How to Conduct a Survey (for term project)

- Read many related papers
- Put in your understanding to classify existing works and methods
- Discuss the trade-offs and Pros and Cons of existing methods and discuss the possible improvement
- Examples
  - S. Byna, Y. Chen, X.-H. Sun, “Taxonomy of data prefetching for multicore processors”, Journal of Computer Science and Technology, vol. 24, no. 3, pp. 405-417, May, 2009 (http://www.cs.iit.edu/~scs/assets/files/4192.pdf)
  - S. Byna, Y. Chen, X.-H. Sun, “A Taxonomy of Data Prefetching MechanismsProc. of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN) Conference, May, 2008

The Three Laws: and their impact

I can improve Amdahl's law

- **Amdahl's law** (1967) shows the inherent limitation of parallel processing
- **Gustafson's law** (scalable computing, 1988) shows there is no inherent limitation for scalable parallel computing, exce engineering issues

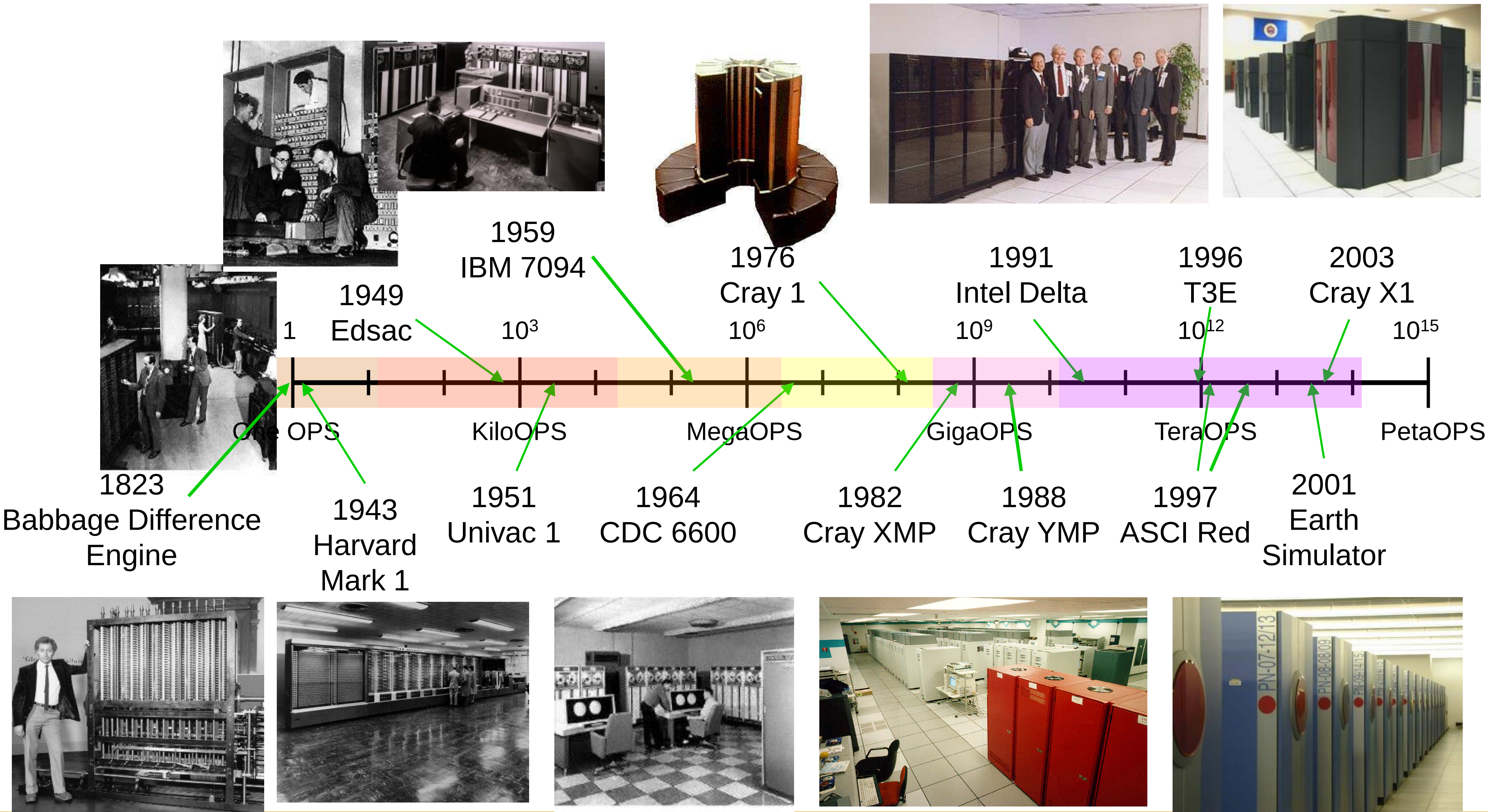
I have a huge memory

- **Sun-Ni's law** (memory-bounded, 1990) shows memory (data) is the constraint of scalable computing (**the** engineering issue)

