Recap: What is Parallel Computer?

A parallel computer is a collection of processing elements that cooperate to solve large problems fast.

Some broad issues:

- **Resource Allocation**:
  - how large a collection?
  - how powerful are the elements?
  - how much memory?

- **Data access, Communication and Synchronization**:
  - how do the elements cooperate and communicate?
  - how are data transmitted between processors?
  - what are the abstractions and primitives for cooperation?

- **Performance and Scalability**:
  - how does it all translate into performance?
  - how does it scale?
Recap: Why Parallel Computing

Application demands:
• Computer simulation becomes the third pillar, complementing the activities of theory and experimentation
• General-purpose computing: Video, Graphics, CAD, Database, etc
• Our insatiable need for computing cycles in challenge applications

Technology Trends
• Number of transistors on chip growing rapidly
• Clock rates expected to go up only slowly

Architecture Trends
• Instruction-level parallelism valuable but limited
• Coarser-level parallelism, as in MPs, the most viable approach

Economics

Parallel Computing Models

Historically (1970s - early 1990s), each parallel machine was unique, along with its programming model and language

Throw away software & start over with each new kind of machine
• Dead Supercomputer Society: http://www.paralogos.com/DeadSuper/

Nowadays we separate the programming model from the underlying parallel machine architecture.
• 3 or 4 dominant programming models
• This is still research
Parallel Model for Various Arch.

Can now write portably correct code that runs on lots of machines.

Writing portably fast code requires tuning for the architecture:
- Not always worth it – sometimes programmer time is more important
- Challenge: design algorithms to make this tuning easy

Aspects of a parallel programming model

Control
- how is parallelism created
- what order can operations happen in
- how do different threads of control synchronize

Naming
- what data is private vs. shared
- how shared data is accessed (or communicated)

Operations
- what are the basic operations
- what operations are atomic

Cost
- how do we account for the cost of operations
A generic parallel architecture

Where is the memory physically located?

Parallel Programming Models

• Shared Address Space (SAS)
• Message Passing (MP)
• Data parallel (DP)
Simple example: Sum $f(A[i])$ from $i=1$ to $i=n$

Parallel decomposition:
- Each evaluation of $f$ and each partial sum is a task

Assign $n/p$ numbers to each of $p$ processes
- each computes independent “private” results and partial sum
- one (or all) collects the $p$ partial sums and computes the global sum

Classes of Data:
(Logically) Shared
- the original $n$ numbers, the global sum
(Logically) Private
- the individual function values
- what about the individual partial sums?

Programming Model 1: Shared Memory

Program is a collection of threads of control.
- Can be created dynamically, mid-execution, in some languages

Each thread has a set of private variables, e.g., local stack variables
Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
- Threads communicate implicitly by writing and reading shared variables.
- Threads coordinate by synchronizing on shared variables
SAS Code for Computing a Sum

```sas
static int s = 0;

Thread 1
for i = 0, n/2-1
  s = s + f(A[i])

Thread 2
for i = n/2, n-1
  s = s + f(A[i])
```

- Problem: a race condition on variable s in the program
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

SAS Code: Race Condition

```sas
static int s = 0;

Thread 1
....
compute f(A[i]) and put in reg0
reg1 = s
reg1 = reg1 + reg0
s = reg1
....

Thread 2
....
compute f(A[i]) and put in reg0
reg1 = s
reg1 = reg1 + reg0
s = reg1
....
```

- Suppose s=27, f(A[i])=7 on Thread1 and =9 on Thread2
- For this program to work, s should be 43 at the end
  - but it may be 43, 34, or 36
- The atomic operations are reads and writes
Synchronized SAS Code

static int s = 0;
static lock lk;

Thread 1

local_s1 = 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
lock(lk);
s = s + local_s1
unlock(lk);

Thread 2

local_s2 = 0
for i = n/2, n-1
    local_s2 = local_s2 + f(A[i])
lock(lk);
s = s + local_s2
unlock(lk);

- Since addition is associative, it’s OK to rearrange order
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks
  - Only one thread can hold a lock at a time; others wait for it

SAS Machine Architecture

Any processor can directly reference any memory location
- Comm occurs implicitly as result of loads and stores

SAS Programming model
- Similar to time-sharing on uniprocessors
  - Except processes run on different processors
  - Good throughput on multi-programmed workloads

Naturally provided on wide range of platforms
- History dates to precursors of mainframes in early 60s
- Wide range of scale: few to hundreds of processors

Popularly known as shared memory machines or model
- Ambiguous: memory may be physically distributed among processors
Example: Intel Pentium Pro Quad

- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth

Example: SUN Enterprise

- 16 cards of either type: processors + memory, or I/O
- All memory accessed over bus, so symmetric
- Higher bandwidth, higher latency bus
SAS Machine Model

- Processors all connected to a large shared memory
  - Symmetric Multiprocessors (SMPs)
  - Small scale: < 32 processors typically

```
© C. Xu, 2001
```

Scaling Up

- Dance-hall: bandwidth still scalable, but lower cost than crossbar
  - latencies to memory uniform, but uniformly large (Uniform Memory Access)
- Distributed shared memory (DSM) or non-uniform memory access (NUMA)
  - Construct shared address space out of simple message transactions across a general-purpose network

```
© C. Xu, 2001
```

```
© C. Xu, 2001
```
**Distributed Shared Memory**

Memory is logically shared, but physically distributed
- Any processor can access any address in memory
- Cache lines (or pages) are passed around machine

SGI Origin is a canonical example
- Scales to 100s
- Limitation is cache coherent protocols – need to keep cached copies of the same address consistent

![Diagram of distributed shared memory](image)

---

**Example: Cray T3E**

- Scale up to 1024 processors, 480MB/s links
- Memory controller generates comm. request for nonlocal references
- No hardware mechanism for coherence (SGI Origin etc. provide this)
**Shared Address Space Abstraction**

Virtual Address Space vs Physical Address Space

- Natural extension of uni-proc model:
  - conventional mem operations for comm
  - special atomic operations for synchronization
- OS uses shared memory to coordinate processes

---

**Programming Model 2: Message Passing**

Program consists of a collection of named processes.

- Usually fixed at program startup time
- Thread of control plus local address space -- NO shared data.
- Logically shared data is partitioned over local processes.

Processes communicate by explicit send/receive pairs

- Coordination is implicit in every communication event.

° First possible solution – what could go wrong?

Processor 1
- $xlocal = A[1]$
- send $xlocal$, proc2
- receive $xremote$, proc2
- $s = xlocal + xremote$

Processor 2
- $xlocal = A[2]$
- send $xlocal$, proc1
- receive $xremote$, proc1
- $s = xlocal + xremote$

° If send/receive acts like the telephone system?
  ° synchronous comm

° The post office?
  ° asynchronous comm

Message Passing Architectures

Complete processing node as building block, including I/O
- Communication via explicit I/O operations
- Processor/Memory/IO form a processing node cannot directly access another processor’s memory.

Each “node” has a network interface (NI) for all communication and synchronization.

MP Programming model: directly access only private address space (local memory), comm. via explicit messages (send/receive)
Example: IBM SP-2

- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (bw limited by I/O bus)

Example Intel Paragon

Sanda’s Intel Paragon XP/S-based Supercomputer

2D grid network with processing node attached to every switch

8 bits, 175 MHz, bidirectional
**Message-Passing Abstraction**

- Send specifies buffer to be transmitted and receiving process
- Recv specifies sending process and application storage to receive into
- Memory to memory copy, but need to name processes
- Optional tag on send and matching rule on receive
- User process names local data and entities in process/tag space too
- In simplest form, the send/recv match achieves pairwise synch event
  - Other variants too
- Many overheads: copying, buffer management, protection

**DSM vs Message Passing**

High-level block diagram similar to distributed-memory SAS
- But comm. integrated at IO level, needn’t be into memory system
- Like networks of workstations (clusters), but tighter integration
- Easier to build than scalable SAS

MP model removed from basic hardware operations
- Library or OS intervention
Programming Model 3: Data Parallel

Single thread of control consisting of parallel operations. Parallel operations applied to all (or a defined subset) of a data structure, usually an array

- Communication is implicit in parallel operators
- Elegant and easy to understand and reason about
- Coordination is implicit – statements executed synchronously
- Similar to Matlab language for array operations

Drawbacks:
- Not all problems fit this model
- Difficult to map onto coarse-grained machines

\[
\begin{align*}
A &= \text{array of all data} \\
fA &= f(A) \\
s &= \text{sum}(fA)
\end{align*}
\]

SIMD Arch for DP Model

- Single Instruction Stream Multiple Data Stream (SIMD)
- A single “control processor” issues each instruction
- Array of many simple, cheap processors with little memory each processor executes the same instruction
  - Some processors may be turned off on some instructions
- Specialized and general communication, cheap global synchronization
**Application of Data Parallelism**

Each PE contains an employee record with his/her salary

\[
\text{If } \text{salary} > 100K \text{ then} \\
\quad \text{salary} = \text{salary} \times 1.05 \\
\text{else} \\
\quad \text{salary} = \text{salary} \times 1.10
\]

- Logically, the whole operation is a single step
- Some processors enabled for arithmetic operation, others disabled

Other examples:
- Finite differences, linear algebra, ...
- Document searching, graphics, image processing, ...

