Parallel Programming

To understand and evaluate design decisions in a parallel machine, we must get an idea of the software that runs on a parallel machine.

--Introduction to Culler et al.’s Chapter 2, beginning 192 pages on software
Outline

• Review
• Applications
• Creating Parallel Programs
• Programming for Performance
• Scaling

Review: Separation of Model and Architecture

• Shared Memory
  – Single shared address space
  – Communicate, synchronize using load / store
  – Can support message passing
• Message Passing
  – Send / Receive
  – Communication + synchronization
  – Can support shared memory
• Data Parallel
  – Lock-step execution on regular data structures
  – Often requires global operations (sum, max, min...)
  – Can support on either SM or MP
Review: A Generic Parallel Machine

- Separation of programming models from architectures
- All models require communication
- Node with processor(s), memory, communication assist

Review: Fundamental Architectural Issues

- **Naming**: How is communicated data and/or partner node referenced?
- **Operations**: What operations are allowed on named data?
- **Ordering**: How can producers and consumers of data coordinate their activities?
- **Performance**
  - **Latency**: How long does it take to communicate in a protected fashion?
  - **Bandwidth**: How much data can be communicated per second? How many operations per second?
Applications

- N-Body Simulation: Barnes-Hut
- Ocean Current Simulation: Ocean
- VLSI Routing: Locus Route
- Ray Tracing
  - Shoot Ray through three dimensional scene (let it bounce off objects)
- Data Mining
  - finding associations
  - Consumers that are college students, and buy beer, tend to buy chips

Barnes-Hut

- Computing the mutual interactions of N bodies
  - n-body problems
  - stars, planets, molecules…
- Can approximate influence of distant bodies
Ocean

- Simulate ocean currents
- discretize in space and time

Creating a Parallel Program

- Can be done by programmer, compiler, run-time system or OS
- A **Task** is a piece of work
  - Ocean: grid point, row, plane
  - Raytrace: 1 ray or group of rays
- **Task grain**
  - small => fine-grain task
  - large => course-grain task
- **Process (thread) performs tasks**
  - According to OS: process = thread(s) + address space
- **Process (threads) executed on processor(s)**
Example: Ocean

- Equation Solver
  - kernel = small piece of important code (Not OS kernel...)
- Update each point based on NEWS neighbors
  - Gauss-Seidel (update in place)
- Compute average difference per element
- Convergence when diff small => exit

Equation Solver Decomposition

while !converged
  for
    for
      • The loops are not independent!
      • Exploit properties of problem
        - Don’t really need up-to-date values (approximation)
        - May take more steps to converge, but exposes parallelism
**Sequential Solver**

- **Recurrence in Inner Loop**

```plaintext
10. procedure Solve (A) /* solve the equation system */
11. float **A; /* A is an (n + 2) x (n + 2) array */
12. begin
13. int i, j, done = 0;
14. float diff = 0, temp;
15. while (!done) /* outermost loop over sweeps */
16.   diff = 0; /* initialize maximum difference to 0 */
17.   for i = 1 to n do /* sweep over non-border points of grid */
18.     for j ← 1 to n do
19.         temp = A[i, j]; /* save old value of element */
22.         diff = abs(A[i, j] - temp);
23.     end for
24.   end for
25.   if (diff/(n*n) < TOL) then done = 1;
26. end while
27. end procedure
```

**Parallel Solver**

- **Identical computation as Original Code**
- **Parallelism along anti-diagonal**
- **Different degrees of parallelism as diagonal grows/shrinks**
The FORALL Statement

while !converged
  forall
    forall
      • Can execute the iterations in parallel
      • Each grid point computation (n^2 parallelism)
  while !converged
    forall
      for
        • Computation for rows is independent (n parallelism)
          – less overhead

Asynchronous Parallel Solver

Each processor updates its region independent of other’s values
• Global synch at end of iteration, to keep things somewhat up-to-date
• non-deterministic