- The **Memory-Wall Problem** (1994) shows memory-bound is a general performance issue for computing, not just for parallel computing

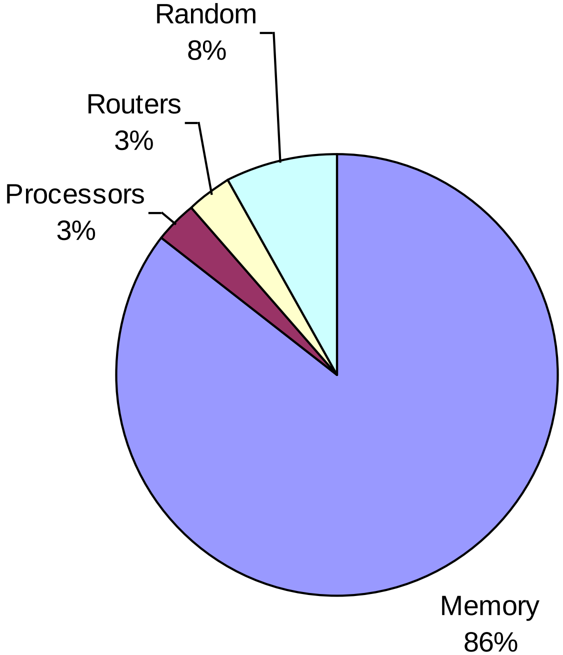
William Wulf, Sally Mckee, "Hitting the memory wall: implications of the obvious," ACM SIGARCH Computer Architecture News Homepage archive, Vol. 23 Issue 1, March 1995

Impact of Scalable Computing

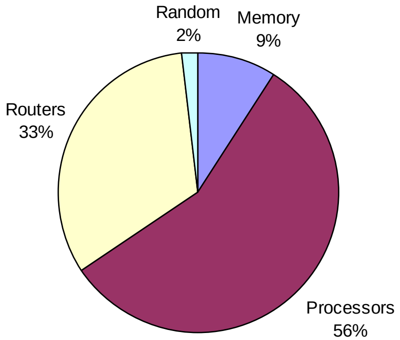


Impact: Computing/Memory Trade-off

Silicon Area Distribution



Power Distribution



Modern microprocessors such as the Pentium Pro, Alpha 21164, Strong Arm SA110, and Longson-3A use 80% or more of their transistors for the on-chip cache

