Announcements

- Accounts have been set up
  - last 30 minutes of lecture
  - if someone has registered for the course but did not fill in the account information form, please talk to me after class

- Homework 1 due today

- Homework 2 will be handed out today
  - due on October 8, 1998

- 3 weeks to project proposal due date (October 15)
  - project groups (3-4 students per group)

Outline

- Last lecture
  - parallel programs
    - key steps: decomposition, assignment, orchestration, mapping
    - case studies: Ocean, Raytrace
  - parallel constructs in different programming models
    - data-parallel, shared memory

- This lecture
  - parallel constructs in different programming models (contd.)
    - message passing, summary and comparison
  - analytical models of parallel computation: PRAM, LogP
  - performance issues: naming, synchronization, latency, bandwidth
  - tutorial (last 30 minutes)
    - how to write and run programs on the HP/Convex Exemplar

{ Culler/Singh/Gupta: Chapter 1 (1.3), 3, Almasi/Gottlieb: Chapter 4, LogP paper }

Grid Solver Example

Expression for updating each interior point:
\[ A(i,j) = 0.2 \times (A(i,j) + A[i-1,j] + A[i+1,j] + A[i,j+1] + A[i,j-1]) \]

- Gauss-Seidel (near-neighbor) sweeps to convergence
  - interior \( n \times n \) points of \( (n+2) \times (n+2) \) grid updated in each sweep
    - updates done \textit{in-place}
    - keep track of difference from previous value
  - accumulate partial differences into global difference at end of every sweep
  - do another sweep if error has not converged
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) /*solve the equation system*/
11. begin
12. float **A;
13. /*A is an (n + 2-by-n + 2) array*/
14. int i, j, done = 0;
15. float temp;
16. diff = 0;
17. while (!done) do
18. for i ← 1 to n do
19. for j ← 1 to n do
20. temp = A[i,j]; /*save old value of element*/
22. end for
23. end for
24. if (diff/(n*n) < TOL) then done = 1;
25. end while
26. end procedure

Grid Solver Example: Red-Black Ordering

- Left-to-right, top-to-bottom ordering not fundamental to Gauss-Seidel
- Red-black ordering
  - decompose grid into two sets of points (as in a chess-board)
  - different ordering of updates: may converge quicker or slower
  - red sweep and black sweep are each fully parallel
  - global synchronization between them (conservative but convenient)
  - Exploit additional asynchrony not present in the sequential algorithm
Message Passing Model: Orchestration Support

- Process creation and termination
  - CREATE
  - WAIT_FOR_END

- Communication: data-transfer + synchronization
  - SEND(src_addr, size, dest, tag)
    - send `size` bytes from `src_addr` to `dest` process, with `tag` identifier
  - RECEIVE(buffer_addr, size, src, tag)
    - receive a message of `size` from `src` process with `tag` identifier, and store it in `buffer_addr`
  - SEND_ASYNC, SEND_PROBE, RECEIVE_ASYNC, RECEIVE_PROBE

- Global synchronization
  - BARRIER

Message Passing Model: Grid Solver Example

- Structurally similar to shared memory program (still SPMD), but differs significantly in orchestration
  - data structures and data access/naming
    - cannot declare grid to be a 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
  - communication
    - transfers of entire rows between traversals
  - synchronization

```c
int pid, n, b;
/*process id, matrix dimension and number of processors to be used*/
float **myA;
main()
begin
  read(n);  read(nprocs);
  CREATE (nprocs-1, Solve);
  Solve();
  WAIT_FOR_END (nprocs–1);
end main
procedure Solve()
begin
  int i,j, pid, n' = n/nprocs, done = 0;
  float temp, tempdiff, mydiff = 0;
  /*private variables*/
  myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
  /*my assigned rows of A*/
  initialize(myA);
  /*initialize my rows of A, in an unspecified way*/
  while (!done) do
    mydiff = 0;
    /*set local diff to 0*/
    if (pid != 0) then
      SEND(&myA[1,0],n*sizeof(float),pid-1,
          ROW);
    if (pid = nprocs-1) then
      SEND(&myA[n',0],n*sizeof(float),pid+1,
          ROW);
    if (pid != 0) then
      RECEIVE(&myA[0,0],n*sizeof(float),pid-1,
          ROW);
    if (pid != nprocs-1) then
      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,*]*/
    for i ← 1 to n' do
      /*for each of my (nonghost) rows*/
      for j ← 1 to n do
        /*for all nonborder elements in that row*/
        temp = myA[i,j];
            myA[i,j+1] + myA[i+1,j]);
        mydiff += abs(myA[i,j] - temp);
    endfor
    endfor
    /*communicate local diff values and determine if done; can be replaced by reduction and broadcast*/
    if (pid != 0) then
      SEND(mydiff,sizeof(float),0,
          DIFF);
    if (pid == 0) then
      REDUCE(0,mydiff,sizeof(float),ADD);
      if (mydiff/(n*n) < TOL) then done = 1;
      endfor
      mydiff += tempdiff; /*accumulate into total*/
    if (mydiff/(n*n) < TOL) then done = 1;
    for i ← 1 to nprocs-1 do
      SEND(done,sizeof(int),i,
          DONE);
    endif
    endwhile
end procedure
```

Message Passing Model: Grid Solver (contd.)

