Parallel Computer Architecture

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

- Broad issues involved:
  - The concurrency and communication characteristics of parallel algorithms for a given computational problem.
  - Computing Resources and Computation Allocation:
    - The number of processing elements (PEs), computing power of each element and amount of physical memory used.
    - What portions of the computation and data are allocated to each PE.
  - Data access, Communication and Synchronization
    - How the elements cooperate and communicate.
    - How data is transmitted between processors.
    - Abstractions and primitives for cooperation.
  - Performance and Scalability
    - Maximize performance enhancement of parallelism: Speedup.
      - By minimizing parallelization overheads.
    - Scalability of performance to larger systems/problems.
The Need And Feasibility of Parallel Computing

- **Application demands:** More computing cycles needed:
  - *Scientific computing:* CFD, Biology, Chemistry, Physics, ...
  - *General-purpose computing:* Video, Graphics, CAD, Databases, Transaction Processing, Gaming...
  - Mainstream multithreaded programs, are similar to parallel programs

- **Technology Trends**
  - Number of transistors on chip growing rapidly. Clock rates expected to go up but only slowly.

- **Architecture Trends**
  - Instruction-level parallelism is valuable but limited.
  - Coarser-level parallelism, as in multiprocessor systems is the most viable approach to further improve performance.

- **Economics:**
  - The increased utilization of commodity of-the-shelf (COTS) components in high performance parallel computing systems instead of costly custom components used in traditional supercomputers leading to much lower parallel system cost.
    - Today’s microprocessors offer high-performance and have multiprocessor support eliminating the need for designing expensive custom PEs
    - Commercial System Area Networks (SANs) offer an alternative to custom more costly networks
Scientific Computing Demands

(Memory Requirement)

Storage requirement

1 TB
100 GB
10 GB
1 GB
100 MB
10 MB

Computational performance requirement

100 MFLOPS
1 GFLOPS
10 GFLOPS
100 GFLOPS
1 TFLOPS

Grand Challenge problems
- Global change
- Human genome
- Fluid turbulence
- Vehicle dynamics
- Ocean circulation
- Viscous fluid dynamics
- Superconductor modeling
- Quantum chromo dynamics
- Vision

Structural biology

Vehicle signature

Pharmaceutical design

72-hour weather

48-hour weather

3D plasma modeling

Chemical dynamics

2D airfoil

Oil reservoir modeling
Scientific Supercomputing Trends

• Proving ground and driver for innovative architecture and advanced computing techniques:
  – Market is much smaller relative to commercial segment
  – Dominated by vector machines starting in the 70s through the 80s
  – Meanwhile, microprocessors have made huge gains in floating-point performance
    • High clock rates.
    • Pipelined floating point units.
    • Instruction-level parallelism.
    • Effective use of caches.

• Large-scale multiprocessors and computer clusters are replacing vector supercomputers
The microprocessor is currently the most natural building block for multiprocessor systems in terms of cost and performance.
General Technology Trends

- Microprocessor performance increases 50% - 100% per year
- Transistor count doubles every 3 years
- DRAM size quadruples every 3 years
Clock Frequency Growth Rate

- Currently increasing 30% per year
Transistor Count Growth Rate

- One billion transistors on chip by early 2004
- Transistor count grows much faster than clock rate
  - Currently 40% per year
Parallelism in Microprocessor VLSI Generations

Bit-level parallelism

Instruction-level

Thread-level (?)

Transistors


1,000,000

10,000,000

100,000,000

SMT: e.g. Intel’s Hyper-threading

i4004

i8008

i8086

i80286

i80386

i8086

R2000

Pentium

R3000

R10000

Parallelism in Microprocessor VLSI Generations
Uniprocessor Attributes to Performance

- Performance benchmarking is program-mix dependent.
- Ideal performance requires a perfect machine/program match.
- Performance measures:
  - Cycles per instruction (CPI)
  - Total CPU time = \( T = C \times \tau = C / f = I_c \times CPI \times \tau \)
    \[ = I_c \times (p + m \times k) \times \tau \]

