Scalability, and Isoefficiency
Efficiency, Speedups and How to Compare Parallel Algorithms
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Major Analytical/Theoretical Techniques

• Typically involves simple algebraic formulae and ratios
  • Typical variables are:
    • Data size (N), number of processors (P), machine constants
    • Model performance of individual operations, components, algorithms in terms of the above
    • Be careful to characterize variations across processors, and model them with (typically) max operators
      • E.g., max\{Load_i\}
  • Remember that constants are important in practical parallel computing
    • Be wary of asymptotic analysis: use it, but carefully

L.V.Kale
Alpha-Beta Cost Model for Communication

• Communication costs involve per-message cost $\alpha$ and per-byte cost $\beta$
  • These may arise in different contexts, each with its own $\alpha/\beta$ values
    • Incurred by CPU in processing messages
    • Latency on the network

• Execution time for the program may involve
  • Sum of computation and communication
  • Max of computation and communication costs if overlapped
    • With irecv, isend
  • Critical paths
    • E.g., reduction. Each processor may be sending a few messages, but the operation takes $\log P$ time, and you may be waiting for that duration

• Alternative model: $\log P$ (and $\log NP$)
  • I prefer $\alpha - \beta$ model, as long as it is applied separately for overhead and latency when needed
Scalability

• The program should scale up to use a large number of processors
  • But what does that mean?

• Any fixed individual simulation (computation) isn’t truly scalable

• Why?
  • Because with a fixed data (and computation) size, it will run out of parallelism eventually
Parallel Efficiency

• Efficiency: Sequential Time / (P \cdot Parallel Time)
  • Parallel time = computation + communication + idle
• Speedup = Sequential Time / Parallel Time
Amdahl’s Law

• Assume that you have an algorithm, only portions of it are parallelizable
  • OpenMP: work on the master thread is not parallelized

• What can we say about the speedup?

• Speedup can never be more than the ratio of total computation time and sequential portion
  • Sequential Execution time = $T_1 + T_{seq}$
  • Parallel Execution time = $\frac{T_1}{p} + T_{seq}$
  • Speedup = $(T_1 + T_{seq})/(\frac{T_1}{p} + T_{seq})$
  • In the limit, $p \rightarrow \infty$: $\frac{T_1}{p} \rightarrow 0$; speedup = $(T_1 + T_{seq})/T_{seq}$
Scalability, and Isoefficiency: continued

Efficiency, Speedups and How to Compare Parallel Algorithms
Strong Scaling

• Given a particular problem instance
  • I.e., a program with a given input

• How many processors can we use profitably?
  • What is “profitably”? 

• Depends on if your goal is:
  • Best use of resources
    • Middle region is not ok
  • Shorter completion time
    • Important problem with a deadline
    • Middle region is ok
Scalability

• The program should scale up to use a large number of processors
  • But what does that mean?

• An individual simulation isn’t truly scalable

• Better definition of scalability:
  • If I double the number of processors, I should be able to retain parallel efficiency by increasing the problem size
Isoefficiency

• Quantify scalability

• How much increase in problem size is needed to retain the same efficiency on a larger machine?

• Efficiency: Sequential Time / (P · Parallel Time)
  • Parallel time = computation + communication + idle

• One way of analyzing scalability:
  • Isoefficiency:
    • Equation for equal-efficiency curves
    • Use $\eta(p,N) = \eta(x,p, y.N)$ to get this equation
      • $\eta(p,N) = C$ for some constant C
    • If no solution: the problem is not scalable
      • In the sense defined by isoefﬁciency
Isoefficiency Basics

• For this class, we will focus on communication volume as the impediment to scalability (most of the time, unless stated otherwise)
  • I.e., number of bytes sent or received

• So, the method for deriving iso-efficiency function is:
  • Write an expression for communication cost: $T_{comm}$
  • Write an expression for computation cost: $T_{comp}$
  • Express both in terms of total (sequential) computation work: $W$
  • Write an expression for the ratio $\frac{T_{comm}}{T_{comp}}$ and equate to a constant
    • So the efficiency can remain constant
  • Solve for $W$ in terms of $P$ (and eliminate all other variables)
    • Approximate expressions as needed
Approximations

• If the computation cost is $a.N^3 + b.N$
  • Approximate this as $a.N^3$, because the cubic term dominates over the linear term, for large $N$
  • $N^2 + k.N \cdot \log N$ approximates to $N^2$
Make Sure to Use the Total Work $W$

• Example:
  • For a matrix of size $N$, say, the algorithm requires doing $N^3$ operations (across all processors (i.e., same as sequential if you are not doing extra work))
  • $W = N^3$; i.e., $N = W^{1/3}$
  • Make sure you replace all occurrences of $N$ in the comm/comp ratio by $W^{1/3}$
Using Other Metrics for Isoefficiency

• Sometimes, the communication volume is not the metric for measuring inefficiency
  • May be:
    • Number of messages
    • Fraction of bisection bandwidth used
    • Critical path

