Parallel Programming (Chapters 2, 3, & 4)

Copyright 2001 Mark D. Hill
University of Wisconsin-Madison

Slides are derived from work by
Sarita Adve (Illinois), Babak Falsafi (CMU),
Alvy Lebeck (Duke), Steve Reinhardt (Michigan),
and J. P. Singh (Princeton). Thanks!

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 combined
  – Can support shared memory (with some effort)
• Data Parallel
  – Lock-step execution on regular data structures
  – Often requires global operations (sum, max, min...)
  – Can support on either SM or MP (SPMD)
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

- **Ocean Current Simulation**
  - Typical of 3-D simulations
  - Has a major effect on world climate
- **N-Body**
  - Evolution of galaxies
- **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

Ocean

- **Simulate ocean currents**
- **discretize in space and time**
Barnes-Hut

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

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)
Steps for Creating a Parallel Program

- Decomposition into tasks
- Assignment of tasks to processes
- Orchestration of data access, communication, etc
- Mapping processes to processors

Decomposition

- Decompose computation into set of tasks
- Task availability can be dynamic
- Maximize concurrency
- Minimize overhead of managing tasks
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 partial sums sequentially

Concurrency Profile: Speedup

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

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

$$\text{Speedup}(p) \leq \frac{\sum_{k=1}^{p} f_k}{\sum_{k=1}^{p} f_k (\frac{1}{k})} = \frac{1}{\sum_{k=1}^{p} \frac{f_k}{k}}$$

- Normalize total work to 1: make concurrency either serial or completely parallel.
  - 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)
### Assignment

- Assign tasks to processes (static or dynamic)
- Balance workload
- Reduce communication (while keeping load balanced)
- Minimize overhead
- Assignment + Decomposition = Partitioning

![Diagram](AssignmentDiagram.png)

### Orchestration

- Choreograph data access, communication and synchronization
- Reduce cost of communication and synchronization
- Preserve data locality (data layout)
- Schedule tasks (order of execution)
- Reduce overhead of managing parallelism
- Must have good primitives (architecture and model)

![Diagram](OrchestrationDiagram.png)
Mapping

- Map processes to physical processors
- Space Sharing
- Static mapping (pin process to processor)
- Dynamic
  - processes migrate under OS control
  - what about orchestration (data locality)?
  - task queues under program control

OS Effects on Mapping

- Ability to bind process to processor
- Space Sharing
  - Physical partitioning of machine
- Gang Scheduling
  - All processes context switched simultaneously
Example: Ocean

- Equation Solver
  - kernel = small piece of heavily used code
- Update each point based on NEWS neighbors
  - Gauss-Seidel (update in place)
- Compute average difference per element
- Convergence when difference is small => exit

Equation Solver Decomposition

while !converged
  for
    for
      • The loops are not independent (linear recurrence)
      • 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

10. procedure Solve (A) /*solve the equation system*/
11. float **A;
12. begin
13. int i, j, done = 0;
14. float diff = 0, temp;
15. while (!done) do /*outermost loop over sweeps*/
16. diff = 0;
17. for i ← 1 to n do /*sweep over nonborder 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

Copyright 1999 Morgan Kaufmann Publishers, Inc
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

• No assignment or orchestration
• Each processor updates its region independent of others’ 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

Equation Solver Assignment

- Each process gets a contiguous block of rows
PARMACS

- Macro Package, runtime system must implement
  - portability

CREATE(p,proc,argvs) Create p processes executing proc(argvs)
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) 
SIGNAL(flag) flag = 1;

Shared Memory Orchestration

10. procedure Solve(A) /*A is entire n-by-n shared array, as in the sequential program*/
11. float **A;
12. begin
13. int i, j, pid, done = 0;
14. float temp, mdiff = 0; /*private variables*/
15. a. int mymin = 1 + (pid * n/proc); /*assume that n is exactly divisible by*/
16. b. int mymax = mymin + n/proc - 1 /*procces for simplicity here*/
17. while (!done) do /*outer loop over all diagonal elements*/
18.     mdiff = diff = 0; /*set global diff to 0 (okay for all to do 0)*/
19.     BARRIER(bar1, proc); /*ensure all reach here before anyone modifies diff*/
20.     for i = mymin to mymax do /*for each of my rows*/
21.         for j = 1 to n do /*for all nonborder elements in that row*/
22.             temp = A(i,j);
25.             mdiff = abs(A[i,j] - temp);
26.         endfor
27.     endfor
28.     UNLOCK(diff_lock); /*update global diff if necessary*/
29.     BARRIER(bar1, proc); /*ensure all reach here before checking if done*/
30.     if (diff < tol) then done = 1; /*check convergence; all get same answer*/
31.     BARRIER(bar1, proc);
32. endwhile
33. end procedure
Equation Solver: The Ugly Code

```c
main()
    parse_arguments();
    A = G_MALLOC(size of big array);
    CREATE(nprocs-1, Solve, A);
    Solve(A)
    WAIT_FOR_END;
end main
Solve(A)
    while done
        for i = my_start to my_end
            for j = 1 to n
                mydiff += abs(A[i,j] - temp);
            LOCK(diff_lock);
            diff += mydiff;
            UNLOCK(diff_lock);
            if (convergence_test) done = 1
        BARRIER
```