  \( I_c \) = Instruction count \quad \tau = CPU \ cycle \ time
  \( p \) = Instruction decode cycles
  \( m \) = Memory cycles \quad \( k \) = Ratio between memory/processor cycles
  \( C \) = Total program clock cycles \quad \( f \) = clock rate

  - MIPS Rate = \( I_c / (T \times 10^6) = f / (CPI \times 10^6) = f \times I_c / (C \times 10^6) \)

  - Throughput Rate: \( W_p = f / (I_c \times CPI) = (MIPS) \times 10^6 / I_c \)

- Performance factors: \( (I_c, p, m, k, \tau) \) are influenced by: instruction-set architecture, compiler design, CPU implementation and control, cache and memory hierarchy and program instruction mix and instruction dependencies.
Raw Uniprocessor Performance: LINPACK

LINPACK (MFLOPS)

CRAY n = 1,000
CRAY n = 100
Micro n = 1,000
Micro n = 100

Vector Processors
CRAY
Xmp/14se
Xmp/416
Ymp
C90
DEC 8200
IBM Power2/990
MIPS R4400
DEC Alpha
HP9000/735
DEC Alpha AXP
HP 9000/750
IBM RS6000/540

Microprocessors
CRAY 1s
Xmp/14se
Micro n = 100

DEC Alpha
HP9000/750
IBM RS6000/540
MIPS M/2000
MIPS M/120
Sun 4/260

Raw Parallel Performance: LINPACK

<table>
<thead>
<tr>
<th>Linpack (GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCray peak</td>
</tr>
<tr>
<td>lMPP peak</td>
</tr>
<tr>
<td>Xmp /416(4)</td>
</tr>
<tr>
<td>Ymp/832(8)</td>
</tr>
<tr>
<td>CM-2</td>
</tr>
<tr>
<td>CM-200</td>
</tr>
<tr>
<td>CM-5</td>
</tr>
<tr>
<td>Paragon XP/S MP</td>
</tr>
<tr>
<td>(6768)</td>
</tr>
<tr>
<td>Paragon XP/S</td>
</tr>
<tr>
<td>(1024)</td>
</tr>
<tr>
<td>T3D</td>
</tr>
<tr>
<td>T932(32)</td>
</tr>
<tr>
<td>Paragon XP/S MP</td>
</tr>
<tr>
<td>(1024)</td>
</tr>
<tr>
<td>ASCI Red</td>
</tr>
<tr>
<td>ASII Red</td>
</tr>
</tbody>
</table>

1985 1987 1989 1991 1993 1995 1996
LINPAK Performance Trends

Uniprocessor Performance

Parallel System Performance

EECC756 - Shaaban

#13  lec # 1  Spring 2003  3-11-2003
Computer System Peak FLOP Rating
History/Near Future

Doubling time = 1.5 yr.

1E+16
1E+14
1E+12
1E+10
1E+8
1E+6
1E+4
1E+2


Peak Speed (flops)
Petaflop

Teraflop
The Goal of Parallel Processing

• Goal of applications in using parallel machines:
  Maximize Speedup over single processor performance

\[
\text{Speedup (p processors)} = \frac{\text{Performance (p processors)}}{\text{Performance (1 processor)}}
\]

• For a fixed problem size (input data set),
  performance = 1/time

\[
\text{Speedup}_{\text{fixed problem}} (p \text{ processors}) = \frac{\text{Time (1 processor)}}{\text{Time (p processors)}}
\]

• Ideal speedup = number of processors = p
  Very hard to achieve
The Goal of Parallel Processing

- Parallel processing goal is to maximize parallel speedup:

  \[
  \text{Speedup} = \frac{\text{Time}(1)}{\text{Time}(p)} \leq \frac{\text{Sequential Work on one processor}}{\text{Max} (\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})} \]

  - Ideal Speedup = \( p \) number of processors
    - Very hard to achieve: Implies no parallelization overheads and perfect load balance among all processors.
  - Maximize parallel speedup by:
    - Balancing computations on processors (every processor does the same amount of work).
    - Minimizing communication cost and other overheads associated with each step of parallel program creation and execution.
  - Performance Scalability:
    - Achieve a good speedup for the parallel application on the parallel architecture as problem size and machine size (number of processors) are increased.
Elements of Parallel Computing

