Overview of Cluster Computing and Parallel Programming

Prof. Chengzhong Xu
Department of Electrical and Computer Engg.
Wayne State University
czxu@ece.eng.wayne.edu

© C. Xu 1999

What is Parallel Architecture?

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?

• 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?
Commercial Computing

Servers rely more on parallelism for high end
- Database and Web servers for online transaction processing
- Decision support
- Data mining and data warehousing
- Financial modeling

- Scale not so large, but use much more wide-spread
- Computational power determines scale of business that can be handled. E.g. AOL story in 1997

© C. Xu 1999
**Generic Organization**

A generic modern multiprocessor

Node: processor(s), memory system, plus *communication assist*
- Network interface and communication controller
- Scalable network
- Convergence allows lots of innovation, now within framework
  - Integration of assist with node, what operations, how efficiently...

**Shared Address Space Architecture**

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

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 at least 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
**Message Passing Architectures**

Complete computer as building block, including I/O
- Communication via explicit I/O operations

Programming model: directly access only private address space (local memory), comm. via explicit messages (send/receive)

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

Programming model more removed from basic hardware operations
- Library or OS intervention

---

**Example: IBM SP-2**

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

© C. Xu 1999
Example Intel Paragon

Clusters have Arrived

Goal: Take a cluster of commodity workstations and make them look like a supercomputer
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.

Why Clusters?

Unlimited Capacity
- Network is a Computer
- Internet is supercomputer

High Availability
- Autonomy of processing nodes

Good Scalability

Cost-effectiveness
- Commodity
- Mass volume
- Rule of Thumb: 10% cost reduction for each volume doubling
Cluster Prehistory: Tandem NonStop

Early (1974) foray into transparent fault tolerance through redundancy
- Mirror everything (CPU, storage, power supplies…), can tolerate any single fault (later: processor duplexing)
- “Hot standby” process pair approach
- What’s the difference between high availability and fault tolerance?

Noteworthy
- “Shared nothing”--why?
- Performance and efficiency costs?
- Later evolved into Tandem Himalaya, which used clustering for both higher performance and higher availability

Pre-NOW Clustering in the 90’s

IBM Parallel Sysplex and DEC OpenVMS
- Targeted at conservative (read: mainframe) customers
- Shared disks allowed under both (why?)
- All devices have cluster-wide names (shared everything?)
- 1500 installations of Sysplex, 25,000 of OpenVMS Cluster

Programming the clusters
- All System/390 and/or VAX VMS subsystems were rewritten to be cluster-aware
- OpenVMS: cluster support exists even in single-node OS!
- An advantage of locking into proprietary interfaces

What about fault tolerance?
Traditional Availability Clusters

VAX Clusters(85) => IBM sysplex(90)
=> Microsoft WolfPack (95), SUN Moon (98)

The Case For NOW: MPP’s a Near Miss

uproc perf. improves 50% / yr (4%/month)
  • 1 year lag: WS = 1.50 MPP node perf.
  • 2 year lag: WS = 2.25 MPP node perf.

No economy of scale in 100s => +$

Software incompatibility (OS & apps) => +$$$$

More efficient utilization of compute resources (statistical multiplexing)

“Scale makes availability affordable” (Pfister)

Which of these do commodity clusters actually solve?
**Philosophy: “Systems of Systems”**

Higher Order systems research: aggressively use off-the-shelf hardware and OS software

Advantages:
- easier to track technological advances
- less development time
- easier to transfer technology (reduce lag)

New challenges (“the case against NOW”):
- maintaining performance goals
- system is changing underneath you
- underlying system has other people's bugs
- underlying system is poorly documented

---

**Clustering and Internet Services**

Aggregate capacity
- TB of disk storage, THz of compute power (if we can harness in parallel!)

Redundancy
- Partial failure behavior: only small fractional degradation from loss of one node
- Availability: industry average across “large” sites during 1998 holiday season was 97.2% availability (source: CyberAtlas)
- Compare: mission-critical systems have “four nines” (99.99%)
Spike Absorption

Internet traffic is self-similar
- Bursty at all granularities less than about 24 hours
- What’s bad about burstiness?