Courtesy of Peter Kogge, UND

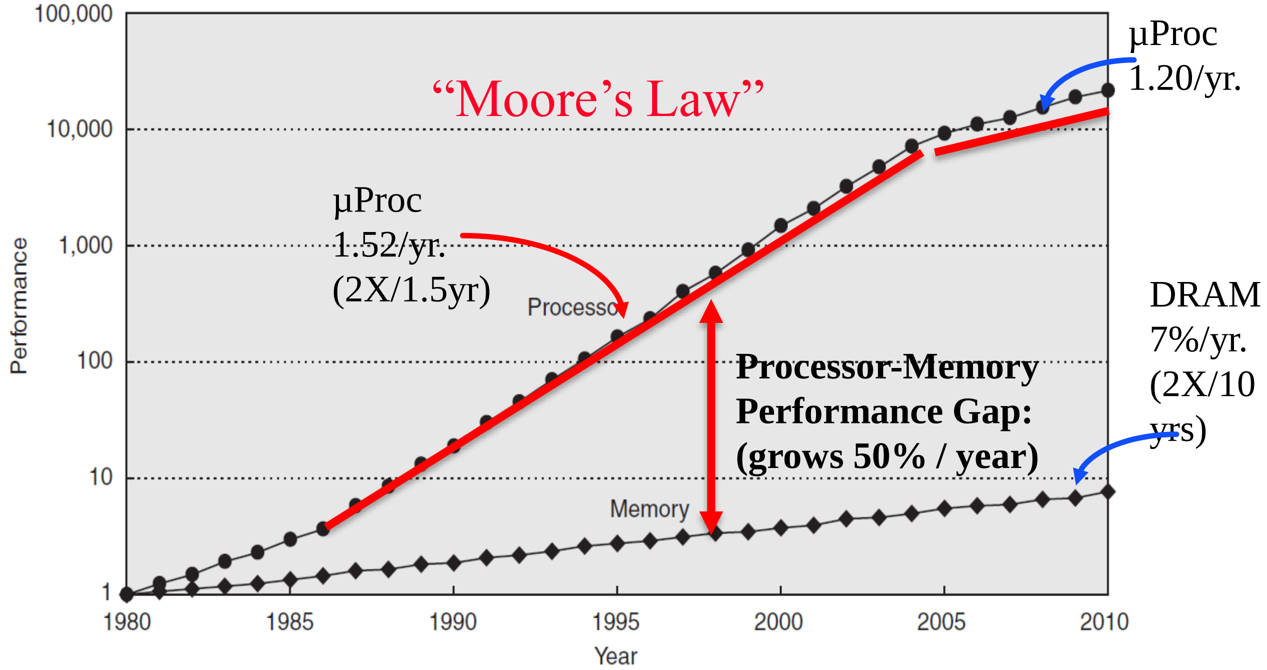
Impact of Memory-Bounded Speedup

- **W = G(M)** shows the trade-off between computing & memory
  - W, the work in floating point operation
  - M, the memory requirement
  - G, the data reuse rate
- **W = G(M)** unifies the models
  - G(p) = 1, Amdahl's law
  - G(p) = p, Gustafson's law
- Reveal memory is the performance bottleneck
  - Memory-bounded algorithms and analysis in

    Dynamic programming, distributed optimization, search, convolution, regression, etc.
  - The Memory-Wall problem (1994)

Technology Behind Memory-Bounded

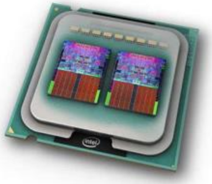
Processor-DRAM Memory Gap



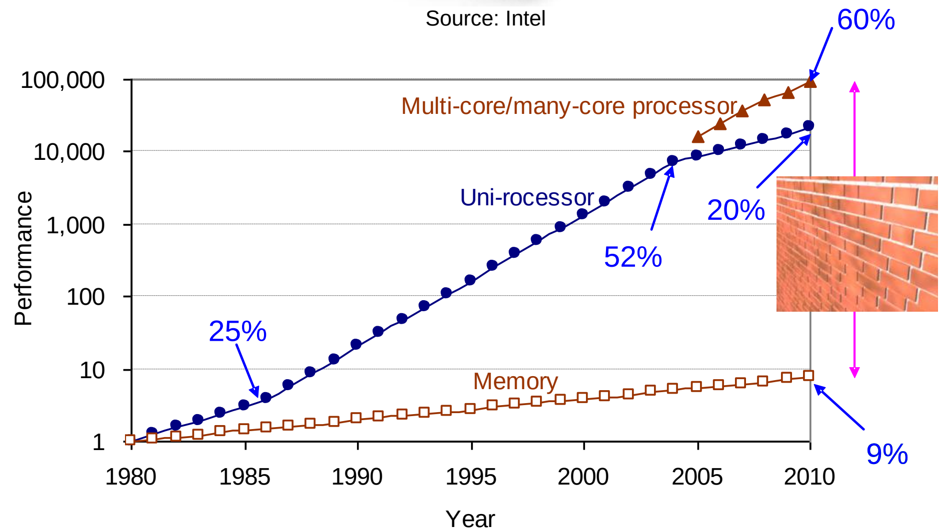
- 1980: no cache in micro-processor; 2010: 3-level cache on chip, 4-level cache off chip
- **1989 the first Intel processor with on-chip L1 cache was Intel 486, 8KB size**
- 1995 the first Intel processor with on-chip L2 cache was Intel Pentium Pro, 256KB size
- 2003 the first Intel processor with on-chip L3 cache was Intel Itanium 2, 6MB size

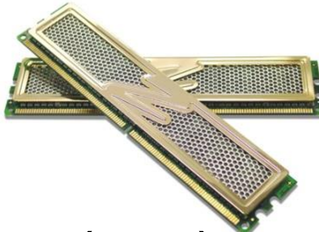
Impact: The Memory-wall Problem

- Processor performance increases rapidly
  - Uni-processor: ~52% until 2004
  - Aggregate multi-core/many-core processor performance even higher since 2004
- Memory: ~9% per year
  - Storage: ~6% per year
- Processor-memory speed gap keeps increasing



Source: Intel





Source: OCZ

Memory-bounded speedup (1990), Memory wall problem (1994)

Impact: Scalability of Multicore

- Based on Amdahl's law Multicore is not scalable

$$\frac{w_c}{perf(r)} + \frac{w_p}{perf(r)} = \frac{w_c}{perf(r)} + \frac{w_p'}{m \cdot perf(r)} \quad => \quad w_p' = m w_p$$

- Based on Gustafson and Sun-Ni's law, it scalable

$$\frac{\dfrac{w_c}{perf(r)} + \dfrac{w_p'}{m \cdot perf(r)}}{\dfrac{w_c}{perf(r)} + \dfrac{w_p}{perf(r)}} = \frac{w_c + m \cdot w_p}{w_c + w_p} = (1 - f') + mf'$$

$$f' = \frac{w_p}{w_c + w_p}$$

- Based on Sun-Ni's law
  - Multicore is scalable, if data access time is fixed and does not increase with the amount of work and the number of cores
  - **Implication:** Data access is the bottleneck needs attention

X.-H. Sun and Y. Chen, "Reevaluating Amdahl's Law in the Multicore Era," Journal of Parallel and Distributed Computing, vol. 70, no. 2, pp. 183-188, Feb. 2010.