- Private portions of grid array
  - use of ghost rows: to store neighbor values
- Core: similar, but indices/bounds in local rather than global space
- Communication
  - receive does not transfer data, send does
  - at beginning of iteration (no asynchrony), whole rows at a time
- Synchronization
  - using sends and receives
  - update of global diff and event synchronization 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*/
### Send and Receive Alternatives

- Can extend functionality
  - stride, scatter-gather, groups
- Semantic flavors: based on when control is returned after call
  - Synchronous
  - Asynchronous

- affect when data structures or buffers can be reused at either end
- affect event synchronization
  - synchronous messages provide synchronization through match
  - separate event synchronization needed with asynchronous messages
- affect ease of programming and performance
- with synchronous messages, our code is deadlocked! Fix?

### Orchestration: Summary

- Data parallel
  - decomposition of data structures (implicit assignment of tasks)
- Shared address space
  - shared and private data explicitly separate
    - no correctness need for data distribution
  - communication implicit in access patterns
  - 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
  - communication is explicit
  - synchronization implicit in communication
    - with synchronous SEND/RECEIVE primitives
    - mutual exclusion for free: only one process updating each address space

### Grid Solver Program

- Decomposition and assignment (partitioning) similar in all three programming models
- Orchestration is different
  - data structures, data access/naming, communication, synchronization

<table>
<thead>
<tr>
<th></th>
<th>Data Parallel</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit global data structure?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment independent of data layout?</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Communication</td>
<td>Implicit</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Implicit</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Explicit replication of border rows?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
High-Performance Parallel Programs

• Tradeoffs between several interacting issues
  – can be addressed/helped by software or hardware

• Models of parallel computation
  – ideal: PRAM
  – realistic: LogP

• Program tuning as successive refinement
  – architecture-independent partitioning
    • view machine as a collection of communicating processors
    • focus: balancing workload, reducing inherent communication & extra work
  – architecture-dependent orchestration
    • view machine as extended memory hierarchy
    • focus: reduce artifactual communication, orchestration overheads

• What are the common issues?

PRAM

• Idealized model of parallel computation
  – collection of P processors and a single memory
  – in one computation step, each processor can perform one operation, read from a memory cell, and write into a memory cell
  – distinctions based on whether or not simultaneous access (particularly stores) permitted to a single memory cell
    • EREW: a cell cannot be simultaneously accessed by two processors
    • CREW: reads are okay, writes are serialized
    • CRCW: most flexible; combining of written results

• Does not model behavior of real parallel machines
  – assumes zero cost of communication
    • infinite bandwidth, zero latency, zero overhead
  – does not model contention
    • simultaneous access permitted to a single memory cell
    • processors are assumed to operate synchronously
    • eliminates need for synchronization primitives

LogP: A More Realistic Model

• Idealized model of parallel computation
  – collection of P processors and a single memory
  – in one computation step, each processor can perform one operation, read from a memory cell, and write into a memory cell
  – distinctions based on whether or not simultaneous access (particularly stores) permitted to a single memory cell
    • EREW: a cell cannot be simultaneously accessed by two processors
    • CREW: reads are okay, writes are serialized
    • CRCW: most flexible; combining of written results
  
• Does not model behavior of real parallel machines
  – assumes zero cost of communication
    • infinite bandwidth, zero latency, zero overhead
  – does not model contention
    • simultaneous access permitted to a single memory cell
    • processors are assumed to operate synchronously
    • eliminates need for synchronization primitives

Implications of LogP

• Eliminates loopholes provided by PRAM-like models
  – communication costs
    • motivates larger-grained applications
    • motivates locality optimizations in algorithms
  – contention for resources is modeled
    • finite-capacity network models network contention
    • g models end-point contention
    • o models occupancy
  – overlap of communication and computation
    • separation of o and L parameters

• LogP is a compromise model which does not take into account
  – caching/replication
  – network topology
  – synchronization overheads
Programming as Successive Refinement

