Outline

- Drivers of Parallel Computing
- Trends in “Supercomputers” for Scientific Computing
- Evolution and Convergence of Parallel Architectures
- Fundamental Issues in Programming Models and Architecture

History

Historically, parallel architectures tied to programming models

- Divergent architectures, with no predictable pattern of growth.

• Uncertainty of direction paralyzed parallel software development!

Modern Layered Framework

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 (hw or sw)

Compilers, libraries and OS are important bridges between application and architecture today
**Parallel Programming Model**

What the programmer uses in writing applications

Specifies communication and synchronization

Examples:
- Multiprogramming: no communication or synch. at program level
- *Shared address space*: like bulletin board
- *Message passing*: like letters or phone calls, explicit point to point
- *Data parallel*: more regimented, global actions on data
  - Implemented with shared address space or message passing

**Communication Abstraction**

User level communication primitives provided by system

- Realizes the programming model
- Mapping exists between language primitives of programming model and these primitives

Supported directly by hw, or via OS, or via user sw

Lot of debate about what to support in sw and gap between layers

Today:
- Hw/sw interface tends to be flat, i.e. complexity roughly uniform
- Compilers and software play important roles as bridges today
- Technology trends exert strong influence

Result is convergence in organizational structure
- Relatively simple, general purpose communication primitives

**Communication Architecture**

= User/System Interface + Implementation

User/System Interface:
- Comm. primitives exposed to user-level by hw and system-level sw
- (May be additional user-level software between this and prog model)

Implementation:
- Organizational structures that implement the primitives: hw or OS
- How optimized are they? How integrated into processing node?
- Structure of network

Goals:
- Performance
- Broad applicability
- Programmability
- Scalability
- Low Cost

**Evolution of Architectural Models**

Historically, machines were tailored to programming models

- Programming model, communication abstraction, and machine organization lumped together as the “architecture”

Understanding their evolution helps understand convergence
- Identify core concepts

Evolution of Architectural Models:
- Shared Address Space (SAS)
- Message Passing
- Data Parallel
- Others (won’t discuss): Dataflow, Systolic Arrays

Examine programming model, motivation, and convergence
Shared Address Space 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 on uniprocessors
  – Except processes run on different processors
  – Good throughput on multiprogrammed 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

Shared Address Space Model

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

Communication Hardware for SAS

• Also natural extension of uniprocessor
• Already have processor, one or more memory modules and I/O controllers connected by hardware interconnect of some sort
• Memory capacity increased by adding modules, I/O by controllers

Add processors for processing!

History of SAS Architecture

“Mainframe” approach
• Motivated by multiprogramming
• Extends crossbar used for mem bw and I/O
• Originally processor cost limited to small
  – later, cost of crossbar
• Bandwidth scales with \( p \)
• High incremental cost; use multistage instead

“Minicomputer” approach
• Almost all microprocessor systems have bus
• Motivated by multiprogramming, TP
• Used heavily for parallel computing
• Called symmetric multiprocessor (SMP)
• Latency larger than for uniprocessor
• Bus is bandwidth bottleneck
  – caching is key: coherence problem
• Low incremental cost
Example: Intel Pentium Pro Quad

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

Example: SUN Enterprise

- Memory on processor cards themselves
  - 16 cards of either type: processors + memory, or I/O
- But all memory accessed over bus, so symmetric
- Higher bandwidth, higher latency bus

Scaling Up

- Problem is interconnect: cost (crossbar) or bandwidth (bus)
- Dance-hall: bandwidth still scalable, but lower cost than crossbar
  - Latencies to memory uniform, but uniformly large
- Distributed memory or non-uniform memory access (NUMA)
  - Construct shared address space out of simple message transactions across a general-purpose network (e.g. read-request, read-response)
- Caching shared (particularly nonlocal) data?

Example: Cray T3E

- Scale up to 1024 processors, 480MB/s links
- Memory controller generates comm. request for nonlocal references
  - Communication architecture tightly integrated into node
- No hardware mechanism for coherence (SGI Origin etc. provide this)
Caches and Cache Coherence

Caches play key role in all cases
• Reduce average data access time
• Reduce bandwidth demands placed on shared interconnect

But private processor caches create a problem
• Copies of a variable can be present in multiple caches
• A write by one processor may not become visible to others
  – They’ll keep accessing stale value in their caches
• Cache coherence problem
• Need to take actions to ensure visibility

Example Cache Coherence Problem

• Processors see different values for u after event 3
• With write back caches, value written back to memory depends on
  happenstance of which cache flushes or writes back value when
  – Processes accessing main memory may see very stale value
• Unacceptable to programs, and frequent!

