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, domainspecific 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 parallelims
- Parallel computing requirements
- Current status
- Programming challenges

Motivations for Parallelism

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.





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

How Much Computing Power Is Needed?

- *Sample 1:* Real-time processing of 3D graphics
  - *data elements:* 10^9 (1,024 in each dimension)
  - *operations/element:* 200
  - *update rate:* 30/sec

  Total requirement: 6×10^{12} IOPS = 6 TIOPS
- *Sample 2:* Simulation of the earth's climate
  - *resolution:* 100 meters
  - *period:* 1 years
  - *ocean and biosphere models:* simple

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

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}), ...

Hardware Challenge and Solution

- Single CPU performance bottleneck:

  Clock ≈ 3 GHz ⇒ Performance ≈ 10 GFLOPS
- 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.

Parallel Computating Requirements

We need three ingredients to succeed:

- Parallel algorithms
  - Providing source of parallelism
- Parallel architectures
  - Implementing parallelism
- Parallel languages and tools
  - Bridging the gap between applications and hardware

The Ideal Scenario:

- The user describes a solution for an application in a high-level (parallel) language.
- The compiler compiles the program onto the target parallel machine.
- The program runs full speed on the machine.

Challenges in Reality

- *It is not always simple to parallelize an application.*
  - Some applications are inherently sequential; others contain inherent sequential components.
- *It is not always easy to write parallel programs.*
  - Require paradigm shifts.
  - Need to deal with non-algorithmic issues, such as data partitioning and synchronization.
  - Languages and tools are often full of systems details.
- *There is a wide spectrum of parallel architectures.*
  - No uniform programming model to use.
  - Parallel programs need to be sensitive to target architecture features, or performance will suffer.

Historical Lessons — ILLIAC IV

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

(Picture credit: www.wikipedia.org)

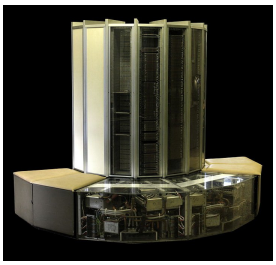
- Built with (pre-VLSI era) discrete components
- 64 PEs in a 2D mesh working in the lock-stepped manner (a separate CU issues instructions)
- Each PE has its own 2048 word 64-bit memory — giving the system a total of 1MB(!) memory
- Peak performance 128 MFLOPS

Main reason for its failure:

- Hardware reliability — technology was not advanced enough.

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.)

Main issue:

- Not scalable

Historical Lessons — The Second Wave

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, ...
- Portland had a large concentration:
  Intel (SSD), NCUBE, Sequent, Floating Point Systems, Cogent, ...

The majority of them did not survive.

Main reasons:

- Language and software developments were legging behind.
- Sequential processor was advancing at the Moore's Law speed.

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)

The Landscape of Today's Parallel Architectures

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

- CPU with vector extensions
- Multicore/Manycore processors
- Hybrids (CPU + GPU accelerators)
- Vector processors
- SIMD machines
- SMPs (Symmetric MultiProcessors)
- NUMAs (Non-Uniform Memory Access machines)
- MPPs (Massively Parallel Processing supercomputers)
- Clusters

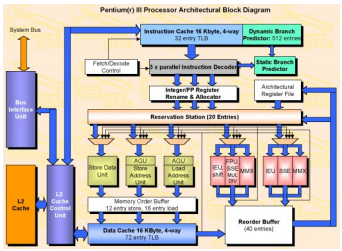
low-level ↓ *high-level*

CPU with Vector Extensions

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

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



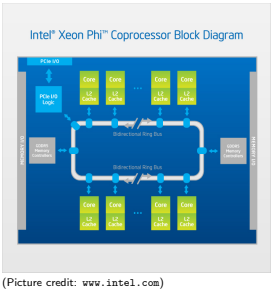
(Picture credit: `www.tomshardware.com`)

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



(Picture credit: `www.intel.com`)

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

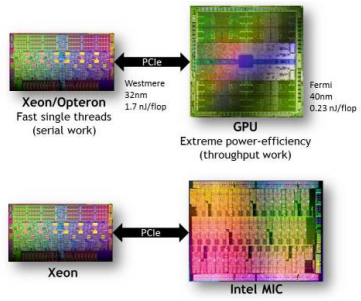
Cons:

- Require OS and application software support
- 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)

Package a CPU and a GPU together.

Examples:



(Picture credit: `www.hpcwire.com`)

Challenges:

- Data movements between host CPU and accelerator GPU are slow.
- Programming is hard.

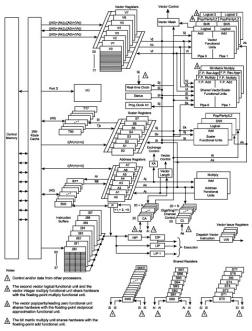
Vector Processors

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:

- 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

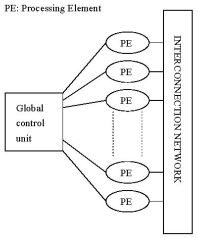


(Picture credit: `docs.cray.com`)

SIMD Systems

SIMD = Single thread of Instructions; Multiple Data items

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



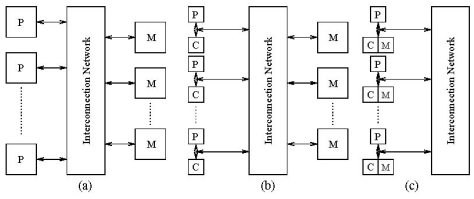
- On each clock cycle, the control processor issues an instruction to all processors using the control bus.
- Each processor performs that instruction and (optionally) returns a result to the memory via the data bus.
- The individual processors may have their own memory, or the whole system may share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

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*.