- Computing Problems
- Algorithms and Data Structures
- High-level Languages
- Mapping
  - Programming
  - Binding (Compile, Load)
- Performance Evaluation
- Hardware Architecture
- Operating System
- Applications Software
Elements of Parallel Computing

1 Computing Problems:
   - Numerical Computing: Science and technology numerical problems demand intensive integer and floating point computations.
   - Logical Reasoning: Artificial intelligence (AI) demand logic inferences and symbolic manipulations and large space searches.

2 Algorithms and Data Structures
   - Special algorithms and data structures are needed to specify the computations and communication present in computing problems.
   - Most numerical algorithms are deterministic using regular data structures.
   - Symbolic processing may use heuristics or non-deterministic searches.
   - Parallel algorithm development requires interdisciplinary interaction.
Elements of Parallel Computing

3 Hardware Resources

– Processors, memory, and peripheral devices form the hardware core of a computer system.

– Processor instruction set, processor connectivity, memory organization, influence the system architecture.

4 Operating Systems

– Manages the allocation of resources to running processes.

– Mapping to match algorithmic structures with hardware architecture and vice versa: processor scheduling, memory mapping, interprocessor communication.

– Parallelism exploitation at: algorithm design, program writing, compilation, and run time.
Elements of Parallel Computing

5 System Software Support
- Needed for the development of efficient programs in high-level languages (HLLs.)
- Assemblers, loaders.
- Portable parallel programming languages
- User interfaces and tools.

6 Compiler Support
- Preprocessor compiler: Sequential compiler and low-level library of the target parallel computer.
- Precompiler: Some program flow analysis, dependence checking, limited optimizations for parallelism detection.
- Parallelizing compiler: Can automatically detect parallelism in source code and transform sequential code into parallel constructs.
Approaches to Parallel Programming

Programmer

Source code written in sequential languages C, C++ FORTRAN, LISP ..

Parallelizing compiler

Parallel object code

Execution by runtime system

(a) Implicit Parallelism

Programmer

Source code written in concurrent dialects of C, C++ FORTRAN, LISP ..

Concurrency preserving compiler

Concurrent object code

Execution by runtime system

(b) Explicit Parallelism
Factors Affecting Parallel System Performance

• **Parallel Algorithm Related:**
  – Available concurrency and profile, grain, uniformity, patterns.
  – Required communication/synchronization, uniformity and patterns.
  – Data size requirements.
  – Communication to computation ratio.

• **Parallel program Related:**
  – Programming model used.
  – Resulting data/code memory requirements, locality and working set characteristics.
  – Parallel task grain size.
  – Assignment: Dynamic or static.
  – Cost of communication/synchronization.

• **Hardware/Architecture related:**
  – Total CPU computational power available.
  – Types of computation modes supported.
  – Shared address space Vs. message passing.
  – Communication network characteristics (topology, bandwidth, latency)
  – Memory hierarchy properties.
Evolution of Computer Architecture

Scalar
- Sequential
- Lookahead

I/E Overlap
- Functional Parallelism
  - Multiple Func. Units
  - Pipeline
    - Implicit Vector
    - Explicit Vector
      - Memory-to-Memory
      - Register-to-Register

SIMD
- Single Instruction stream over Multiple Data streams
- SIMD
  - Associative Processor
  - Processor Array

MIMD
- Multiple Instruction streams over Multiple Data streams
- MIMD
  - Multicomputer
  - Multiprocessor

I/E: Instruction Fetch and Execute
SIMD: Single Instruction stream over Multiple Data streams
MIMD: Multiple Instruction streams over Multiple Data streams

Massively Parallel Processors (MPPs)
Historically, parallel architectures tied to programming models

- Divergent architectures, with no predictable pattern of growth.
Parallel Programming Models

- Programming methodology used in coding applications
- Specifies communication and synchronization

Examples:

- **Multiprogramming:**
  No communication or synchronization at program level. A number of independent programs.