Cache Coherence

Reading a location should return latest value written (by any process)

Easy in uniprocessors
• Except for I/O: coherence between I/O devices and processors
• But infrequent, so software solutions work

Would like same to hold when processes run on different processors
• E.g. as if the processes were interleaved on a uniprocessor

But coherence problem much more critical in multiprocessors
• Pervasive and performance-critical
• A very basic design issue in supporting the prog. model effectively

It’s worse than that: what is the “latest” value with indept. processes?
• Memory consistency models

SGI Origin 2000

Hub chip provides memory control, communication and cache coherence support
• Plus I/O communication etc
Shared Address Space Machines Today

- Bus-based, cache coherent at small scale
- Distributed memory, cache-coherent at larger scale
  - Without cache coherence, are essentially (fast) message passing systems
- Clusters of these at even larger scale

Message-Passing Programming Model

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

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. needn’t be integrated into memory system, only I/O
  - History of tighter integration, evolving to spectrum incl. clusters
  - Easier to build than scalable SAS
  - Can use clusters of PCs or SMPs on a LAN

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

Evolution of Message-Passing Machines

Early machines: FIFO on each link
  - Hw close to prog. Model; synchronous ops
  - Replaced by DMA, enabling non-blocking ops
    - Buffered by system at destination until recv

Diminishing role of topology
  - Store&forward routing: topology important
  - Introduction of pipelined routing made it less so
  - Cost is in node-network interface
  - Simplifies programming
**Example: IBM SP-2**

- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (bw limited by I/O bus)
  - Doesn’t need to see memory references

**MicroChannel bus**

**I/O**

**DMA**

**DRAM**

**IBM SP-2 node**

**Power 2 CPU**

**Memory controller**

4-way interleaved DRAM

**i860**

**General interconnection network formed from 8-port switches**

**2D grid network with processing node attached to every switch**

**Example Intel Paragon**

- Network interface integrated in memory bus, for performance

**Intel Paragon node**

**Memory bus (64-bit, 50 MHz)**

**4-way interleaved DRAM**

**Intel Paragon node**

**Send/recv supported on SAS machines via buffers**

**Can construct global address space on MP using hashing**

**Software shared memory (e.g. using pages as units of comm.)**

**Hardware organization converging too**

- Tighter NI integration even for MP (low-latency, high-bandwidth)
- At lower level, even hardware SAS passes hardware messages
  - Hw support for fine-grained comm makes software MP faster as well

**Data Parallel Systems**

**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 associated with each data element

**Architectural model**

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

**Original motivations**

- Matches simple differential equation solvers
- Centralize high cost of instruction fetch/sequencing
Application of Data Parallelism

- Each PE contains an employee record with his/her salary
  
  If salary > 100K then
  
  salary = salary * 1.05
  
  else
  
  salary = salary * 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,

Evolution and Convergence

Rigid control structure (SIMD in Flynn taxonomy)

- SISD = uniprocessor, MIMD = multiprocessor

Popular when cost savings of centralized sequencer high

- 60s when CPU was a cabinet
  - Replaced by vectors in mid-70s
    - More flexible w.r.t. memory layout and easier to manage
  - Revived in mid-80s when 32-bit datapath slices just fit on chip
  - No longer true with modern microprocessors

Other reasons for demise

- Simple, regular applications have good locality, can do well anyway
- Loss of applicability due to hardwiring data parallelism
  - MIMD machines as effective for data parallelism and more general

Prog. model converges with SPMD (single program multiple data)

- Contributes need for fast global synchronization
- Structured global address space, implemented with either SAS or MP

Convergence: Generic Parallel Architecture

A generic modern multiprocessor

- Node: processor(s), memory system, plus communication assist
  - Network interface and communication controller
- Scalable network
- Communication assist provides primitives with perf profile
  - Build your programming model on this
- Convergence allows lots of innovation, now within framework
  - Integration of assist with node, what operations, how efficiently...
The Model/System Contract

Model specifies an interface (contract) to the programmer

- **Naming**: How are logically shared data and/or processes referenced?
- **Operations**: What operations are provided on these data?
- **Ordering**: How are accesses to data ordered and coordinated?
- **Replication**: How are data replicated to reduce communication?