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>RCV_PROBE</td>
<td>RCV_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>
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 Orchestration

• 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
Message Passing Orchestration

```c
10. procedure Solve()
11. begin
12. int i, j, pid, n = nprocs, done = 0;
13. float temp, temppdiff, mydiff = 0; /*private variables*/
14. mya = malloc(2-d array of size 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 diff to 0*/
18.     if (pid = 0) then SEND(mya[1,0], n*sizeof(float), pid-1, ROW);
19.     if (pid = nprocs-1) then SEND(mya[n,1], 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 RECEIVE(mya[n,1], n*sizeof(float), pid+1, ROW);
22.     for i = 1 to n do
23.         for j = 1 to n do
24.             temp = mya[i,j] + tempdiff; /*compute new value*/
25.             mya[i,j] = temp; /*update with new value*/
26.         end for
27.     end for
28.     for i = 1 to n do
29.         for j = 1 to n do
30.             temppdiff = mya[i,j] - mya[i,j-1]; /*compute temp diff*/
31.             if temppdiff < 0 then
32.                 mydiff = mydiff + temppdiff; /*accumulate diff*/
33.             end if
34.         end for
35.     end for
36.     if mydiff > 0.0 then
37.         REINDEX(mya, done, int(1, 2)); /*update done flag*/
38.     end if
39.     if done = 1 then done = 0; /*reset done flag*/
40.     for i = 1 to n do
41.         for j = 1 to n do
42.             mya[i,j] = mya[i,j] + mydiff; /*add diff to each entry*/
43.         end for
44.     end for
45.     done = done + 1; /*increment done count*/
46. end while
47. end procedure
```

Copyright 1999 Morgan Kaufmann Publishers, Inc.
Message Passing Orchestration

17. for i ← 1 to n' do /*for each of my (nonghost) rows*/
18. for j ← 1 to n do /*for all nonborder elements in that row*/
19. temp = myA[i,j];
21. mydiff += abs(myA[i,j] - temp);
22. endfor
23. endfor
24. /*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 nproc-1 do /*for each other process*/
25f. RECEIVE(tmpdiff, sizeof(float), i, DIFF);
25g. mydiff += tmpdiff; /*accumulate into total*/
25h. endif
25i. if (mydiff/(n*n) < TOL) then done = 1;
25j. for i ← 1 to nproc-1 do /*for each other process*/
25k. SEND(done, sizeof(int), i, DONE);
25l. endif
25m. endif
26. endwhile
27. end procedure

Example: Programmer / Software

• LocusRoute (standard cell router)

while (route_density_improvement > threshold)
{
    for (i = 1 to num_wires) do
    {
        rip old wire out
        explore new route
        place wire using best new route
    }
}
Shared Memory Implementation

- **Shared memory algorithm**
  - Divide cost array into regions
    - logically assign regions to PEs
  - Assign wires to PEs based on the region in which center lies
  - Do load balancing using stealing when local queue empty
- **Pros:**
  - Good load balancing
  - Mostly local accesses
  - High cache hit ratio
- **Cons:**
  - non-deterministic
  - potential hot spots
  - amount of parallelism

Message Passing Implementations

- **Method 1:**
  - Distribute wires and cost array regions as in SM implementation
  - When wire-path crosses to remote region
    - send computation to remote PE, or
    - send message to access remote data
- **Method 2:**
  - Distribute only wires as in SM implementation
  - Fully replicate cost array on each PE
    - one owned region, and potential stale copy of others
    - send updates so copies are not too stale
  - Consequences:
    - waste of memory in replication
    - stale data => poorer quality results or more iterations
- **Both methods require lots of thought for programmer**
Review: Creating a Parallel Program

- Can be done by programmer, compiler, run-time system or OS
- Steps for creating parallel program
- Decomposition
- Assignment of tasks to processes
- Orchestration
- Mapping