- **Shared memory address space:**
  Parallel program threads or tasks communicate using a shared memory address space

- **Message passing:**
  Explicit point to point communication is used between parallel program tasks.

- **Data parallel:**
  More regimented, global actions on data
  - Can be implemented with shared address space or message passing
Flynn’s 1972 Classification of Computer Architecture

• Single Instruction stream over a Single Data stream (SISD): Conventional sequential machines.

• Single Instruction stream over Multiple Data streams (SIMD): Vector computers, array of synchronized processing elements.

• Multiple Instruction streams and a Single Data stream (MISD): Systolic arrays for pipelined execution.

• Multiple Instruction streams over Multiple Data streams (MIMD): Parallel computers:
  • Shared memory multiprocessors.
  • Multicomputers: Unshared distributed memory, message-passing used instead.
Flynn’s Classification of Computer Architecture

Fig. 1.3 page 12 in
Current Trends In Parallel Architectures

• The extension of “computer architecture” to support communication and cooperation:
  – OLD: Instruction Set Architecture
  – NEW: Communication Architecture

• Defines:
  – Critical abstractions, boundaries, and primitives (interfaces)
  – Organizational structures that implement interfaces (hardware or software)

• Compilers, libraries and OS are important bridges today
# Modern Parallel Architecture

## Layered Framework

<table>
<thead>
<tr>
<th>CAD</th>
<th>Database</th>
<th>Scientific modeling</th>
<th>Parallel applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprogramming</td>
<td>Shared address</td>
<td>Message passing</td>
<td>Data parallel</td>
</tr>
</tbody>
</table>

- **Compilation or library**
- **Operating systems support**
- **Communication hardware**
- **Physical communication medium**

- Programming models
- Communication abstraction
- User/system boundary
- Hardware/software boundary
Shared Address Space Parallel Architectures

• Any processor can directly reference any memory location
  – Communication occurs implicitly as result of loads and stores

• Convenient:
  – Location transparency
  – Similar programming model to time-sharing in uniprocessors
    • Except processes run on different processors
    • Good throughput on multiprogrammed workloads

• Naturally provided on a wide range of platforms
  – 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
Shared Address Space (SAS) Parallel Programming Model

- Process: virtual address space plus one or more threads of control
- Portions of address spaces of processes are shared

- Writes to shared address visible to other threads (in other processes too)
- Natural extension of the uniprocessor model:
  - Conventional memory operations used for communication
  - Special atomic operations needed for synchronization
- OS uses shared memory to coordinate processes
Models of Shared-Memory Multiprocessors

• The Uniform Memory Access (UMA) Model:
  – The physical memory is shared by all processors.
  – All processors have equal access to all memory addresses.
  – Also referred to as Symmetric Memory Processors (SMPs).

• Distributed memory or Nonuniform Memory Access (NUMA) Model:
  – Shared memory is physically distributed locally among processors. Access to remote memory is higher.

• The Cache-Only Memory Architecture (COMA) Model:
  – A special case of a NUMA machine where all distributed main memory is converted to caches.
  – No memory hierarchy at each processor.
Models of Shared-Memory Multiprocessors

Uniform Memory Access (UMA) Model
or Symmetric Memory Processors (SMPs).

Interconnect:
Bus, Crossbar, Multistage network
P: Processor
M: Memory
C: Cache
D: Cache directory

Distributed memory or
Nonuniform Memory Access (NUMA) Model

Cache-Only Memory Architecture (COMA)
Uniform Memory Access Example: Intel Pentium Pro Quad

- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth
Uniform Memory Access 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
Distributed Shared-Memory Multiprocessor System Example: Cray T3E

- Scale up to 1024 processors, 480MB/s links
- Memory controller generates communication requests for nonlocal references
- No hardware mechanism for coherence (SGI Origin etc. provide this)
Message-Passing Multicomputers

• Comprised of multiple autonomous computers (nodes) connected via a suitable network.

• Each node consists of one or more processors, local memory, attached storage and I/O peripherals.