Underlying implementation addresses performance issues

- **Communication Cost**: Latency, bandwidth, overhead, occupancy

We’ll look at the aspects of the contract through examples

Supporting the Contract

<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>
<tr>
<td>Operating systems support</td>
<td>Communication abstraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication hardware</td>
<td>Hardware/software boundary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Given prog. model can be supported in various ways at various layers.
In fact, each layer takes a position on all issues (naming, ops, performance etc), and any set of positions can be mapped to another by software.

Key issues for supporting programming models are:

- What primitives are provided at comm. abstraction layer
- How efficiently are they supported (hw/sw)
- How are programming models mapped to them

Recap of Parallel Architecture

Parallel architecture is important thread in evolution of architecture

- At all levels
- Multiple processor level now in mainstream of computing

Exotic designs have contributed much, but given way to convergence

- Push of technology, cost and application performance
- Basic processor-memory architecture is the same
- Key architectural issue is in communication architecture
  - How communication is integrated into memory and I/O system on node

Fundamental design issues

- Functional: naming, operations, ordering
- Performance: organization, replication, performance characteristics

Design decisions driven by workload-driven evaluation

- Integral part of the engineering focus

Old Conventional Wisdems Facing “Walls”

| Power is free, transistors are expensive | Power wall |
| Multiply is slow memory access is fast | Memory wall |

Increasing Instruction Level Parallelism via compilers, innovation (Out-of-order, speculation, VLIW, …)

Old: Uniprocessor Performance 2x / 1.5 years
New: Uniprocessor Performance 2x / 5 years?
Technology Trends Once Again

Uniprocessor Performance (SPECint)


- VAX: 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Multi-Cores Are In

“We are dedicating all of our future product development to multicore designs. … This is a sea change in computing”

*Paul Otellini, Intel (2005)*

<table>
<thead>
<tr>
<th>Manufacturer/Year</th>
<th>AMD/’05</th>
<th>IBM/’04</th>
<th>Intel/’06</th>
<th>Sun/’05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors/chip</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Threads/Processor</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Threads/chip</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>32</td>
</tr>
</tbody>
</table>

Intel demonstrated 80-core research chip
CISCO’s metro-chip has 188 cores
**Parallel Programs**

**Why Bother with Programs?**

They’re what runs on the machines we design
- Helps make design decisions
- Helps evaluate systems tradeoffs
Led to the key advances in uniprocessor architecture
- Caches and instruction set design
More important in multiprocessors
- New degrees of freedom
- Greater penalties for mismatch between program and architecture

**Important for Whom?**

Algorithm designers
- Designing algorithms that will run well on real systems
Programmers
- Understanding key issues and obtaining best performance
Architects
- Understand workloads, interactions, important degrees of freedom
- Valuable for design and for evaluation

**Lectures about Programs in This Course**

- Parallel programs
  - Process of parallelization
  - What parallel programs look like in major programming models
- Programming for performance
  - Key performance issues and architectural interactions
- Workload-driven architectural evaluation
  - Beneficial for architects and for users in procuring machines
Unlike on sequential systems, can’t take workload for granted
  - Software base not mature; evolves with architectures for performance
  - So need to open the box
Let’s begin with parallel programs ...
Outline

Motivating Problems (application case studies)

Steps in creating a parallel program

What a simple parallel program looks like
  • In the three major programming models
  • What primitives must a system support?

Later: Performance issues and architectural interactions

Motivating Problems

Simulating Ocean Currents
  • Regular structure, scientific computing

Simulating the Evolution of Galaxies
  • Irregular structure, scientific computing

Rendering Scenes by Ray Tracing
  • Irregular structure, computer graphics

Filtering (Pipeline parallelism)
  • Irregular structure, information processing
  • Not discussed here (read in book)

Simulating Ocean Currents

(a) Cross sections
(b) Spatial discretization of a cross section

• Model as two-dimensional grids
• Discretize in space and time
  – finer spatial and temporal resolution => greater accuracy
• Many different computations per time step
  – set up and solve equations
• Concurrency across and within grid computations

Simulating Galaxy Evolution

• Simulate the interactions of many stars evolving over time
• Computing forces is expensive
• $O(n^2)$ brute force approach
• Hierarchical Methods take advantage of force law: $G \frac{m_1 m_2}{r^2}$

Star on which forces are being computed
Star too close to approximate
Small group far enough away to approximate to center of mass
Large group far enough away to approximate

• Many time-steps, plenty of concurrency across stars within one
**Rendering Scenes by Ray Tracing**