- Not all issues dealt with up front
- Partitioning often independent of architecture, and done first
  - view machine as a collection of communicating processors
    - PRAM + communication costs
    - balancing the workload
    - reducing required amount of inherent communication
- Then, interactions with architecture (orchestration)
  - view machine as extended memory hierarchy
    - extra communication due to architectural interactions
    - cost of communication depends on how it is structured
    - may inspire changes in partitioning
- Our objective is to understand the tradeoffs
  - details in Lectures 4-10

Partitioning for Performance

- 3 major focus areas
  - Balancing the workload + reducing wait time at synchronization points
  - Reducing inherent communication
  - Reducing extra work
- Trade off even among these algorithmic issues
  - minimize communication ⇒ run on 1 processor ⇒ extreme load imbalance
  - maximize load balance ⇒ random assignment of tiny tasks ⇒ no control over communication
  - good partition may imply extra work to compute or manage it
- Goal is to compromise
  - fortunately, often not difficult in practice

Focus 1: Load Balance and Synchronization Time

- Limits on speedup
  \[
  \text{speedup}_{\text{problem}}(p) \leq \frac{\text{sequential work}}{\max(\text{work on any processor})}
  \]
  - work includes data access and other costs
  - not just equal work, but must be busy at the same time
- Four parts to the problem
  - identify enough concurrency
  - decide how to manage it
  - determine the granularity at which to exploit it
  - reduce serialization and cost of synchronization

Identifying Concurrency

- Techniques seen for the Equation Solver kernel
  - loop structure
  - fundamental dependencies (independent of loop structure)
  - new algorithms
In general: Two orthogonal levels of parallelism
- **Function (Task) parallelism**
  - entire large tasks (procedures) can be done in parallel
  - degree usually modest, and does not grow with input size
  - difficult to load balance
- **Data parallelism**
  - more scalable: proportional to input size
  - function parallelism can reduce synchronization between data parallel phases
Managing Concurrency

Static versus Dynamic techniques

• Static techniques
  – algorithmic assignment based on input: does not change
  – low run-time overhead, but requires predictable computation
  – preferable when applicable
    caveat: multiprogrammed/heterogeneous environments

• Dynamic techniques
  – adapt at run time to balance load
  – but, can increase communication and task management overheads

Determining Task Granularity

• Task granularity: amount of work associated with a task
  – scaled with respect to parallelism overheads in the system
    • communication, synchronization, etc.

• General rule:
  – coarse-grained ⇒ often poor load balance
  – fine-grained ⇒ more overhead, often more communication,
    requires more synchronization (contention)

• Overheads influenced by both task size, and assignment
  – dynamic tasking requires a threshold task size

Reducing Serialization

• Influenced by assignment and orchestration (includes how tasks are
  scheduled on physical resources)

• Event synchronization
  – conservative (global) versus point-to-point synchronization
    • e.g., barriers versus locks
  – however, fine-grained synchronization more difficult to program and can
    produce more synchronization operations

• Mutual exclusion
  – main goal is to reduce contention: separate locks for separate data
  – smaller critical sections
  – stagger critical sections in time
Implications of Load Balance

- Extends speedup limit expression to
  \[ \text{speedup}_{\text{problem}}(p) \leq \frac{\text{sequential work}}{\max \left( \text{work on any processor} + \text{synchronization wait time} \right)} \]

- Generally, the responsibility of the programmer
  - algorithmic decisions, based on fairly simple machine model
    - PRAM + communication has non-zero cost

- How can architecture help?
  - fine-grained communication (low overhead, latency)
    - allows smaller tasks, better load balance (low-overhead access to queues)
  - naming logically shared data in the presence of task stealing
    - need to access data of stolen tasks
    - hardware shared address space advantageous

Focus 2: Reducing Inherent Communication

- Simple machine view: communication is expensive!
  \[ \text{speedup}_{\text{problem}}(p) \leq \frac{\text{sequential work}}{\max \left( \text{work on any processor} + \text{synchronization wait time} + \text{communication costs} \right)} \]

  - metric: communication to computation ratio
  - provides guidance on which communication aspect is important
    - if computation is execution time, ratio gives average BW need
    - if computation is operation count, gives extremes in impact of latency and BW
      - latency: assume no latency hiding
      - bandwidth: assume all latency is hidden
      - real-life is somewhere in between

  - Solution: assign tasks that access same data to same process
    - solving communication and load balance is NP-hard (in general)
  - however, simple heuristic solutions work well
    - exploit application structure: e.g., domain decomposition

Focus 3: Reducing Extra Work

- Extends speedup limit expression
  \[ \text{speedup}_{\text{problem}}(p) \leq \frac{\text{sequential work}}{\max \left( \text{work on any processor} + \text{synchronization wait time} + \text{communication costs} + \text{extra work} \right)} \]