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
Red-Black Parallel Solver

- Red-Black
  - like checkerboard update of Red point depends only on Black points
  - alternate iterations over red, then black
  - but convergence may change
  - deterministic

PARMACS

- Macro Package, runtime system must implement
  - portability

CREATE(p,proc,args) Create p processes executing proc(args)
G_MALLOC(size) Allocate shared data of size bytes
LOCK(name)
UNLOCK(name)
BARRIER(name,number) Wait for number processes to arrive
WAIT_FOR_END(number) Wait for number processes to terminate
WAIT(flag) while(!flag);
SIGNAL(flag) flag = 1;
Shared Memory Programming

19. procedure solve(a)
   20.   begin
19.   end
13.   int i, j, done = 0;
14.   float tmp, mydiff = 0;
15.   int mymax = 1 * (gpid + n/proc - 1) /* assume that n is exactly divisible by*/
16.   while (!done) do /* once loop over all diagonal elements */
17.     mydiff = diff[i] = 0; /* get global diff to check if all are close enough */
18.     while (i < n) do /* for each of my rows */
19.       for j = 0 to n do /* for all non-zero elements in this row */
20.         tmp = a[i,j];
21.         a[i,j] = phi[i,j] * a[i,j] + a[i,j-1];
22.         mydiff = abs(a[i,j] - temp);
23.     done[i] = true;
24.   endwhile
25.   endprocedure

Message Passing Primitives

Table 2.3 Some Basic Message-Passing Primitives

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>CREATE{procedure}</td>
<td>Create process that starts at procedure</td>
</tr>
<tr>
<td>SEND</td>
<td>SEND(src_addr, size, dest, tag)</td>
<td>Send size bytes starting at src_addr to the dest process, with tag identifier</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>RECEIVE(buffer_addr, size, src, tag)</td>
<td>Receive a message with the tag identifier from the src process, and put size bytes of it into buffer starting at buffer_addr</td>
</tr>
<tr>
<td>SEND_PROBE</td>
<td>SEND_PROBE(tag, dest)</td>
<td>Check if message with identifier tag has been sent to process dest (only for asynchronous message passing, and meaning depends on semantics, as discussed in this section)</td>
</tr>
<tr>
<td>RECV_PROBE</td>
<td>RECV_PROBE(tag, src)</td>
<td>Check if message with identifier tag has been received from process src (only for asynchronous message passing, and meaning depends on semantics)</td>
</tr>
<tr>
<td>BARRIER</td>
<td>BARRIER(name, number)</td>
<td>Global synchronization among number processes; none gets past BARRIER until number have arrived</td>
</tr>
<tr>
<td>WAIT_FOR_END</td>
<td>WAIT_FOR_END(number)</td>
<td>Wait for number processes to terminate</td>
</tr>
</tbody>
</table>

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Smith & Singh
CS/ECE 757
19

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Smith & Singh
CS/ECE 757
20
Sends and Receives

- **Synchronous**
  - send: returns control to sending process after receive is performed
  - receive: returns control when data is written into address space
  - can deadlock on pairwise data exchanges

- **Asynchronous**
  - Blocking
    - send: returns control when the message has been sent from source data structure (but may still be in transit)
    - receive: like synchronous, but no ack is sent to sender
  - Non-Blocking
    - send: returns control immediately
    - receive: posts “intent” to receive
    - probe operations determine completion

Message Passing Programming

- **Create separate processes**
- **Each process a portion of the array**
  - $n/nprocs (+2)$ rows
  - boundary rows are passed in messages
    - deterministic because boundaries only change between iterations