- Shoot rays into scene through pixels in image plane
- Follow their paths
  - they bounce around as they strike objects
  - they generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays

All case studies have abundant concurrency

---

**Creating a Parallel Program**

Assumption: Sequential algorithm is given

- Sometimes need very different algorithm, but beyond scope

Pieces of the job:

- Identify work that can be done in parallel
- Partition work and perhaps data among processes
- Manage data access, communication and synchronization

*Note:* work includes computation, data access and I/O

Main goal: Speedup (plus low prog. effort and resource needs)

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

For a fixed problem:

\[ \text{Speedup (p)} = \frac{\text{Time(1)}}{\text{Time(p)}} \]

---

**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
**Decomposition**

Break up computation into tasks to be divided among processes
- Tasks may become available dynamically
- No. of available tasks may vary with time

i.e. identify concurrency and decide level at which to exploit it

**Goal:** Enough tasks to keep processes busy, but not too many
- No. of tasks available at a time is upper bound on achievable speedup

**Limited Concurrency: Amdahl’s Law**

- Most fundamental limitation on parallel speedup
- If fraction $s$ of seq execution is inherently serial, speedup $\leq 1/s$
- Example: 2-phase calculation
  - sweep over $n$-by-$n$ grid and do some independent computation
  - sweep again and add each value to global sum
- Time for first phase $= n^2/p$
- Second phase serialized at global variable, so time $= n^2$
- Speedup $\leq \frac{2n^2}{n^2} = 2$
- Trick: divide second phase into two
  - accumulate into private sum during sweep
  - add per-process private sum into global sum
- Parallel time is $n^2/p + n^2/p + p$, and speedup at best $\frac{2n^2}{2n^2 + p^2}$

**Pictorial Depiction**

![Diagram showing decomposition and concurrency]

**Concurrency Profiles**

- Cannot usually divide into serial and parallel part
- Area under curve is total work done, or time with 1 processor
- Horizontal extent is lower bound on time (infinite processors)
- Speedup is the ratio: $\frac{\sum f_k}{\sum_{k=1}^{\infty} f_{\frac{n^2}{p}}} \cdot \frac{\frac{1}{s} + \frac{L}{p}}{\frac{1}{s} + \frac{L}{p}}$
- Amdahl’s law applies to any overhead, not just limited concurrency
Types of Parallelism

Task Parallelism  
- Parallelism explicit in algorithm  
- Between filters without producer/consumer relationship

Data Parallelism  
- Peel iterations of filter, place within scatter/gather pair (fission)  
- Parallelize filters with state

Pipeline Parallelism  
- Between producers and consumers  
- Stateful filters can be parallelized

Assignment

Specifying mechanism to divide work up among processes  
- E.g. which process computes forces on which stars, or which rays  
- Together with decomposition, also called partitioning  
- Balance workload, reduce communication and management cost

Structured approaches usually work well  
- Code inspection (parallel loops) or understanding of application  
- Well-known heuristics  
- Static versus dynamic assignment

As programmers, we worry about partitioning first  
- Usually independent of architecture or prog model  
- But cost and complexity of using primitives may affect decisions

As architects, we assume program does reasonable job of it
Orchestration

• Naming data
• Structuring communication
• Synchronization
• Organizing data structures and scheduling tasks temporally

Goals
• Reduce cost of communication and synch. as seen by processors
• Preserve locality of data reference (incl. data structure organization)
• Schedule tasks to satisfy dependences early
• Reduce overhead of parallelism management

Closest to architecture (and programming model & language)
• Choices depend a lot on comm. abstraction, efficiency of primitives
• Architects should provide appropriate primitives efficiently

Mapping

After orchestration, already have parallel program

Two aspects of mapping:
• Which processes will run on same processor, if necessary
• Which process runs on which particular processor
  – mapping to a network topology

One extreme: space-sharing
• Machine divided into subsets, only one app at a time in a subset
• Processes can be pinned to processors, or left to OS

Another extreme: complete resource management control to OS
• OS uses the performance techniques we will discuss later

Real world is between the two
• User specifies desires in some aspects, system may ignore

Usually adopt the view: process <-> processor

Parallelizing Computation vs. Data

Above view is centered around computation
• Computation is decomposed and assigned (partitioned)

Partitioning Data is often a natural view too
• Computation follows data: owner computes
• Grid example; data mining; High Performance Fortran (HPF)

But not general enough
• Distinction between comp. and data stronger in many applications
  – Barnes-Hut, Raytrace (later)
• Retain computation-centric view
• Data access and communication is part of orchestration