Impact: Direct Apply on Many-core Design

Yu-Hang Liu and Xian-He Sun, *"C^2-bound: A Capacity and Concurrency driven Analytical Model for Manycore Design,"* in Proc. of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis 2015 (SC'15). Texas, Austin, USA, Nov. 2015.

Yu-Hang Liu, Xian-He Sun, *"Evaluating the Combined Effect of Memory Capacity and Concurrency for Many-core Chip Design,"* ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS), vol. 2, no. 2, pp. 9:1-9:25, Apr. 2017.

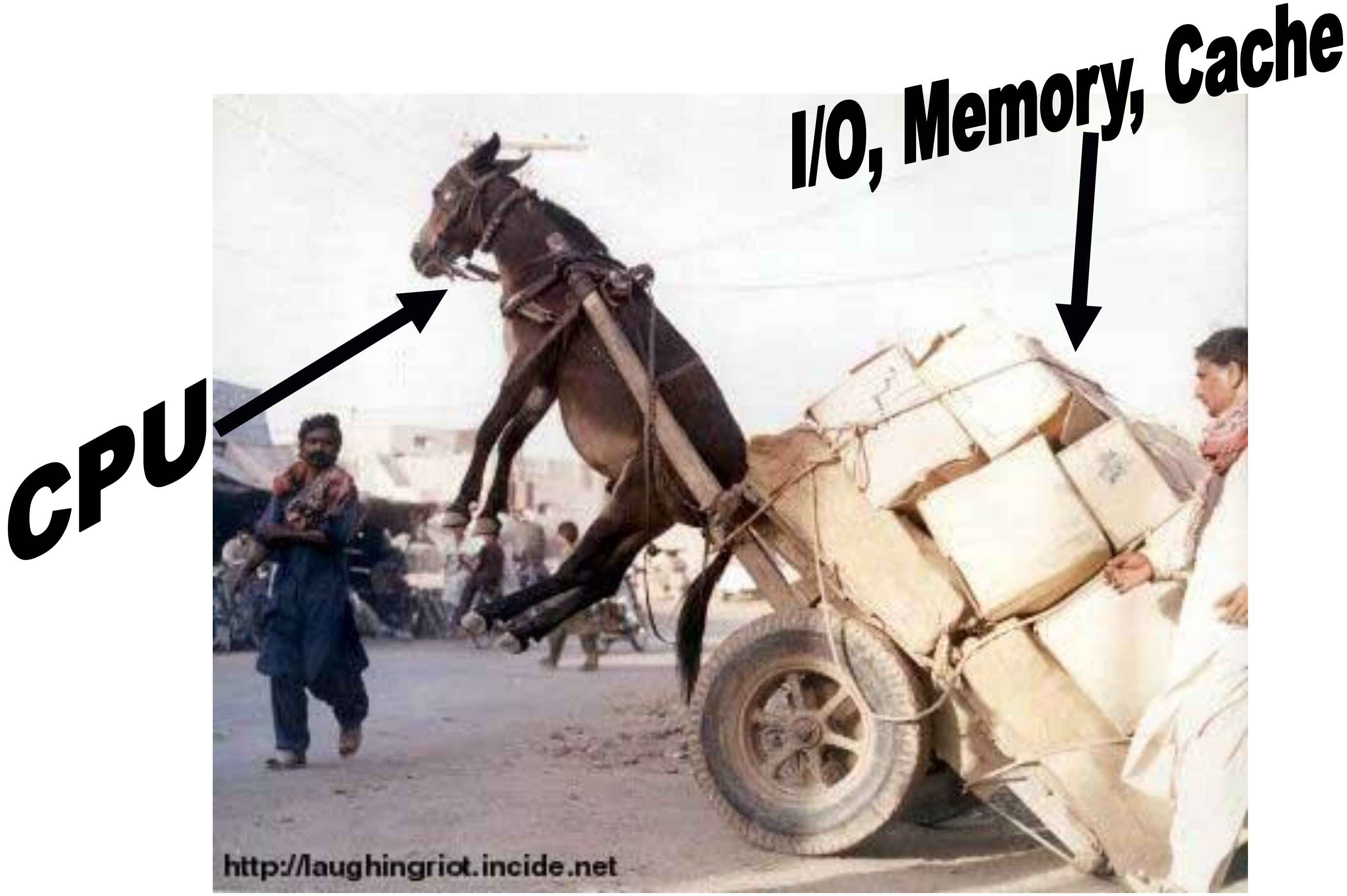
The Beauty of Mathematics

- The ability of abstract
- In depth understanding
  of the engineering issues
- Creative thinking