Shared-Memory MIMD Systems

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).
- Through a set of *distributed* memory modules attached to the processors. These systems are called *non-uniform memory access machines* (NUMAs).

Main Issues:

- Scalability of the interconnection network
- Memory-cache consistency

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

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, IMP, Intel, ...

NUMA Systems

NUMA = Non-Uniform Memory Access

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.

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.

MPP Systems

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

MPPs have the scalability and locality advantages over SMPs.

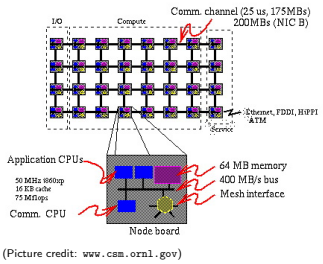
Main Issues:

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

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



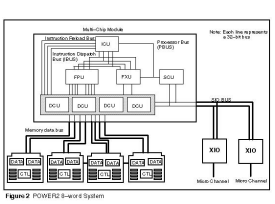
(Picture credit: `www.csm.ornl.gov`)

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

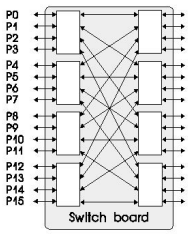
MPPs with a Hierarchical Interconnection

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.

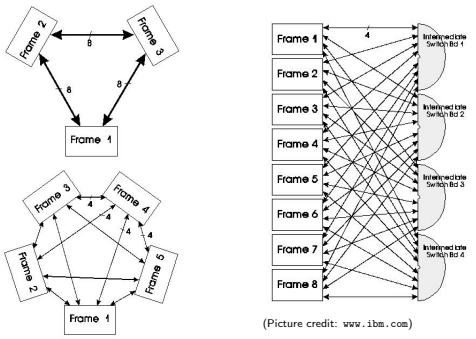


(Picture credit: `www.ibm.com`)



MPPs with a Hierarchical Interconnection (cont.)

- *IBM SP2 Communication System:*



(Picture credit: `www.ibm.com`)

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:

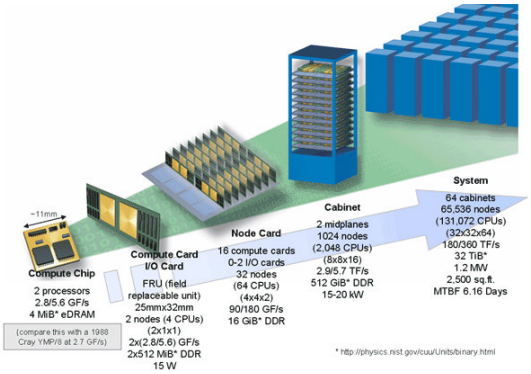
- The latest processors can easily be incorporated into the system as they become available.
- They tend to be more scalable.

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]



How Are Today's Parallel Machines Programmed?

- Vector/SIMD Systems:
  - A language that supports vector operations (*e.g.* Fortran 90)
  - A sequential language + a vectorizing compiler (*e.g.* Fortran + PGI compiler)
- CPU/GPU Hybrid Systems:
  - A specialized GPU language (*e.g.* CUDA or OpenCL)
- Shared-Memory Systems:
  - A sequential language + an explicit thread library (*e.g.* C + Pthreads)
  - A sequential language + meta directives (*e.g.* OpenMP)
- 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)

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

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

- architecture independence
- portability across hardware configurations
- global address space

Question: Is there any hope for a unified parallel programming language?

PGAS — A New Programming Model for Message-Passing

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

- a global address space directly accessible by any process (like a share-memory model)
- a local-view programming style (like a distributed memory model)
- 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.

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.)

An Illustrative Example

Problem: Compute n values and then add them together.

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

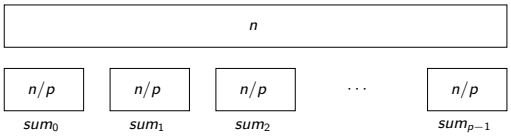
Sequential Solution:

```
#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);
}
```


Parallel Solution

- 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.



$$sum = \sum_{k=0}^{p-1} sum_k$$

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.

Parallel Version 2 (OpenMP Version)

```
#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);
#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.

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) {a
  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;
  high = low + (N/size);
  psum = 0;
  for (i = low; i < high; i++)
    psum += compute(i);

  dest = 0;
  MPI_Reduce(&psum, &sum, 1, MPI_INT, MPI_SUM, dest, MPI_COMM_WORLD);
  if (rank == dest)
    printf("The result sum is %d\n", sum);
  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.

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);
```

Message-passing version:

```
use BlockDist;
config const N = 100;
const D = {1..N};
const BlockD = D dmapped Block(boundingBox=D);
var sum: int;

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

- At an ideal high level.
- Uniform across parallel architectures.
- "Can the language deliver performance?" is still an unanswered question.

A Road Map of Course Topics

Programming Issues
- synchronization
- memory models
- data partitioning

Programming Models
- data parallel
- shared-memory
- message-passing
- PGAS

⇒

Languages and Tools
- Fortran90, OpenCL
- Pthreads, OpenMP
- MPI
- Chapel

Parallel Algorithms
- master-slave
- map-reduce
- control vs. data

⇒

Performance Analysis
- time complexity
- speedup