Some recent machines:
- Thinking Machines CM-1, CM-2 (and CM-5), Maspar MP-1 and MP-2.
- Built-in feature on modern CPU

---

**Vector Machine for DP Model**

Vector architectures are based on a single processor

- **Multiple functional units**
- **All performing the same operation**
- **Instructions may specific large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel**

Historically important
- **Overtaken by MPPs in the 90s**

Re-emerging in recent years
- **At a large scale in the Earth Simulator (NEC SX6) and Cray X1**
- **At a small sale in SIMD media extensions to microprocessors**
  - SSE, SSE2 (Intel: Pentium/IA64)
  - Altivec (IBM/Motorola/Apple: PowerPC)
  - VIS (Sun: Sparc)

Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn’t have to
Vector Processors

Vector instructions operate on a vector of elements
• These are specified as operations on vector registers

A supercomputer vector register holds ~32-64 elts
• The number of elements is larger than the amount of parallel hardware, called vector pipes or lanes, say 2-4

The hardware performs a full vector operation in
• \( \frac{\text{elements-per-vector-register}}{\text{pipes}} \)

Cray X1 Node

Cray X1 builds a larger “virtual vector”, called an MSP
• 4 SSPs (each a 2-pipe vector processor) make up an MSP
• Compiler will (try to) vectorize/parallelize across the MSP

Figure source J. Levesque, Cray
Cray X1: Parallel Vector Arch.

Cray combines several technologies in the X1
12.8 Gflop/s Vector processors (MSP)
Shared caches (unusual on earlier vector machines)
4 processor nodes sharing up to 64 GB of memory
Single System Image to 4096 Processors
Remote put/get between nodes (faster than MPI)

Earth Simulator Architecture

Parallel Vector Architecture
- High speed (vector) processors
- High memory bandwidth (vector architecture)
- Fast network (new crossbar switch)

Rearranging commodity parts can’t match this performance
Clusters have Arrived

What’s a Cluster?

Collection of independent computer systems working together as if a single system.
Coupled through a scalable, high bandwidth, low latency interconnect.

© C. Xu, 2001
**PC Clusters: Contributions of Beowulf**

An experiment in parallel computing systems

Established **vision** of low cost, high end computing

Demonstrated effectiveness of PC clusters for some (not all) classes of applications

Provided networking software

Conveyed findings to broad community (great PR)

Tutorials and book

Design standard to rally community!

Standards beget:
books, trained people, **Open source SW**

Adapted from Gordon Bell, presentation at Salishan, 2000

---

**Clusters of SMPs**

SMPs are the fastest commodity machine, so use them as a building block for a larger machine with a network

Common names:

• CLUMP = Cluster of SMPs
• Hierarchical machines, constellations

Most modern machines look like this:

• Millennium, IBM SPs, (not the t3e)... What is the right programming model???

• Treat machine as “flat”, always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
• Shared memory within one SMP, but message passing outside of an SMP.
Cluster of SMP Approach

A supercomputer is a stretched high-end server
Parallel system is built by assembling nodes that are modest size, commercial, SMP servers – just put more of them together

WSU CLUMP: Cluster of SMPs

© C. Xu, 2001
Computational/Data Grid

- Coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations
  - direct access to computers, sw, data, and other resources, rather than file exchange
  - Such sharing rules defines a set of individuals and/or institutions, which form a virtual organization
  - Examples of VOs: application service providers, storage service providers, cycle providers, etc

- Grid computing is to develop protocols, services, and tools for coordinated resource sharing and problem solving in VOs
  - Security solutions for management of credentials and policies
  - RM protocols and services for secure remote access
  - Information query protocols and services for configuration
  - Data management, etc

© C. Xu, 2001
**Top 500 Supercomputers**

Listing of the 500 most powerful computers in the world
- Yardstick: Rmax from LINPACK MPP benchmark
  \[ Ax=b, \text{ dense problem} \]
- Dense LU Factorization (dominated by matrix multiply)

Updated twice a year SC'xy in the States in November
- Meeting in Mannheim, Germany in June
- All data (and slides) available from [www.top500.org](http://www.top500.org)
- Also measures N-1/2 (size required to get ½ speed)

© C. Xu, 2001

---

**Fastest Computer Over Time**

<table>
<thead>
<tr>
<th>Year</th>
<th>GFlop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>0</td>
</tr>
<tr>
<td>1992</td>
<td>5</td>
</tr>
<tr>
<td>1994</td>
<td>10</td>
</tr>
<tr>
<td>1996</td>
<td>15</td>
</tr>
<tr>
<td>1998</td>
<td>20</td>
</tr>
<tr>
<td>2000</td>
<td>25</td>
</tr>
</tbody>
</table>

In 1980 a computation that took 1 full year to complete can now be done in 1 month!
In 1980 a computation that took 1 full year to complete can now be done in 4 days!