- Complex Specificity, Simple Genericity
- Abstract the complex specificity into simple genericity
- Engineering, mathematics, philosophy
- Everybody understand something, at a different level
- Your understanding determine your ability to apply it

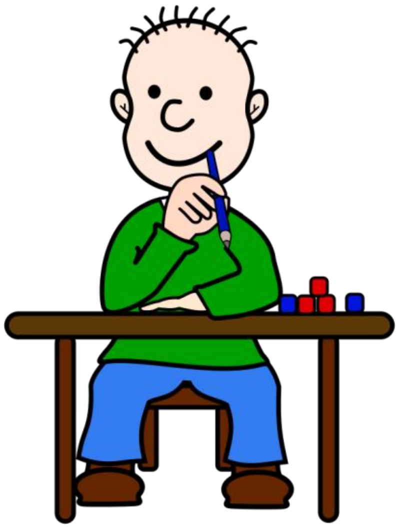
- 厚积薄发，可遇不可求

Big Data Makes Memory-Bound Even Worse



- Source: Bob Colwell keynote ISCA'29 2002 http://systems.cs.colorado.edu/ISCA2002/Colwell-ISCA-KEYNOTE-2002-final.ppt

How do we solve the memory-bound constraint or the memory-wall problem



SEE YOU NEXT TIME
且听下回分解

Performance Evaluation of Parallel Processing (2)

Xian-He Sun
Illinois Institute of Technology
Sun@iit.edu

Outline

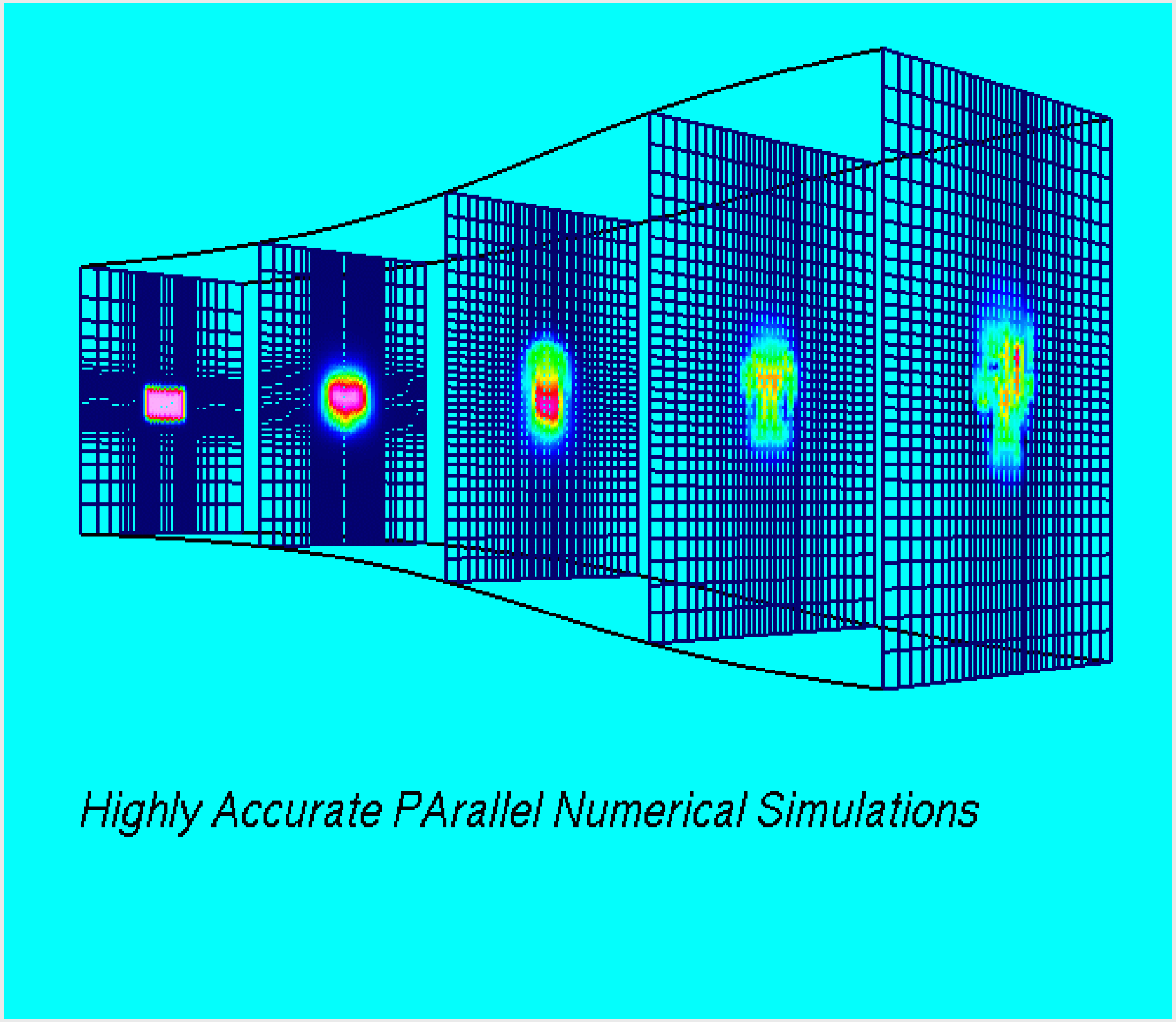
- Performance metrics
  - Speedup
  - Efficiency
  - Scalability
- Examples
- Summary
- Reading: Kumar – ch 5