- **To test for convergence, each process computes mydiff and sends to proc 0**
  - synchronized via send/receive
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. mydiff = 0; /*set local diff to 0*/
17. if (pid != 0) then SEND(myA[1,0], n*sizeof(float), pid-1, ROW); 
18. if (pid = nprocs-1) then
19. SEND(myA[n',0], n*sizeof(float), pid+1, ROW); 
20. if (pid != 0) then RECEIVE(myA[0,0], n*sizeof(float), pid-1, ROW); 
21. if (pid != nprocs-1) then
22. RECEIVE(myA[n'+1,0], n*sizeof(float), pid+1, ROW); /*border rows of neighbors have now been copied into myA[0,*] and myA[n'-1,*]*/
23. for i ← 1 to n' do /*for each of my (nonghost) rows*/
24. for j ← 1 to n do /*for all nonborder elements in that row*/
25. temp = myA[i,j]; 
27. mydiff ← abs(myA[i,j]) - temp; 
28. endfor
29. endfor /*communicate local diff values and determine if done; can be replaced by reduction and broadcast*/
30. if (pid != 0) then 
31. SEND(mydiff, sizeof(float), 0, DIFF); 
32. RECEIVE(done, sizeof(int), 0, DONE); 
33. else /*pid 0 does this*/
34. for i ← 1 to nprocs-1 do /*for each other process*/
35. RECEIVE(tempdiff, sizeof(float), -DIFF); 
36. mydiff ← tempdiff; /*accumulate into total*/
37. endif
38. endif 
39. endif
40. endif
41. endif
42. endfor /*for all rows*/
43. if (mydiff <= 0.0) then done = 1; 
44. for i ← 1 to nprocs-1 do /*for each other process*/
45. SEND(done, sizeof(int), 1, DONE); 
46. endif
47. endif
48. endif
49. endif
50. endwhile
51. end procedure
Programming for Performance

• Partitioning, Granularity, Communication, etc.
• Caches and Their Effects

Where Do Programs Spend Time?

• Sequential
  – Busy computing
  – Memory system stalls
• Parallel
  – Busy computing
  – Stalled for local memory
  – Stalled for remote memory (communication)
  – Synchronizing (load imbalance and operations)
  – Overhead
• Speedup \( p \) = \( \frac{\text{time}(1)}{\text{time}(p)} \)
  – Amdahl’s Law
  – Superlinear
**Speedup**

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\max(\text{Work on any processor})}
\]

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\max(\text{Work + Synch Wait + Communication})}
\]

**Concurrency Profile**

- Plot number of concurrent tasks over time
- Example: operate on \(n^2\) parallel data points, then sum them
  - sum sequentially
  - first sum blocks in parallel then sum \(p\) partial sums sequentially

---

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Smith & Singh

CS/ECE 757

(C) 2003 J. E. Smith

CS/ECE 757

Page 14
Concurreny Profile: Speedup

- \( \text{Speedup} \leq \frac{\text{Area under concurrency profile}}{\text{Horizontal extent of concurrency profile}} \)

- \( f_k = \text{fraction of work with concurrency } k (\leq p) \)
- \( p = \text{number of processors} \)

- \[
\text{Speedup}(p) \leq \sum_{k=1}^{p} f_k \left( \frac{1}{k} \right) = \frac{1}{\sum_{k=1}^{p} \frac{1}{k}}
\]

- Normalize total work to 1; make concurrency either serial or completely parallel \( \Rightarrow \text{Amdahl's Law} \)

\[
\frac{1}{s + \frac{1-s}{p}}
\]

Note: book has an unusual formulation of Amdahl's law (based on fraction of time rather than fraction of work)

Partitioning for Performance

- **Balance workload**
  - reduce time spent at synchronization

- **Reduce communication**

- **Reduce extra work**
  - determining and managing good assignment

- **These are at odds with each other**
Data vs. Functional Parallelism

- **Data Parallelism**
  - same ops on different data items
- **Functional (control, task) Parallelism**
  - pipeline
- **Impact on load balancing?**
- **Functional is more difficult**
  - longer running tasks

