Metrics in Application Performance

- **Speedup**
  - ratio of time on n nodes to time on a single node
  - hold problem size fixed
  - should really compare to best serial time
  - goal is linear speedup
  - super-linear speedup is possible due to:
    - adding more memory
    - search problems

- **Iso-Speedup**
  - scale data size up with number of nodes
  - goal is a flat horizontal curve

- **Amdahl's Law**
  - max speedup is $1/(\text{serial fraction of time})$

- **Computation to Communication Ratio**
  - goal is to maximize this ratio
How to Write Parallel Programs

- **Use old serial code**
  - compiler converts it to parallel
  - called the dusty deck problem

- **Serial Language plus Communication Library**
  - no compiler changes required!
  - PVM and MPI use this approach

- **New language for parallel computing**
  - requires all code to be re-written
  - hard to create a language that provides performance on different platforms

- **Hybrid Approach**
  - HPF - add data distribution commands to code
  - add parallel loops and synchronization operations
Application Example - Weather

- **Typical of many scientific codes**
  - computes results for three dimensional space
  - compute results at multiple time steps
  - uses equations to describe physics/chemistry of the problem
  - grids are used to discretize continuous space
    - granularity of grids is important to speed/accuracy

- **Simplifications (for example, not in real code)**
  - earth is flat (no mountains)
  - earth is round (poles are really flat, earth buldges at equator)
  - second order properties
Grid Points

• **Divide Continuous space into discrete parts**
  - for this code, grid size is fixed and uniform
    • possible to change grid size or use multiple grids
  - use three grids
    • two for latitude and longitude
    • one for elevation
    • Total of $M \times N \times L$ points

• **Design Choice: where is the grid point?**
  - left, right, or center of the grid
    - in multiple dimensions this multiples:
      • for 3 dimensions have 27 possible points
Variables

- **One dimensional**
  - $m$ - geo-potential (gravitational effects)

- **Two dimensional**
  - $\pi$ - "shifted" surface pressure
  - $\text{sigmadot}$ - vertical component of the wind velocity

- **Three dimensional (primary variables)**
  - $\langle u, v \rangle$ - wind velocity/direction vector
  - $T$ - temperature
  - $q$ - specific humidity
  - $p$ - pressure

- **Not included**
  - clouds
  - precipitation
  - can be derived from others
Serial Computation

- Convert equations to discrete form
- Update from time $t$ to $t + \text{delta } t$

```plaintext
deforeach longitude, latitude, altitude
    ustar[i,j,k] = n * pi[i,j] * u[i,j,k]
    vstar[i,j,k] = m[j] * pi[i,j] * v[i,j,k]
    sdot[i,j,k] = pi[i,j] * sigmadot[i,j]
end
deforeach longitude, latitude, altitude
    D = 4 * ((ustar[i,j,k] + ustar[i-1,j,k]) * (q[i,j,k] + q[i-1,j,k]) +
    terms in {i,j,k} {+, -} {1, 2})
    piq[i,j,k] = piq[i,j,k] + D * delat
    similar terms for piu, piv, piT, and pi
end
deforeach longitude, latitude, altitude
    q[i,j,k] = piq[i,j,k]/pi[i,j,k]
    u[i,j,k] = piu[i,j,k]/pi[i,j,k]
    v[i,j,k] = piv[i,j,k]/pi[i,j,k]
    T[i,j,k] = piT[i,j,k]/pi[i,j,k]
end
```
Shared Memory Version

- in each loop nest, iterations are independent
- use a parallel for-loop for each loop nest
- synchronize (barrier) after each loop nest
  - this is overly conservative, but works
  - could use a single sync variable per item, but would incur excessive overhead
- potential parallelism is $M \times N \times L$
- private variables: $D, i, j, k$

Advantages of shared memory
- easier to get something working (ignoring performance)

Hard to debug
- other processors can modify shared data
Distributed Memory Weather

- decompose data to specific processors
  - assign a cube to each processor
    - maximize volume to surface ratio
    - minimizes communication/computation ratio
  - called a $<$block,block,block$>$ distribution

- need to communicate $\{i,j,k\}\{+,-\}\{1,2\}$ terms at boundaries
  - use send/receive to move the data
  - no need for barriers, send/receive operations provide sync
    - sends earlier in computation too hide communication time

- Advantages
  - easier to debug?
  - consider data locality explicitly with data decomposition

- Problems
  - harder to get the code running
Seismic Code

● Given echo data, compute under sea map
● Computation model
  – designed for a collection of workstations
  – uses variation of RPC model
  – workers are given an independent trace to compute
    • requires little communication
    • supports load balancing (1,000 traces is typical)
● Performance
  – max mfops = O((F \* nz \* B^*)^{1/2})
  – F - single processor MFLOPS
  – nz - linear dimension of input array
  – B^* - effective communication bandwidth
    • B^* = B/(1 + BL/w) \approx B/7 for Ethernet (10msec lat., w=1400)
  – real limit to performance was latency not bandwidth
Database Applications