Why Scalable Computing

– Scalable
  - More accurate solution
  - Sufficient parallelism
  - Maintain efficiency

– Efficient in parallel computing
  - Load balance
  - Communication

– Mathematically effective
  - Adaptive
  - Accuracy



Why Scalable Computing (2)

Small Work

- Appropriate for small machine
  - Parallelism overheads begin to dominate benefits for larger machines
    - Load imbalance
    - Communication to computation ratio
  - May even achieve slowdowns
  - Does not reflect real usage, and inappropriate for large machine
    - Can exaggerate benefits of improvements

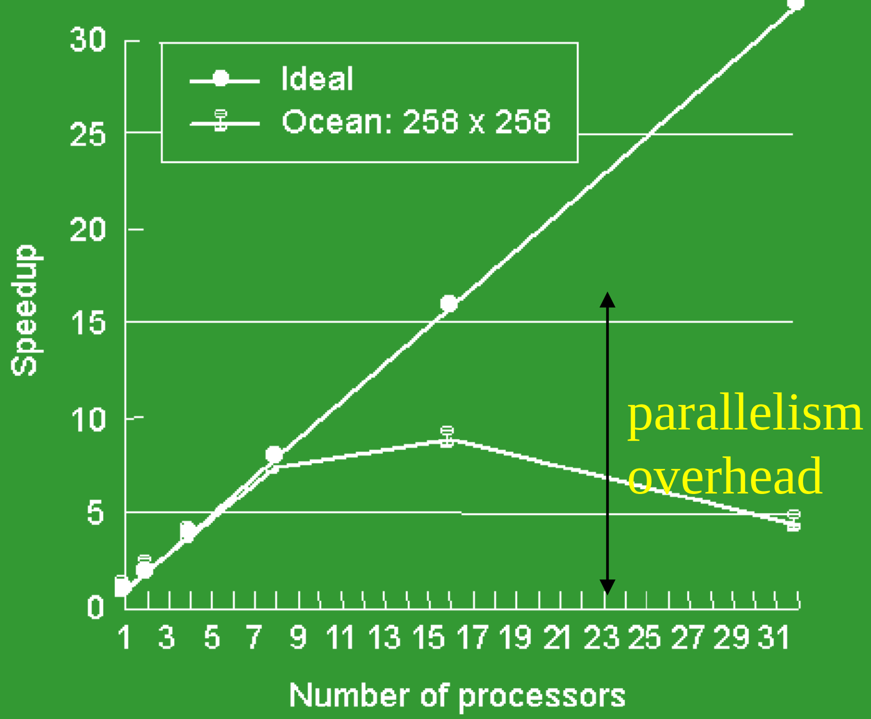
Why Scalable Computing (3)

Large Work

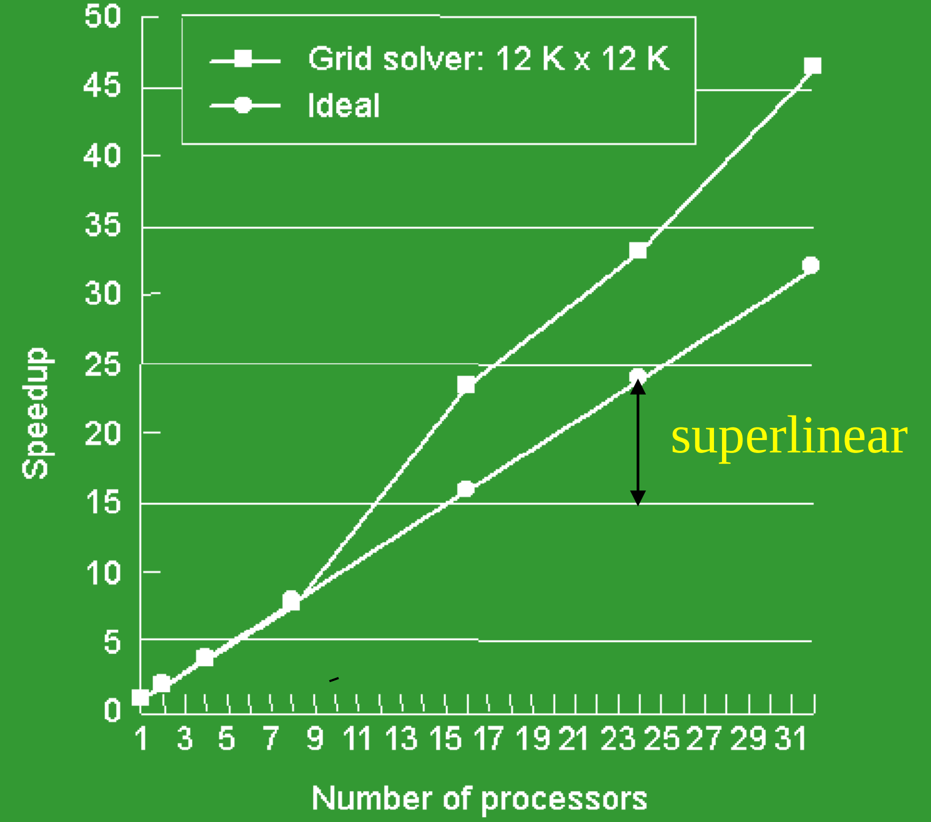
- Appropriate for big machine
  - Difficult to measure improvement
  - May not fit for small machine
    - Can’t run
    - Thrashing to disk
    - Working set doesn’t fit in cache
  - Fits at some *p*, leading to superlinear speedup