Impact of Task Granularity

- **Granularity = Amount of work associated with task**
- **Large tasks**
  - worse load balancing
  - lower overhead
  - less contention
  - less communication
- **Small tasks**
  - too much synchronization
  - too much management overhead
  - might have too much communication (affinity scheduling)
Impact of Concurrency

- Managing Concurrency
  - load balancing

- Static
  - Can not adapt to changes

- Dynamic
  - Can adapt
  - Cost of management increases
  - Self-scheduling (guided self-scheduling)
  - Centralized task queue
    - contention
  - Distributed task queue
    - Can steal from other queues
    - Arch: Name data associated with stolen task

Task Queues

(d) Centralized task queue
    All processes insert tasks
    Q
    All remove tasks

(b) Distributed task queues (one per process)
    P0 inserts P1 inserts P2 inserts P3 inserts
    Q0 Q1 Q2 Q3
    Others may steal
    P0 removes P1 removes P2 removes P3 removes

Copyright 1999 Morgan Kaufmann Publishers, Inc.
**Impact of Synchronization and Serialization**

- **Too coarse synchronization**
  - barriers instead of point-to-point synch
  - poor load balancing
- **Too many synchronization operations**
  - lock each element of array
  - costly operations
- **Coarse grain locking**
  - lock entire array
  - serialize access to array
- **Architectural aspects**
  - cost of synchronization operation
  - synchronization name space
Where Do Programs Spend Time?

- **Sequential**
  - Performing real computation
  - Memory system stalls

- **Parallel**
  - Performing real computation
  - Stalled for local memory
  - Stalled for remote memory (communication)
  - Synchronizing (load imbalance and operations)
  - Overhead

- **Speedup** \( (p) = \frac{\text{time}(1)}{\text{time}(p)} \)
  - Amdahl's Law (low concurrency limits speedup)
  - Superlinear speedup possible (how?)

Cache Memory 101

- **Locality + smaller HW is faster = memory hierarchy**
  - **Levels:** each smaller, faster, more expensive/byte than level below
  - **Inclusive:** data found in top also found in the bottom

- **Definitions**
  - **Upper** is closer to processor
  - **Block:** minimum unit of data present or not in upper level
  - **Frame:** HW (physical) place to put block (same size as block)
  - **Address** = \( \text{Block address} + \text{block offset address} \)
  - **Hit time:** time to access upper level, including hit determination

- **3C Model**
  - compulsory, capacity, conflict

- **Add another C: communication misses**
Multiprocessor as an Extended Mem. Hier.

- Example:
  - computation: 2 instructions per cycle (IPC)
    » or 500 cycles per 1000 instructions
  - 1 long miss per 1000 instructions
    » 200 cycles per long miss
  - => 700 cycles per 1000 instructions (40% slowdown)

Cache Coherent Shared Memory
Cache Coherent Shared Memory

P1

P2

ld r2, x

Interconnection Network

Main Memory

ld r2, x

Time

ld r2, x

Interconnection Network

Main Memory

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Smith & Singh

CS/ECE 757

41

42
Cache Coherent Shared Memory

P1 P2

ld r2, x
add r1, r2, r4
st x, r1

Interconnection Network

x
Main Memory

Reorder Computation for Caches

• Exploit Temporal and Spatial Locality
  – Temporal locality affects replication
  – Touch too much data == capacity misses

• Computation Blocking

Naïve Computation Order

Blocked Computation order
Reorder Data for Caches: Before

Elements on Same Page

Elements on Same Cache Block

Reorder Data for Caches: With Blocking

Elements on Same Page

Elements on Same Cache Block
Scaling: Why is it important?

- **Over time:**
  - computer systems become larger and more powerful
    » more powerful processors
    » more processors
    » also range of system sizes within a product family
  - problem sizes become larger
    » simulate the entire plane rather than the wing
  - required accuracy becomes greater
    » forecast the weather a week in advance rather than 3 days

- **Scaling:**
  - How do algorithms and hardware behave as systems, size, accuracies become greater?