- Too much data to fit in memory (or sometimes disk)
  - data mining applications (K-Mart has a 4-5TB database)
  - imaging applications (NASA has a site with 0.25 petabytes)
    - use a fork lift to load tapes by the pallet

- Sources of parallelism
  - within a large transaction
  - among multiple transactions

- Join operation
  - form a single table from two tables based on a common field
  - try to split join attribute in disjoint buckets
    - if know data distribution is uniform its easy
    - if not, try hashing
Speedup in Join parallelism

- **Books claims a speed up of** $1/p^2$ **is possible**
  - split each relation into $p$ buckets
    - each bucket is a disjoint subset of the joint attribute
  - each processor only has to consider $N/p$ tuples per relation
    - join is $O(n^2)$ so each processor does $O((N/p)^2)$ work
    - so speedup is $O(N^2/p^2)/O(N^2) = O(1/p^2)$

- **this is a lie!**
  - could split into $1/p$ buckets on one processor
  - time would then be $O(p \times (N/p)^2) = O(N^2/p)$
  - so speedup is $O(N^2/p^2)/O(N^2/p) = O(1/p)$
    - Amdahls law is not violated
Parallel Search (TSP)

- may appear to be faster than 1/n
  - but this is not really the case either

- Algorithm
  - compute a path on a processor
    - if our path is shorter than the shortest one, send it to the others.
    - stop searching a path when it is longer than the shortest.
  - before computing next path, check for word of a new min path
  - stop when all paths have been explored.

- Why it appears to be faster than 1/n speedup
  - we found the a path that was shorter sooner
  - however, the reason for this is a different search order!
Ensuring a fair speedup

- \( T_{\text{serial}} = \text{faster of} \)
  - best known serial algorithm
  - simulation of parallel computation
    - use parallel algorithm
    - run all processes on one processor
  - parallel algorithm run on one processor

- If it appears to be super-linear
  - check for memory hierarchy
    - increased cache or real memory may be reason
  - verify order operations is the same in parallel and serial cases
Quantitative Speedup

- Consider master-worker
  - one master and \( n \) worker processes
  - communication time increases as a linear function of \( n \)
  
  \[ T_p = T_{COMP_p} + T_{COMM_p} \]
  
  \[ T_{COMP_p} = \frac{T_s}{P} \]
  
  \[ \frac{1}{S_p} = \frac{T_p}{T_s} = \frac{1}{P} + \frac{T_{COMM_p}}{T_s} \]
  
  \[ T_{COMM_p} \text{ is } P \times T_{COMM_1} \]
  
  \[ \frac{1}{S_p} = \frac{1}{P} + P \times \frac{T_{COMM_1}}{T_s} = \frac{1}{P} + \frac{P}{r_1} \]
  
  where \( r_1 = \frac{T_s}{T_{COMM_1}} \)
  
  \[ \frac{d(1/S_p)}{dP} = 0 \quad \Rightarrow \quad P_{opt} = r_1^{1/2} \text{ and } S_{opt} = 0.5 \times r_1^{1/2} \]

- For hierarchy of masters
  - \( T_{COMM_p} = (1 + \log P)T_{COMM_1} \)
  - \( P_{opt} = r_1 \) and \( S_{opt} = r_1/(1 + \log r_1) \)
MPI

**Goals:**
- Standardize previous message passing:
  - PVM, P4, NX
- Support copy free message passing
- Portable to many platforms

**Features:**
- point-to-point messaging
- group communications
- profiling interface: every function has a name shifted version

**Buffering**
- no guarantee that there are buffers
- possible that send will block until receive is called

**Delivery Order**
- two sends from same process to same dest. will arrive in order
- no guarantee of fairness between processes on recv.
MPI Communicators

- Provide a named set of processes for communication
- All processes within a communicator can be named
  - numbered from 0…n-1
- Allows libraries to be constructed
  - application creates communicators
  - library uses it
  - prevents problems with posting wildcard receives
    * adds a communicator scope to each receive
- All programs start will MPI_COMM_WORLD
Non-Blocking Functions

- **Two Parts**
  - post the operation
  - wait for results
- **Also includes a poll option**
  - checks if the operation has finished
- **Semantics**
  - must not alter buffer while operation is pending
MPI Misc.

- **MPI Types**
  - All messages are typed
    - base types are pre-defined:
      - int, double, real, {,unsigned}{short, char, long}
    - can construct user defined types
      - includes non-contiguous data types

- **Processor Topologies**
  - Allows construction of Cartesian & arbitrary graphs
  - May allow some systems to run faster

- **What’s not in MPI-1**
  - process creation
  - I/O
  - one sided communication
MPI Housekeeping Calls

- Include `<mpi.h>` in your program
- If using mpich, ...

- First call `MPI_Init(&argc, &argv)`
- `MPI_Comm_rank(MPI_COMM_WORLD, &myrank)`
  - Myrank is set to id of this process
- `MPI_Wtime`
  - Returns wall time
- At the end, call `MPI_Finalize()`