Announcements

- Accounts have been set up
  - last 20 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 is being handed out today
  - due on October 10, 2001

- 3 weeks to project proposal due date (October 17)
  - project groups (2-3 students per group)

Outline

- Last lecture
  - parallel programs
    - key steps: decomposition, assignment, orchestration, mapping
    - case studies: Ocean, Raytrace

- This lecture
  - parallel constructs in different programming models
    - data-parallel, shared memory, message passing
  - analytical models of parallel computation: PRAM, LogP
  - performance issues: naming, synchronization, latency, bandwidth
  - tutorial (last 20 minutes)
    - pthreads library, using the course machines, ...

[ Culler/Singh/Gupta: Chapter 1 (1.3), 3; Andrews: 1-5 (intro material); LogP paper ]

Grid Solver Example

Expression for updating each interior point:

\[
\]

- 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 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

[ Culler/Singh/Gupta: Chapter 1 (1.3), 3; Andrews: 1-5 (intro material); LogP paper ]
Grid Solver Example: Decomposition

- Option 1: Look at loop iterations
  - dependence analysis: if not enough concurrency, then look further
  - not much concurrency here at this level (both loops are sequential)

- Option 2: Examine fundamental dependences, ignoring loop structure
  - concurrency \(O(n)\) along anti-diagonals (southwest to northeast)
  - serialization \(O(n)\) along diagonal

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
Grid Solver Example: Code for Decomposition

- Simpler example of asynchrony: no dependencies

```plaintext
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
```

- for_all leaves assignment to the system
- but implicit global synch. at end of for_all loop
- as shown: a task is a single grid point, so O(n^2) tasks
- to decompose into rows: make line 18 loop sequential; O(n) tasks

Grid Solver Example: Assignment

- Static assignment (given decomposition into rows)
  - block: row i is assigned to process

```
\[
\begin{array}{cccccccc}
  p_0 & p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
```

- cyclic: process j is assigned rows j, j+p, ...
- block assignment reduces communication requirement

- Dynamic assignment
  - get a row index, work on the row, get a new row, and so on

Grid Solver Example: Orchestration

- Requirements
  - communication
    - values of neighboring grid points must be available
  - synchronization
    - next iteration (or alternating red-black sweeps) cannot proceed until all grid points have been evaluated in the current sweep

- Language support for orchestration is programming model specific
  - data parallel
    - concurrent loops, computation and data structure decomposition, collective operations
  - shared memory
    - process creation, mutual exclusion, global synchronization, post-wait
  - message passing
    - process creation, synchronous and asynchronous send/receive, global synchronization

Data Parallel Models: Orchestration Support

- Dynamic allocation of shared data
  - G_MALLOC (global malloc)

- Concurrent loops
  - for_all
    - parallel processes are implicitly active: only within for_all body

- Decomposition of data and computation
  - DECOMP arr[BLOCK, *, nprocs]
    - specifies assignment of data elements to processes
    - owner-computes: specifies assignment of iterations to processes

- Collective operations
  - REDUCE, others such as broadcast, etc.
    - all-to-all operations, implemented efficiently by the underlying system
Shared Memory Models: Orchestration Support

- Process creation and termination
  - `CREATE(p, proc, args)`
  - `WAIT_FOR_END(number)`
- Dynamic allocation of shared data
  - `G_MALLOC`
- Mutual exclusion
  - `LOCK(name)`: acquire mutually exclusive access
  - `UNLOCK(name)`: release access
- Global synchronization
  - `BARRIER(name, number)`: no process gets past barrier until `number` have arrived
- Point-to-point synchronization
  - `WAIT(flag)`: wait for flag to be set (spin or block)
  - `SIGNAL(flag)`: set flag, wake up waiting processes

Shared Memory Model: Grid Solver Example

- Grid declared as a shared array
  - all processes can access it just as in sequential program
- Single program multiple data (SPMD) style
  - assignment controlled by values of variables used as loop bounds

```
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)
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;
15. DECOMP A[BLOCK,*, nprocs];
16. while (!done) do
17. mydiff = 0; /*initialize maximum difference to 0*/
18. for_all i ← 1 to n do /*sweep over non-border points of grid*/
19. for_all j ← 1 to n do
20. 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) < TOL) then done = 1; /*check convergence; all get same answer*/
26. endwhile
27. end procedure
```
Shared Memory Model: Grid Solver (contd.)