- Common sources of extra work
  - computing a good partition (e.g., in a sparse matrix computation)
  - using redundant computation to avoid communication
  - task, data, and process management overhead
    - applications, languages, run-time systems, OS
  - imposing structure on communication
    - coalescing messages, allowing effective naming

- How can architecture help?
  - efficient support of communication and synchronization (orchestration)

Architecture-independent Partitioning: Summary

- Useful for early development
  - focus on partitioning and mapping
    - understanding algorithm structure
    - simple machine model: ideal (PRAM) + non-zero communication cost

- However, unrealistic for real performance
  - simple view of machine does not model communication accurately
    - wrongly models direct costs as well as imbalances
    - partially addressed by more realistic models such as LogP

- Moreover, communication costs determined not only by amount
  - depends on structuring of communication (naming, synchronization)
  - cost of communication in system (latency, bandwidth)
    - common set of issues helped/addressed by both programming model and parallel architecture
Memory-oriented View of a Multiprocessor

- Multiprocessor as an extended memory hierarchy
  - levels: registers, caches, local memory, remote memory (topology)
    - glued together by communication architecture
  - levels communicate at a certain granularity of data transfer
  - differences in access costs and bandwidth
  - need to exploit spatial and temporal locality in hierarchy
    - similar to uniprocessors: extra communication-high communication costs
    - trade off against partitioning goals

Artificial Communication Costs

Accesses not satisfied in local hierarchy levels cause communication

- Inherent
  - determined by program
  - assumes unlimited capacity, small transfers, perfect knowledge
- Artifactual
  - determined by program implementation and architecture interactions
  - some reasons:
    - poor allocation of data across distributed memories
    - redundant communication of data
    - unnecessary data in a transfer or unnecessary transfers (system granularity)
    - finite replication capacity
      - four kinds of cache misses: compulsory, capacity, conflict, coherence
      - finite capacity affects capacity and conflict misses
    - tradeoff between reducing artificial communication cost and improving spatial locality

Orchestration for Performance

Two areas of focus
- Reducing amount of communication
  - inherent: change logical data sharing patterns in algorithm
  - artifactual: exploit spatial, temporal locality in extended hierarchy
    - techniques often similar to those on uniprocessors
    - shared address space machines support this in hardware, distributed memory machines support the same techniques in software
- Structuring communication to reduce cost

Reducing Amount of Communication

- Exploiting temporal locality
  - structure communication so working sets map well to hierarchy
  - More useful when O(n^(k+1)) computation with O(n^k) data (factorization)
- Exploiting spatial locality
  - system granularity
  - tradeoffs with reducing inherent communication
    - block vs. row decomposition
Structuring Communication to Reduce Cost

\[
\text{communication cost} = f \left( a + l + \frac{n_c}{m} + \frac{t_c - \text{overlap}}{B} \right)
\]

- frequency of messages
- message overhead
- network delay per message
- cost induced by contention
  - in the network
  - end-point contention
- \(n_c\): total data sent
- \(m\): number of messages
- \(B\): bandwidth along path
- portion of latency that can be overlapped

Summary of Performance Tradeoffs

- Load balance [synchronization wait time]
  - fine-grain tasks
  - random or dynamic assignment
- Inherent communication volume [data access costs]
  - coarse-grained tasks
  - tension between locality and load balance
- Extra work [processor overheads + data access costs]
  - coarse-grained tasks
  - simple assignment
- Artifactual communication costs [data access costs]
  - big transfers: amortize overhead and latency
  - small transfers: reduce contention and occupancy

Efficient naming, synchronization, and communication reduce incentive for creating ill-behaved programs

Lecture Summary

- High performance parallel programs
  - models of parallel computation
    - ideal: PRAM
    - realistic: LogP
  - programming as successive refinement
    - architecture-independent partitioning
      - machine is viewed as a collection of communicating processors
      - balance workload, reduce inherent communication, reduce extra work
    - tug-of-war even among these issues
    - architecture-dependent orchestration
      - machine is viewed as an extended memory hierarchy
      - artifactual communication costs
  - common issues: naming, synchronization, latency, bandwidth
  - issues can be addressed/helped by hardware or software

Next Lecture

- Small-scale shared memory machines
  - bus-based architectures
  - snoopy cache-coherence protocols
  - case study: Convex Exemplar
- Tutorial
  - programming with threads

Readings

- Culler/Singh/Gupta: Chapter 4
- Almasi/Gottlieb: Chapter 10 (Sections 10.3.1, 10.3.2)