Demonstrating Scaling Problems

Small Ocean problem
On SGI Origin2000



Big equation solver problem
On SGI Origin2000



User want to scale problems as machines grow!

How to Scale

- Scaling a machine
  - Make a machine more powerful （*scale up*）
  - Machine size
    - <processor, memory, communication, I/O>
  - Scaling a machine in parallel processing （***scale out***)
    - Add more identical nodes
- Problem size
  - Input configuration
  - data set size : the amount of storage required to run it on a single processor
  - memory usage : the amount of memory used by the program

How to Scale

- We are interested in Scale Out
  - Scaling a machine in parallel processing （***scale out***)
    - Add more identical nodes
  - Scaling a virtual machine in a Cloud environment
  - Scaling the number of cells in a many-core environment
- Problem size
  - Input configuration
  - data set size : the amount of storage required to run it on a single processor
  - memory usage : the amount of memory used by the program

Two Key Issues in Problem Scaling

- Under what constraints should the problem be scaled?
  - Some properties must be fixed as the machine scales
- How should the problem be scaled?
  - Which parameters?
  - How?

Constraints To Scale

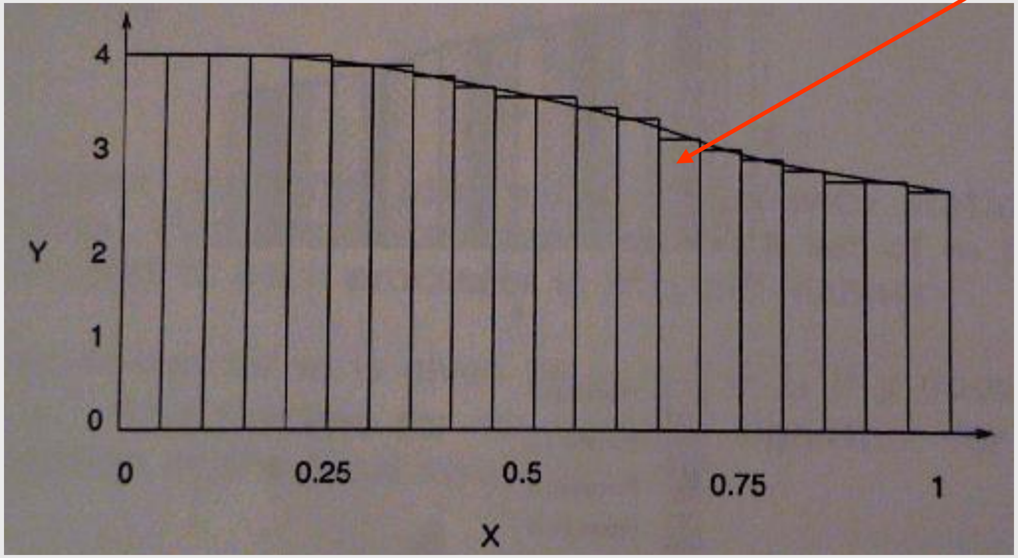
- Two types of constraints
  - Problem-oriented
    - Ex) Time
  - Resource-oriented
    - Ex) Memory
- Work to scale
  - Metric-oriented
    - Floating point operation, instructions
  - User-oriented
    - Easy to change but may difficult to compare
    - Ex) particles, rows, transactions
    - Difficult cross comparison