- Single program multiple data
  - not lockstep or even necessarily same instructions
  - assignment of iterations controlled by values of variables
  - done condition evaluated redundantly by all
  - code that does the update identical to sequential program
    - since grid array is in shared address space
    - each process has private mydiff variable
- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why all the barriers?

Grid Solver: 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

\[
P_1 \quad r_1 \leftarrow \text{diff} \quad P_2 \quad r_1 \leftarrow \text{diff}
\]

\[
P_1 \quad r_1 \leftarrow r_1 + r_2 \quad P_2 \quad r_1 \leftarrow r_1 + r_2
\]

\[
\text{diff} \leftarrow r_1 \quad \text{diff} \leftarrow r_1
\]

- need the sets of operations to be atomic (mutually exclusive)
- use of mydiff reduces contention for the lock

Grid Solver: Need for Barriers

- Line 25d
  - ensures that all processes have updated diff
  - needed for ensuring that the tolerance check is correct
- Line 25f
  - ensures that each process waits for all others to get done before exiting
  - otherwise, diff can be reset by a faster process
- Line 16a
  - ensures that there is no race condition between Lines 16 and 25b
  - else, an arbitrary slow process can reset diff after a faster process has updated it for the next iteration
- Barriers are a form of many-to-many synchronization
  - WAIT/SIGNAL is an example of one-to-many synchronization
  - WAIT_FOR_END is an example of many-to-one synchronization

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

```
1. int pid, n, b;
   /*process id, matrix dimension and number of processors to be used*/
2. float **myA;
3. main()
   begin
   4. read(n); read(nprocs);
      /*read input matrix size and number of processes*/
6a. CREATE (nprocs-1, Solve);
6b. Solve()->/*task process becomes a worker too*/
6c. WAIT_FOR_END (nprocs-1);
      /*wait for all child processes created to terminate*/
9. end main
10. procedure Solve()
    begin
    11. int i, j, pid, n' = n/nprocs, done = 0;
12. float temp, tempdiff, mydiff = 0;
      /*private variables*/
6. myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
      /*my assigned rows of A*/
7. initialize(myA);
      /*initialize my rows of A, in an unspecified way*/
15. while (!done) do
    16. mydiff = 0;
       /*set local diff to 0*/
    16a. if (pid != 0) then
       SEND (&myA[1,0],n*sizeof(float),pid-1,ROW);
    16b. if (pid = nprocs-1) then
       SEND (&myA[n',0],n*sizeof(float),pid+1,ROW);
    16c. if (pid != 0) then
       RECEIVE (&myA[0,0],n*sizeof(float),pid-1,ROW);
    16d. 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,*]*/
    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];
       myA[i,j+1] + myA[i+1,j]);
    21. mydiff += abs(myA[i,j] - temp);
    22. endfor
    23. endfor
    24. endwhile
    25a. if (pid != 0) then
       /*process 0 holds global total diff*/
    25b. REDUCE(0,mydiff,sizeof(float),ADD);
    25c. if (pid == 0) then
       25d. if (mydiff/(n*n) < TOL) then done = 1;
       25e. for i ← 1 to nprocs-1 do
          RECEIVE(tempdiff,sizeof(float),i,DIFF);
       25f. mydiff += tempdiff;
       25g. endif
    25h. endfor
    25i. if (mydiff/(n*n) < TOL) then done = 1;
   25j. for i ← 1 to nprocs-1 do
       SEND(done,sizeof(int),i,DONE);
    25k. endfor
    25l. endif
    25m. endif
   25n. endif
   26. 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**
25b. REDUCE(0,mydiff,sizeof(float),ADD);
25c. if (pid == 0) then
25i. if (mydiff/(n*n) < TOL) then done = 1;
25m. BROADCAST(0,done,sizeof(int),DONE);
```

Send and Receive Alternatives

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

```
Send/Receive
Synchronous
Blocking asynch. Nonblocking asynch.
```

- 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 separated
  - 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

Parallel Programs: Summary

- Key steps:
  - decomposition: identify concurrent tasks
  - assignment: decide which task is performed by which process
  - orchestration: handle required communication and synchronization
  - mapping: decide which process executes on which processor

- Parallel constructs in different programming models
  - data parallel: array decomposition, concurrent loops, collective operations
  - message passing: process creation, send/receive, global synchronization
  - shared memory: process creation, mutual exclusion, global and point-to-point synchronization

- What about performance?