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>Orchestratio</td>
<td>Yes</td>
<td>Reduce communication volume</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce nonoverlapped communication via data locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication and synchronization cost as seen by the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 dependencies 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>

But low resource usage and development effort

Implications for algorithm designers and architects
• Algorithm designers: high-perf., low resource needs
• Architects: high-perf., low cost, reduced programming effort
  – e.g. gradually improving perf. with programming effort may be preferable to sudden threshold after large programming effort
What Parallel Programs Look Like

Parallelization of An Example Program

Motivating problems all lead to large, complex programs
Examine a simplified version of a piece of Ocean simulation
  • Iterative equation solver
Illustrate parallel program in low-level parallel language
  • C-like pseudocode with simple extensions for parallelism
  • Expose basic comm. and synch. primitives that must be supported
  • State of most real parallel programming today

Grid Solver Example

• Simplified version of solver in Ocean simulation
• Gauss-Seidel (near-neighbor) sweeps to convergence
  – interior n-by- n points of (n+2)-by-(n+2) updated in each sweep
  – updates done in-place in grid, and diff. from prev. value computed
  – accumulate partial diffs into global diff at end of every sweep
  – check if error has converged (to within a tolerance parameter)
  – if so, exit solver; if not, do another sweep

Expression for updating each interior point:

1. int n; /* size of matrix: (n + 2)-by-(n + 2) elements*/
2. float **A, diff = 0;
3. main()
4. begin
5.  read(n) ; /*read input parameter: matrix size*/
6.  A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.  initialize(A); /*initialize the matrix A somehow*/
8.  Solve (A); /*call the routine to solve equation*/
9.  end main
10. procedure Solve (A)
11.  /* solve the equation system*/
12.  float **A;
13.  /* A is an (n+2)-by-(n+2) array*/
14.  begin
15.  int i, j, done = 0;
16.  float diff = 0, temp;
17.  while (!done) do
18.    /*outermost loop over sweeps*/
19.    diff = 0;
20.    /*initialize maximum difference to 0*/
21.    for i ← 1 to n do /*sweep over nonborder points of grid*/
22.      for j ← 1 to n do /*sweep over border points of grid*/
23.        temp = A[i,j]; /*save old value of element*/
25.        diff += abs(A[i,j] - temp);
26.      end for
27.    end for
28.    if (diff/(n*n) < TOL) then done = 1;
29.  end while
30.  end procedure
Decomposition

• Simple way to identify concurrency is to look at loop iterations
  – dependence analysis; if not enough concurrency,
• Not much concurrency here at this level (all loops sequential)
• Examine fundamental dependences, ignoring loop structure

- Concurrency $O(n)$ along anti-diagonals, serialization $O(n)$ along diag.
- Retain loop structure, use point-to-point synchronizations; problems?
- Restructure loops, use global synchronizations; problems?

Exploit Application Knowledge

• Reorder grid traversal: red-black ordering
  ![Red-black grid traversal diagram]

  - Different ordering of updates: may converge quicker or slower
  - Red sweep and black sweep are each fully parallel:
  - Global synchronization among 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 nondeterministic

Decomposition Only

15. while (!done) do /*a sequential loop*/
16.   diff = 0;
17.   for_all i ← 1 to n do /*a parallel loop nest*/
18.     for_all j ← 1 to n do
19.       temp = A[i,j];
22.       diff += abs(A[i,j] - temp);
23.   end for_all
24. end for_all
25. if (diff/(n*n) < TOL) then done = 1;
26. end while

• Decomposition into elements: degree of concurrency $n^2$
• To decompose into rows, make line 18 loop sequential; degree $n$
• for_all leaves assignment left to system
  – but implicit global synch. at end of for_all loop

Assignment

• Static assignments (given decomposition into rows)
  – block assignment of rows: Row $i$ is assigned to process $\frac{i}{p}$
  – cyclic assignment of rows: process $i$ is assigned rows $i, i+p, and so on$

• Dynamic assignment
  – get a row index, work on the row, get a new row, and so on
• Static assignment into rows reduces concurrency (from $n$ to $p$)
  – block assign. reduces communication by keeping adjacent rows together
• Let’s dig into orchestration under three programming models
**Data Parallel Solver**