In 1980 a computation that took 1 full year to complete can today be done in 1 hour!
Chip Technology

Speedup and Amdahl’s Law

© C. Xu, 2001
Speedup due to Enhancement

Suppose a person wants to travel from city A to city B by city C. The routes from A to C are in mountains and the routes from C to B are in desert. The distances from A to C, and from C to B are 80 miles and 200 miles, respectively.

From A to C, walk at speed of 4 mph
From C to B, walk or drive (at speed of 100 mph)

Question 1: How long will it take for the entire trip walking?
   Answer: \( \frac{80}{4} + \frac{200}{4} = 70 \) hours

Question 2: How much faster from A to B by a car as opposed to walk?
   Answer: \( \frac{70}{\left(\frac{80}{4} + \frac{200}{100}\right)} = 3.18 \)

Speedup formula

\[
\text{Speedup}(E) = \frac{\text{ExTime(without E)}}{\text{ExTime(with E)}} = \frac{\text{Performance(with E)}}{\text{Performance(without E)}}
\]

Suppose that enhancement E accelerates a fraction \( F \) of the task by a factor \( S \) and the remainder of the task is unaffected then,

\[
\text{Speedup}(E) = \frac{\text{ExTime(without E)}}{\left((1 - F) + \frac{F}{S}\right) \times \text{ExTime(without E)}} = \frac{1}{\left(1 - F\right) + \frac{F}{S}} \leq \frac{1}{1 - F}
\]
**Example**

Suppose an enhancement runs 10 times faster than the original machine, but is only usable 40% of the time.

**Question:** what is the overall speedup?

**Answer:**

\[ \text{Fraction\_enhance} = 0.4 \]
\[ \text{Speedup\_enhanced} = 10 \]
\[ \text{Speedup\_overall} = \frac{1}{0.6 + 0.4/10} = 1.56 \]

--

**Speedup due to parallel computing**

Suppose only part of an application seems parallel.

**Amdahl’s law**

- Let \( f \) be the fraction of work done sequentially, so \((1-f)\) is fraction parallelizable.
- \( n \) = number of processors.

\[
\text{Speedup}(n) = \frac{\text{Time}(1)}{\text{Time}(n)} \\
\leq \frac{1}{f + \frac{(1-f)}{n}} \\
\leq \frac{1}{f} \\
\]

**Efficiency** = Speedup/n

Even if the parallel part speeds up perfectly, we may be limited by the sequential portion of code.
## Amdahl’s Law Implications

Amdahl’s law suggests that the speedup of a program running in parallel is limited by the time spent in the serial section of the program. For a serial section of 5% and 20 processors, the speedup is 10.26. According to Gustafson’s law, the speedup is 19.05.

![Graph showing speedup factors for different serial fractions and numbers of processors.](image)

## Gustafson’s Law

Parallel processing is to solve larger programs in a fixed time. Let $s$ be the serial execution time, and $p$ the execution time in parallel. The Scaled Speedup Factor is given by:

$$S_s(n) = \frac{s + np}{s + p} = s + np = n + (1 - n)s$$

Suppose a serial section of 5% and 20 processors.

- According to Amdahl’s law, the speedup is 10.26.
- According to Gustafson’s law, the speedup is 19.05.
**Overhead of Parallelism**

Given enough parallel work, this is the most significant barrier to getting desired speedup.

Parallelism overheads include:

- cost of starting a thread or process
- cost of communicating shared data
- cost of synchronizing
- extra (redundant) computation

Each of these can be in the range of milliseconds (= millions of flops) on some systems

**Tradeoff:** Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work.

---

**Parallel Programming Challenge**

Performance Tuning (Chemical Modeling)

---

© C. Xu, 2001
Summary

• Shared address space architecture and Programming model
  • Symmetric Multiprocessor (SMP)
  • Non-uniform memory architecture (NUMA), Distributed Shared Memory (DSA)
• Message passing architecture and programming model
• Data parallel architecture and programming model
• Convergence: Generic Architecture
• Speedup and Amdahl’s law