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 artificial communication, orchestration overheads

- What are the common issues?

PRAM: Idealized Model of Parallel Machines

- 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

- Good model for algorithm development, but …
- 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
  - assumes synchronous operation of processors
    - eliminates need for synchronization primitives
LogP: A More Realistic Model

- 4 parameters
  - $L$: an upper bound on the latency
  - $o$: the overhead for transmission or reception of a message
    - the processor cannot perform other operations for this duration
  - $g$: the gap, defined as the minimum time interval between consecutive
    message transmissions or consecutive message receptions
    - $1/g$ equals per-processor communication bandwidth
    - network has finite capacity: $L/g$ messages from/to a processor
  - $P$: the number of processor/memory modules
    - local operations are assumed to take unit time

“time to squeeze message onto thinnest wire”: $n/BW = n^*g$

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

- However, LogP is still a compromise model not accounting for
  - 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 processors 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

Managing Concurrency (contd.)

- Dynamic techniques
  - profile-based (semi-static)
    - profile work distribution at run time and repartition dynamically
  - dynamic tasking
    - pool of tasks; take and add tasks until done

\[\begin{align*}
\text{All processes} & \quad \text{insert tasks} \\
\text{Q0} & \\
\text{All remove tasks} \\
& \\
& \\
& \\
& \\
\text{Others may steal} \\
& \\
& \\
& \\
& \\
& \\
\end{align*}\]

(a) Centralized task queue

\[\begin{align*}
\text{P0 inserts} & \\
\text{Q1 inserts} & \\
\text{Q2 inserts} & \\
\text{Q3 inserts} & \\
\text{P0 removes} & \\
\text{Q1 removes} & \\
\text{Q2 removes} & \\
\text{Q3 removes} & \\
\end{align*}\]

(b) Distributed task queues (one per process)
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
  \[
  \text{speedup}_{\text{problem}}(p) \leq \frac{\text{sequential work}}{\text{max}\left(\frac{\text{work on any processor} + \text{synchronization wait time}}{\text{communication costs}}\right)}
  \]

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

- 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 unprocessors: extra communication high communication costs
    - trade off against partitioning goals

Artifactual Communication Costs

Accesses not satisfied in local hierarchy levels cause communication

- Inherent
  - determined by program
  - assuming 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 artifactual 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

- Very brief introduction in the remainder of the lecture
  - rest of the course focuses primarily on this aspect
    - Lectures 4-7 focus on hardware approaches
    - Lectures 8-11 focus on software and hybrid approaches

Reducing Amount of Communication

- Exploiting temporal locality
  - structure program so working sets map well to hierarchy
  - More useful when $O(nk+1)$ computation with $O(n^2)$ data (factorization)

- Exploiting spatial locality
  - system granularity
  - tradeoffs with reducing inherent communication
    - block vs. row decomposition

Structuring Communication to Reduce Cost

$\text{communication cost} = f \left( o + l + \frac{n_d}{m} \frac{m}{B} + t_c - \text{overlap} \right)$

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

Summary of Performance Tradeoffs

- Load balance $[\text{synchronization wait time}]$
  - fine-grain tasks
  - random or dynamic assignment

- Inherent communication volume $[\text{data access costs}]$
  - coarse-grained tasks
  - tension between locality and load balance

- Extra work $[\text{processor overheads + data access costs}]$
  - coarse-grained tasks
  - simple assignment

- Artifactual communication costs $[\text{data access costs}]$
  - big transfers: amortize overhead and latency
  - small transfers: reduce contention and occupancy

$\text{Efficient naming, synchronization, and communication operations allow one to achieve performance without getting everything right}$
Lecture Summary

- High performance parallel programs
  - models of parallel computation
    - ideal: PRAM
    - realistic: LogP
  - programming as successive refinement
    - architecture-independent partitioning
      - balance workload, reduce inherent communication, reduce extra work
    - architecture-dependent orchestration
      - machine is viewed as an extended memory hierarchy
        » artificial 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: Sun Enterprise

Readings

- Culler/Singh/Gupta: Chapter 4