1. int n, nprocs; /*grid size (n + 2-by-n + 2) and number of processes*/
2. float **A, diff = 0;
3. main()
4. begin
5. read(n); read(nprocs); /*read input grid size and number of processes*/
6. A = G_MALLOC(a 2-d array of size n+2 by n+2 doubles);
7. Initialize(A); /*initialize the matrix A somehow*/
8. Solve(A); /*call the routine to solve equation*/
9. end main
10. procedure Solve(A)
/*solve the equation system*/
11. float **A;
/*A is an (n + 2-by-n + 2) array*/
12. begin
13. int i, j, done = 0;
14. float mydiff = 0, temp;
14a. DECOMP A[BLOCK,*, nprocs];
15. while (!done) do /*outermost loop over sweeps*/
16. mydiff = 0; /*initialize maximum difference to 0*/
17. for_all i ← 1 to n do /*sweep over non-border points of grid*/
18. for_all j ← 1 to n do
19. temp = A[i,j]; /*save old value of element*/
22. mydiff += abs(A[i,j] - temp);
23. end_for_all
24. end_for_all
24a. REDUCE (mydiff, diff, ADD);
25. if (diff/(n*n) < TO L) then done = 1;
26. end while
27. end procedure

**Shared Address Space Solver**

Single Program Multiple Data (SPMD)

Processes

- Solve
- Solve
- Solve
- Solve

Sweep

Test Concurrency

- Assignment controlled by values of variables used as loop bounds

**Notes on SAS Program**

- SPMD: not lockstep or even necessarily same instructions
- Assignment controlled by values of variables used as loop bounds
  - unique pid per process, used to control assignment
- Done condition evaluated redundantly by all
- Code that does the update identical to sequential program
  - each process has private mydiff variable
- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why the need for all the barriers?
**Need for Mutual Exclusion**

- Code each process executes:
  
  ```
  load the value of diff into register r1
  add the register r2 to register r1
  store the value of register r1 into diff
  ```

- A possible interleaving:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 ← diff</td>
<td>{P1 gets 0 in its r1}</td>
</tr>
<tr>
<td>r1 ← diff</td>
<td>{P2 also gets 0}</td>
</tr>
<tr>
<td>r1 ← r1+r2</td>
<td>{P1 sets its r1 to 1}</td>
</tr>
<tr>
<td>r1 ← r1+r2</td>
<td>{P2 sets its r1 to 1}</td>
</tr>
<tr>
<td>diff ← r1</td>
<td>{P1 sets cell_cost to 1}</td>
</tr>
<tr>
<td>diff ← r1</td>
<td>{P2 also sets cell_cost to 1}</td>
</tr>
</tbody>
</table>

- Need the sets of operations to be atomic (mutually exclusive)

**Global Event Synchronization**

BARRIER(nprocs): wait here till nprocs processes get here

- Built using lower level primitives
- Global sum example: wait for all to accumulate before using sum
- Often used to separate phases of computation

<table>
<thead>
<tr>
<th>Process P_1</th>
<th>Process P_2</th>
<th>Process P_nprocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>set up eqn system</td>
<td>set up eqn system</td>
<td>set up eqn system</td>
</tr>
<tr>
<td><strong>Barrier</strong> (name, nprocs)</td>
<td><strong>Barrier</strong> (name, nprocs)</td>
<td><strong>Barrier</strong> (name, nprocs)</td>
</tr>
<tr>
<td>solve eqn system</td>
<td>solve eqn system</td>
<td>solve eqn system</td>
</tr>
<tr>
<td><strong>Barrier</strong> (name, nprocs)</td>
<td><strong>Barrier</strong> (name, nprocs)</td>
<td><strong>Barrier</strong> (name, nprocs)</td>
</tr>
<tr>
<td>apply results</td>
<td>apply results</td>
<td>apply results</td>
</tr>
<tr>
<td><strong>Barrier</strong> (name, nprocs)</td>
<td><strong>Barrier</strong> (name, nprocs)</td>
<td><strong>Barrier</strong> (name, nprocs)</td>
</tr>
</tbody>
</table>

- Conservative form of preserving dependences, but easy to use

WAIT_FOR_END (nprocs-1)

**Group Event Synchronization**

Subset of processes involved

- Can use flags or barriers (involving only the subset)
- Concept of producers and consumers

Major types

- Single-producer, multiple-consumer
- Multiple-producer, single-consumer
- Multiple-producer, single-consumer

**Mutual Exclusion**

Provided by LOCK-UNLOCK around *critical section*

- Set of operations we want to execute atomically
- Implementation of LOCK/UNLOCK must guarantee mutual excl.