Spike Absorption
- Diurnal variation: peak vs. average demand typically a factor of 3 or more
- Starr Report: CNN peaked at 20M hits/hour (compared to usual peak of 12M hits/hour; that’s +66%)

Really the holy grail: capacity on demand
- Is this realistic?

Clustering and Internet Workloads

Internet vs. “traditional” workloads
- e.g. Database workloads (TPC benchmarks)
- e.g. traditional scientific codes (matrix multiply, simulated annealing and related simulations, etc.)

Some characteristic differences
- Read mostly
- Quality of service (best-effort vs. guarantees)
- Task granularity
- “Embarrassingly parallel”
- …but are they balanced? (we’ll return to this later)
Meeting the Cluster Challenges

Software & programming models
Partial failure and application semantics
System administration

System Administration on a Cluster

Total cost of ownership (TCO) way high for clusters
• Median sysadmin cost per machine per year (1996): ~$700
• Cost of a headless workstation today: ~$1500

Previous Solutions
• Pay someone to watch
• Ignore or wait for someone to complain
• “Shell Scripts From Hell” (not general → vast repeated work)

Need an extensible and scalable way to automate the gathering, analysis, and presentation of data
System Administration, cont’d.

Extensible Scalable Monitoring For Clusters of Computers (Anderson & Patterson, UC Berkeley)

Relational tables allow properties & queries of interest to evolve as the cluster evolves

Extensive visualization support allows humans to make sense of masses of data

Multiple levels of caching decouple data collection from aggregation

Data updates can be “pulled” on demand or triggered by push

Visualizing Data: Example

Display aggregates of various interesting machine properties on the NOW’s

Note use of aggregation, color
Case Study: The Berkeley NOW

History of an early research cluster
• NOW-0: four HP-735’s
• NOW-1: 32 headless Sparc-10’s and Sparc-20’s
• NOW-2: 100 UltraSparc 1’s, Myrinet interconnect
  • inktomi.berkeley.edu: four Sparc-10’s
    – www.hotbot.com: 160 Ultra’s, 200 CPU’s total
• NOW-3: eight 4-way SMP’s

Myrinet interconnection
• In addition to commodity switched Ethernet
• Originally Sparc SBus, now available on PCIbus

The Adventures of NOW: Applications

AlphaSort: 8.41 GB in one minute, 95 UltraSparcs
• runner up: Ordinal Systems nSort on SGI Origin, 5 GB)
• pre-1997 record, 1.6 GB on an SGI Challenge

40-bit DES key crack in 3.5 hours
• “NOW+”: headless and some headed machines

inktomi.berkeley.edu (now inktomi.com)
• now fastest search engine, largest aggregate capacity

TranSend proxy & Top Gun Wingman Pilot browser
• ~15,000 users, 3-10 machines
Cluster Summary

Clusters have potential advantages…but serious challenges to achieving them in practice
  • Kind of like Network Computers?

Everyone and their brother is now selling a cluster
  • Who’s selling a system, and who’s selling a promise?
  • Can clustering be sold as a “secret sauce”? 

Programming for Parallel Computers
Steps in Creating a Parallel Program

4 steps: Decomposition, Assignment, Orchestration, Mapping
- Done by programmer or system software (compiler, runtime, ...)
- Issues are the same, so assume programmer does it all explicitly

Some Important Concepts

Task:
- Arbitrary piece of undecomposed work in parallel computation
- Executed sequentially; concurrency is only across tasks
- E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
- Fine-grained versus coarse-grained tasks

Process (thread):
- Abstract entity that performs the tasks assigned to processes
- Processes communicate and synchronize to perform their tasks

Processor:
- Physical engine on which process executes
- Processes virtualize machine to programmer
  - first write program in terms of processes, then map to processors
High-level Goals

High performance (speedup over sequential program)