- **Intuitively:**
  - “Performance” should scale linearly with cost

---

Cost

- **Cost is a function of more than just the processor.**
- **Cost is a complex function of many hardware components and software**
- **Cost is often not a "smooth" function**
  - Often a function of packaging
    » how many pins on a board
    » how many processors on a board
    » how many boards in a chassis
Aside on Cost-Effective Computing

• Isn’t Speedup(P) < P inefficient?
• If only throughput matters, use P computers instead?

• But much of a computer’s cost is NOT in the processor [Wood & Hill, IEEE Computer 2/95]
• Let Costup(P) = Cost(P)/Cost(1)
• Parallel computing cost-effective:
  • Speedup(P) > Costup(P)
• E.g. for SGI PowerChallenge w/ 500MB:
  • Costup(32) = 8.6

Questions in Scaling

• Fundamental Question:
  What do real users actually do when they get access to larger parallel machines?

• Constant Problem Size
  – Just add more processors

• Memory Constrained
  – Scale data size linearly with # of processors
  – Can significantly increase execution time

• Time Constrained
  – Keep same wall clock time as processors are added
  – Solve largest problem in same amount of time
**Problem Constrained Scaling**

- User wants to solve same problem, only faster
  - E.g., Video compression & VLSI routing

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

- Assessment
  - Good: easy to do & explain
  - May not be realistic
  - Doesn’t work well for much larger machine (c.f., Amdahl’s Law)

**Time Constrained Scaling**

- Execution time is kept fixed as system scales
  - User has fixed time to use machine or wait for result

- Performance = Work/Time as usual, and time is fixed, so

\[
\text{Speedup}_{TC}(p) = \frac{\text{Work}(p)}{\text{Work}(1)}
\]

- Assessment
  - Often realistic (e.g., best weather forecast over night)
  - Must understand application to scale meaningfully (would scientist scale grid, time step, error bound, or combination?)
  - Execution time on a single processor can be hard to get (no uniprocessor may have enough memory)
Memory Constrained Scaling

- Scale so memory usage per processor stays fixed
- Scaled Speedup: Is Time(1) / Time(p)?

\[ \text{Speedup}_{MC}(p) = \frac{\text{Work}(p)}{\text{Time}(p)} \times \frac{\text{Time}(1)}{\text{Work}(1)} = \frac{\text{Increase in Work}}{\text{Increase in Time}} \]

- Assessment
  - Realistic for memory-constrained programs (e.g., grid size)
  - Can lead to large increases in execution time if work grows faster than linearly in memory usage
  - e.g. matrix factorization
    » 10,000-by 10,000 matrix takes 800MB and 1 hour on uniprocessor
    » With 1,000 processors, can run 320K-by-320K matrix
    » but ideal parallel time grows to 32 hours!

Scaling Down

- Scale down to shorten evaluation time on hardware and especially on simulators
- “Scale up” issues apply in reserve
- Must watch out if problem size gets too small
  - Communication dominates computation (e.g., all boundary elements)
  - Problem size gets too small for realistic caches, yielding too many cache hits
    » Scale caches down considering application working sets
    » E.g., if a on a realistic problem a realistic cache could hold a matrix row but not whole matrix
    » Scale cache so it hold only row or scaled problem’s matrix
The SPLASH2 Benchmarks

- **Kernels**
  - Complex 1D FFT
  - Blocked LU Factorization
  - Blocked Sparse Cholesky Factorization
  - Integer Radix Sort

- **Applications**
  - Barnes-Hut: interaction of bodies
  - Adaptive Fast Multipoles (FMM): interaction of bodies
  - Ocean Simulation
  - Hierarchical Radiosity
  - Ray Tracer (Raytrace)
  - Volume Renderer (Volrend)
  - Water Simulation with Spatial Data Structure (Water-Spatial)
  - Water Simulation without Spatial Data Structure (Water-Nsquared)