Use Pthreads:

```c
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock( &mutex1 );
diff += mydiff;
pthread_mutex_unlock( &mutex1 );
```

Issues with Mutex on a large machine?
**Point-to-point Event Synchronization**

One process notifies another of an event so it can proceed

- Common example: producer-consumer (bounded buffer)
- Concurrent programming on uniprocessor: semaphores
- Shared address space parallel programs: semaphores, or use ordinary variables as flags

**P1**

\[
\begin{align*}
\text{A: } & \text{while (flag == 0) } \downarrow \\
& \text{do something else} \\
\text{B: } & \text{flag = 1;}
\end{align*}
\]

**Synchronization and Queues**

- Pipeline parallelism
  - Queues between stages of the producer-consumer pipeline
- Multiple threads may cooperate on a stage of the pipeline e.g. in data parallel or task-parallel manner
- Only the cooperating threads in a stage need to synchronize (lock, barrier, etc.) together within that stage
- But accesses to the same queue need to be synchronized

**Programming with Pthreads: Creation and Join**

```c
#include <pthread.h>
void *parallel_work( void *ptr );
main(){
    pthread_t threads[N];
    int i;
    ... 
    for (i=1; i < N; i++)
        pthread_create( &threads[i], NULL,
                        parallel_work, (void*) &input[i] );
    parallel_work( &input[0] ); /* optional */
    for (i=1; i < N; i++)
        pthread_join( threads[i], NULL );
} 
void *parallel_work( void *ptr )
{
    ... /* work here */
}
```

**More on Programming with Pthreads**

Number of threads ≠ Number of processors?

- Under what conditions you want them to be the same?
- Under what conditions you want them to be different?

Should “Master” do parallel work or not?

- Do parallel work: creating N-1 threads
- Master is special (like in the textbook)
  - Master coordinates and sleeps
  - Create N threads
More on Programming with Pthreads

Passing data
- Pass input/output via I/O pointers
- Pass via shared data

Thread characteristics
- Controlled through an attributes struct, similar to an C++ object
- Set your preferences before creating threads such as uniprocessor vs. multiprocessors

Thread termination
- Threads terminate when its function returns
- You can terminate early by calling pthread_exit()
- You can kill other threads (e.g. Master) by calling pthread_cancel().

Mesa-Style Monitor

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
...

pthread_mutex_lock( &mutex );
queue_insert( q, &item );
pthread_signal( &cond );
pthread_mutex_unlock( &mutex );

pthread_mutex_lock( &mutex );
while ( q == NULL )
pthread_cond_wait( &cond, &mutex );
queue_remove( q );
pthread_mutex_unlock( &mutex );
```

Is this consumer-producer?
Any issues with the code?

Consumer-Producer with Mesa-Style Monitor

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;

Insert(Buf_t buffer, item_t item)
{
    pthread_mutex_lock(&mutex);
    if (count==N)
        pthread_cond_wait(&full, &mutex);
    insert item into buffer
    count++;
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
}

Item_t Remove(Buf_t buffer)
{
    pthread_mutex_lock(&mutex);
    while (!count)
        pthread_cond_wait(&empty, &mutex);
    remove item from buffer
    count--;
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
}
```

Any issues with this?

Consumer-Producer with Mesa-Style Monitor

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;

Insert(Buf_t buffer, item_t item)
{
    pthread_mutex_lock(&mutex);
    while (count==N)
        pthread_cond_wait(&full, &mutex);
    insert item into buffer
    count++;
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
}

Item_t Remove(Buf_t buffer)
{
    pthread_mutex_lock(&mutex);
    while (!count)
        pthread_cond_wait(&empty, &mutex);
    remove item from buffer
    count--;
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
}
```

This is the idiom of Mesa-style monitor
Message Passing Grid Solver

- Cannot declare A to be shared array any more
- Need to compose it logically from per-process private arrays
  - usually allocated in accordance with the assignment of work
  - process assigned a set of rows allocates them locally
- Transfers of entire rows between traversals
- Structurally similar to SAS (e.g. SPMD), but orchestration different
  - data structures and data access/naming
  - communication
  - synchronization