Table 2.1 Steps in the Parallelization Process and Their Goals

<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture-Dependent?</th>
<th>Major Performance Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Mostly no</td>
<td>Expose enough concurrency but not too much</td>
</tr>
<tr>
<td>Assignment</td>
<td>Mostly no</td>
<td>Balance workload</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication volume</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Reduce noninherent communication via data locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication and synchronization cost as seen by the processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce serialization at shared resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Schedule tasks to satisfy dependences early</td>
</tr>
<tr>
<td>Mapping</td>
<td>Yes</td>
<td>Put related processes on the same processor if necessary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploit locality in network topology</td>
</tr>
</tbody>
</table>

© C. Xu 1999

What Parallel Programs Look Like
**Example: Gauss Seidel Grid Solver**

Expression for updating each interior point:


- Simplified version of solver in Ocean simulation

---

**Decomposition**

- Simple way to identify concurrency is to look at loop iterations — *dependence analysis*; if not enough concur., then look further
- Not much concurrency here at this level (all loops *sequential*)
- Examine fundamental dependences, ignoring loop structure
**Exploit Application Knowledge**

- Reorder grid traversal: red-black ordering

- Red sweep and black sweep are each fully parallel:
- Global synch between them (conservative but convenient)
- Ocean uses red-black; we use simpler, asynchronous one to illustrate
  - no red-black, simply ignore dependences within sweep
  - sequential order same as original, parallel program *deterministic*

**Assignment**

- **Static assignments** (given decomposition into rows)
  - block assignment: Row $i$ is assigned to process $\left\lfloor \frac{i}{p} \right\rfloor$

- Cyclic assignment: Process $i$ is assigned rows $i$, $i+p$, $i+2p$

Block cyclic:
Deciding How to Manage Concurrency

*Static* versus *Dynamic* techniques

**Static:**
- Algorithmic assignment based on input; won’t change
- Low runtime overhead
- Computation must be predictable
- Preferable when applicable (except in multiprogrammed/heterogeneous environment)

**Dynamic:**
- Adapt at runtime to balance load
- Can increase communication and reduce locality
- Can increase task management overheads

---

**Dynamic Assignment**

Dynamic Tasking:

- E.g. “Self-scheduling” of loop iterations
  - Get a row index, work on the row, then get a new ros, and so on.
- Deal with unpredictability in program or environment (e.g. Branch and bound optimization )
  - computation, communication, and memory system interactions
  - multiprogramming and heterogeneity
  - used by runtime systems and OS too
- Pool of tasks; take and add tasks until done
**Dynamic Tasking with Task Queues**

Centralized versus distributed queues

Task stealing with distributed queues

- Can compromise comm and locality, and increase synchronization
- Whom to steal from, how many tasks to steal, ...
- Termination detection
- Maximum imbalance related to size of task

---

**Meeting the Cluster Challenges**

Software & programming models

Partial failure and application semantics

System administration
Software Challenges

Message-passing & Active Messages
Shared memory: Network RAM
  • CC-NUMA, Software DSM: *Anyone who thinks cache misses can take milliseconds is an idiot.* (Paraphrasing Larry McVoy at OSDI 96)
MP vs SM a long-standing religious debate
Arbitrary object migration (“network transparency”)
  • What are the problems with this?
  • Hints: RPC, checkpointing, residual state

Partial Failure Management

What does *partial failure* mean for…
  • a transactional database?
  • A read-only database striped across cluster nodes?
  • A compute-intensive shared service?
What are appropriate “partial failure abstractions”?
  • Incomplete/imprecise results?
  • Longer latency?
Software Challenges, Again?

Real issue: we have to think differently about programming…
  • …to harness clusters?
  • …to get decent failure semantics?
  • …to really exploit software modularity?

Traditional uniprocessor programming idioms/models don’t seem to scale up to clusters

Question: Is there a “natural to use” cluster model that scales down to uniprocessors?
  • If so, is it general or application-specific?
  • What would be the obstacles to adopting such a model?