• Use the dominant metric for overhead to determine equal efficiency equation
  • \( T_{\text{parallel}} = T_{\text{comp}} + T_{\text{overhead}} \)
Gauss-Jacobi Relaxation

Sequential Pseudocode:

```java
while (maxError > Threshold) {
    Re-apply Boundary conditions
    maxError = 0;
    for i = 0 to N-1 {
        for j = 0 to N-1 {
            if (|B[i,j] - A[i,j]| > maxError)
                maxError = |B[i,j] - A[i,j]|
        }
    }
    swap B and A
}
```

Decomposition by:
- Row
- Blocks
- Or Column
Isoefficiency of Jacobi Relaxation

Row decomposition

• Computation per proc:

• Communication:

• Comm-to-comp ratio:

• Efficiency:

• Isoefficiency:

Block decomposition

• Computation per proc:

• Communication:

• Comm-to-comp ratio:

• Efficiency:

• Isoefficiency:
Isoefficiency of Jacobi Relaxation

Row decomposition

• Computation per proc:
  • A * N * (N/P)

• Communication:
  • 16 * N

• Comm-to-comp ratio:
  • \( (16 \times P) / (A \times N) = \gamma \)

• Efficiency: \( 1 / (1 + \gamma) \)

• Isoefficiency:
  • Keep \( \gamma \) constant
  • \( 16P/(A.N) = \gamma \), ie \( P/N = k \)
  • problem-size \( W = N^2 \), \( N = \sqrt{W} \)
  • \( P/\sqrt{W} = k; W = (P/k)^2 \)

Block decomposition

• Computation per proc:
  • A * N * (N/P)

• Communication:
  • \( 32 \times N / P^{1/2} \)

• Comm-to-comp ratio:
  • \( (32 \times P^{1/2}) / (A \times N) \)

• Efficiency: \( 1 / (1 + \gamma) \)

• Isoefficiency:
  • \( P^{1/2}/(w)^{1/2} = k \); \( w = p/k \)
  • Linear in \( p \)
Scalability, and Isoefficiency: continued

Isoefficiency for Classical Molecular Dynamics Algorithms

© 2018 L. V. Kale at the University of Illinois Urbana-Champaign
Simple Molecular Dynamics

• Collection of [charged] atoms, with bonds
  • Newtonian mechanics
  • Thousands to millions of atoms
  • 1 femtosecond time-step, millions needed!

• At each time step
  • Calculate forces on each atom
    • Bonds: treated like springs
    • Non-bonded: electrostatic and van der Waal’s
      • Short distance: every time step
      • Long range: ignore for this discussion
  • Calculate velocities and advance positions
The Basic Algorithm

Read atom info and initialize
for each timestep {
  for i = 1, N
    for j = 1, N
      if (distance(A[i],j) < CutOff)
        A[i].force = calculateForceBetween(A[i],j)

  for i = 1,N
    for each atom j that i is bonded to:
      add forces on A[i] due to A[i]

  // integrate
  for I = 1, N
    updatePositions(A[i])
}

A is an array of atoms
Each A[i] has:
  double px,py,pz ; // positions
  double vx,vy,vz; // velocities
  double fx, fy, fz; // forces
  double mass;

Aside: this ignores symmetry (Newton’s third law), which could be used to reduce the computation by half. It’s ok for now.
Parallelization Using Atom Replication

• Done in early codes
  • trying to extend their sequential code to run on parallel machines
• Each processor keeps the entire array
• Each processor calculates some subset of the interactions
  • This may involve calculating forces on arbitrary sets of atoms
  • So, need to add up all the forces and make them available to all processors (who then updates the positions for all N)
  • All-reduce with an array of size N takes time N.logP
    • This is the communication cost
Analysis of Replicated Data

• Computation: for each atom, a constant number of atoms are within its cutoff radius
  • Approx. 300 for typical simulations with 12 Angstrom cutoff
  • With relatively uniform densities
  • In any case, the number of interactions doesn’t increase with N

• Computation time per processor: k.N/p
  • (with some large k)

• Communication: N. logP

• Communication/Computation Ratio: (P log P)/k
  • ! No N in the expression. Just keeps increasing with P, with no scope for reducing it with a larger problem size
Traditional Approaches: Non-isoefficient

- Replicated data:
  - All atom coordinates stored on each processor
  - Communication/computation ratio: $P \log P$

- Partition the atoms array across processors
  - Nearby atoms may not be on the same processor
  - C/C ratio: $O(P)$

- Distribute force matrix to processors
  - Matrix is sparse, non uniform
  - C/C ratio: $\sqrt{P}$
Spatial Decomposition

- Atoms distributed to cubes based on their location
  - Relatively uniform atom density
- Size of each cube
  - Just a bit larger than cut-off radius
  - Communicate only with neighbors
  - Work: for each pair of neighbor objects
- Communication to computation ratio: $O(1)$
  - E.g. imagine 1 cube per process
- However:
  - Load imbalance
  - Limited parallelism