```
1. int pid, n, b;
/*process id, matrix dimension and number of processors to be used*/
2. float **myA;
3. main()
   4.begin
   5.read(n); read(nprocs); /*read input matrix size and number of processes*/
6. a. CREATE (nprocs-1, Solve);
7. b. Solve(); /*main process becomes a worker too*/
8. c. WAIT_FOR_END (nprocs–1); /*wait for all child processes created to terminate*/
9.end main
10. procedure Solve()
11. begin
12. int i,j, pid,
n’ = n/nprocs,
done = 0;
13. float temp, tempdiff,
mydiff = 0;
/*private variables*/
14. myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
/*my assigned rows of A*/
15. initialize(myA); /*initialize my rows of A, in an unspecified way*/
16. while (!done) do
17. mydiff = 0;
/*set local diff to 0*/
18a. if (pid != 0) then
SEND (&myA[1,0],n*sizeof(float),pid-1, ROW);
18b. if (pid = nprocs-1) then
SEND (&myA[n’,0],n*sizeof(float),pid+1, ROW);
18c. if (pid != 0) then
RECEIVE (&myA[0,0],n*sizeof(float),pid-1, ROW);
18d. if (pid != nprocs-1) then
RECEIVE (&myA[n’+1,0],n*sizeof(float), pid+1, ROW);
/*border rows of neighbors now have been copied into myA[0,*] and myA[n’+1,*]*/
19. for i ← 1 to n’ do
/*for each of my (nonghost) rows*/
20. for j ← 1 to n do
/*for all nonborder elements in that row*/
21. temp = myA[i,j];
23. endfor
24. endfor
/*communicate local diff values and determine if done; can be replaced by reduction and broadcast*/
25a. if (pid != 0) then
/*process 0 holds global total diff*/
25b. SEND (mydiff,sizeof(float),0, DIFF);
25c. RECEIVE (done,sizeof(int),0, DONE);
25d. else
/*pid 0 does this*/
25e. for i ← 1 to nprocs-1 do
/*for each other process*/
25f. RECEIVE (tempdiff,sizeof(float),*, DIFF);
25g. mydiff += tempdiff; /*accumulate into total*/
25h. endif
25i if (mydiff/(n*n) < TOL) then done = 1;
25j. for i ← 1 to nprocs-1 do
/*for each other process*/
25k. SEND (done,sizeof(int),i, DONE);
25l. endfor
25m. endif
26. endwhile
27. end procedure
```

Notes on Message Passing Program

- Use of ghost rows
- Receive does not transfer data, send does
  - unlike SAS which is usually receiver-initiated (load fetches data)
- Communication done at beginning of iteration, so no asynchrony
- Communication in whole rows, not element at a time
- Core similar, but indices/bounds in local rather than global space
- Synchronization through sends and receives
  - Update of global diff and event synch for done condition
  - Could implement locks and barriers with messages
- Can use REDUCE and BROADCAST library calls to simplify code

```
/*communicate local diff values and determine if done, using reduction and broadcast*/
25a. REDUCE(0,mydiff,sizeof(float),ADD);
25b. if (pid == 0) then
/*process id, matrix dimension and number of processors to be used*/
25c. RECEIVE(done,sizeof(int),0,DONE);
25d. else
/*pid 0 does this*/
25e. for i ← 1 to nprocs-1 do
/*for each other process*/
25f. RECEIVE(tempdiff,sizeof(float),*, DIFF);
25g. ENDIF
```

Send and Receive Alternatives

Can extend functionality: stride, scatter-gather, groups
Semantic flavors: based on when control is returned
Affect when data structures or buffers can be reused at either end
Send/Receive

```
Synchronous
Asynchronous
Blocking asynch.  Nonblocking asynch.
```

- Affect event synch (mutual excl. by fiat: only one process touches data)
- Affect ease of programming and performance
Synchronous messages provide built-in synch. through match
  - Separate event synchronization needed with asynch. messages
With synch. messages, our code is deadlocked.  Fix?
**Orchestration: Summary**

Shared address space
- Shared and private data explicitly separate
- Communication implicit in access patterns
- No correctness need for data distribution
- Synchronization via atomic operations on shared data
- Synchronization explicit and distinct from data communication

Message passing
- Data distribution among local address spaces needed
- No explicit shared structures (implicit in comm. patterns)
- Communication is explicit
- Synchronization implicit in communication (at least in synch. case)
  - mutual exclusion by fiat

**Correctness in Grid Solver Program**

Decomposition and Assignment similar in SAS and message-passing
Orchestration is different
- Data structures, data access/naming, communication, synchronization

<table>
<thead>
<tr>
<th></th>
<th>SAS</th>
<th>Msg-Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit global data structure?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment indep of data layout?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Communication</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Explicit replication of border rows?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Requirements for performance are another story ...