• Local memory is only accessible by local processors in a node.

• Inter-node communication is carried out by message passing through the connection network

• Process communication achieved using a message-passing programming environment.
  – Programming model more removed from basic hardware operations

• Include:
  – A number of commercial Massively Parallel Processor systems (MPPs).
  – Computer clusters that utilize commodity of-the-shelf (COTS) components.
Message-Passing Abstraction

- Send specifies buffer to be transmitted and receiving process
- Receive specifies sending process and application storage to receive into
- Memory to memory copy possible, 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/receive match achieves pairwise synch event
- Many overheads: copying, buffer management, protection
Message-Passing Example: IBM SP-2

- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (bandwidth limited by I/O bus)
Message-Passing Example: Intel Paragon

Sandia’s Intel Paragon XP/S-based Supercomputer

2D grid network with processing node attached to every switch

Memory bus (64-bit, 50 MHz)

ni

DMA

Driver

Mem ctrl

4-way interleaved DRAM

Intel Paragon node

i860

L1 $

i860

L1 $
Message-Passing Programming Tools

- Message-passing programming environments include:
  - Message Passing Interface (MPI):
    - Provides a standard for writing concurrent message-passing programs.
    - MPI implementations include parallel libraries used by existing programming languages.
  - Parallel Virtual Machine (PVM):
    - Enables a collection of heterogeneous computers to used as a coherent and flexible concurrent computational resource.
    - PVM support software executes on each machine in a user-configurable pool, and provides a computational environment of concurrent applications.
    - User programs written for example in C, Fortran or Java are provided access to PVM through the use of calls to PVM library routines.
Data Parallel Systems

SIMD in Flynn taxonomy

- Programming model
  - Operations performed in parallel on each element of data structure
  - Logically single thread of control, performs sequential or parallel steps
  - Conceptually, a processor is associated with each data element

- Architectural model
  - Array of many simple, cheap processors each with little memory
    - Processors don’t sequence through instructions
  - Attached to a control processor that issues instructions
  - Specialized and general communication, cheap global synchronization

- Example machines:
  - Thinking Machines CM-1, CM-2 (and CM-5)
  - Maspar MP-1 and MP-2,
Dataflow Architectures

• Represent computation as a graph of essential dependences
  – Logical processor at each node, activated by availability of operands
  – Message (tokens) carrying tag of next instruction sent to next processor
  – Tag compared with others in matching store; match fires execution

Research Dataflow machine prototypes include:
• The MIT Tagged Architecture
• The Manchester Dataflow Machine

\[
\begin{align*}
a &= (b + 1) \times (b - c) \\
d &= c \times e \\
f &= a \times d
\end{align*}
\]
Systolic Architectures

- Replace single processor with an array of regular processing elements
- Orchestrate data flow for high throughput with less memory access

- Different from pipelining
  - Nonlinear array structure, multidirection data flow, each PE may have (small) local instruction and data memory
- Different from SIMD: each PE may do something different
- Initial motivation: VLSI enables inexpensive special-purpose chips
- Represent algorithms directly by chips connected in regular pattern
Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

Alignments in time

Rows of A

<table>
<thead>
<tr>
<th>a0,2</th>
<th>a0,1</th>
<th>a0,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1,2</td>
<td>a1,1</td>
<td>a1,0</td>
</tr>
<tr>
<td>a2,2</td>
<td>a2,1</td>
<td>a2,0</td>
</tr>
</tbody>
</table>

T = 0

Columns of B

b2,2
b2,1 —— b1,2
b2,0 —— b1,1 —— b0,2
b1,0 —— b0,1
b0,0

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/
Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

Alignments in time

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/
Systolic Array Example: 3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

Alignments in time

T = 2

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/
Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

Alignments in time

$$T = 3$$

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/
Systolic Array Example: 3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

Alignments in time

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/
Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

Alignments in time

\[ T = 5 \]

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/
Systolic Array Example: 3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

Alignments in time

\[ T = 6 \]

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/
Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product

Alignments in time

\[ T = 7 \]

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/