i)); // do work over the
writeln("The result sum is " + sum); // partition domain
```

- At an ideal high level. Almost as the sequential version, no dealing with threads
- Uniform across parallel architectures.
- “Can the language deliver performance?” is still an unanswered -Main issue question. // Open Source updated twice a year.

The most promising direction at the time.

A Road Map of Course Topics

Programming Issues

- synchronization
- memory models
- data partitioning MPI

Programming Models

- data parallel earnest
- shared-memory Pthreads openMP
- message-passing
- PGAS Chapel?

Languages and Tools

- Fortran90, OpenCL CPU
- Pthreads, OpenMP
- MPI
- Chapel

Parallel Algorithms

- master-slave
- map-reduce large scale
- control vs. data ways to look at the program

Performance Analysis

- time complexity
- speedup what degree of parallelization required does it worth to do parallel?

Very limited to data parallel

Examples →

CPU

General info