Examples: Compute π: Problem

- Consider parallel algorithm for computing the value of π=3.1415…through the following numerical integration

$$\pi = \int_0^1 \frac{4}{1+x^2}dx$$

$$\boxed{\frac{4}{1+x^2}}$$

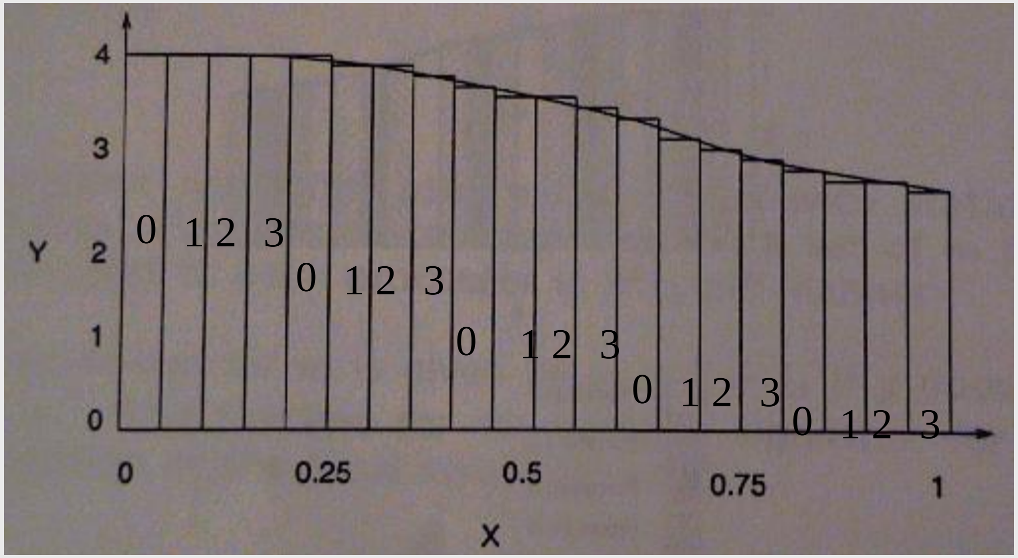


Compute π: Sequential Algorithm

```
computepi()
{
    h=1.0/n;
    sum =0.0;
    for (i=0;i<n;i++) {
        x=h*(i+0.5);
        sum=sum+4.0/(1+x*x);
    }
    pi=h*sum;
}
```

Compute π: Parallel Algorithm

- Each processor computes on a set of about n/p points which are allocated to each processor in a cyclic manner
- Finally, we assume that the local values of π are accumulated among the p processors under synchronization



Compute π: Parallel Algorithm

```
computepi()
{
    id=my_proc_id();
    nprocs=number_of_procs():
    h=1.0/n;
    sum=0.0;
    for(i=id;i<n;i=i+nprocs) {
        x=h*(i+0.5);
        sum=sum+4.0/(1+x*x);
    }
    localpi=sum*h;
    use_tree_based_combining_for_critical_section();
        pi=pi+localpi;
    end_critical_section();
}
```

Compute π: Analysis

- Assume that the computation of π is performed over n points
- The sequential algorithm performs 6 operations (two multiplications, one division, three additions) per points on the x-axis. Hence, for n points, the number of operations executed in the sequential algorithm is:

$$T_s = 6n$$

3 additions

1 division

```
for (i=0;i<n;i++) {
        x=h*(i+0.5);
        sum=sum+4.0/(1+x*x);
}
```

2 multiplications

Compute π: Analysis

- The parallel algorithm uses p processors with static interleaved scheduling. Each processor computes on a set of m points which are allocated to each process in a cyclic manner
- The expression for m is given by $m \le \frac{n}{p} + 1$ if p does not exactly divide n. The runtime for the parallel algorithm for the parallel computation of the local values of π is:

$$T_p = 6m * t_0 = (6\frac{n}{p} + 6)t_0$$

Compute π: Analysis

- The accumulation of the local values of π using a tree-based combining can be optimally performed in $\log_2(p)$ steps
- The total runtime for the parallel algorithm for the computation of π including the parallel computation and the combining is:

$$T_p = 6m * t_0 = (6\frac{n}{p} + 6)t_0 + \log(p)(t_0 + t_c)$$

- The speedup of the parallel algorithm is:

$$S_p = \frac{T_s}{T_p} = \frac{6n}{6\frac{n}{p} + 6 + \log(p)(1 + t_c / t_0)}$$

Compute π: Analysis

- The Amdahl's fraction for this parallel algorithm can be determined by rewriting the previous equation as:

$$S_p = \frac{p}{1 + \frac{p}{n} + \frac{pc\log(p)}{6n}} \Rightarrow S_p = \frac{p}{1 + (p-1)\alpha(n,p)}$$

- Hence, the Amdahl's fraction α(n,p) is:

$$\alpha(n,p) = \frac{p}{(p-1)n} + \frac{pc\log(p)}{6n(p-1)}$$

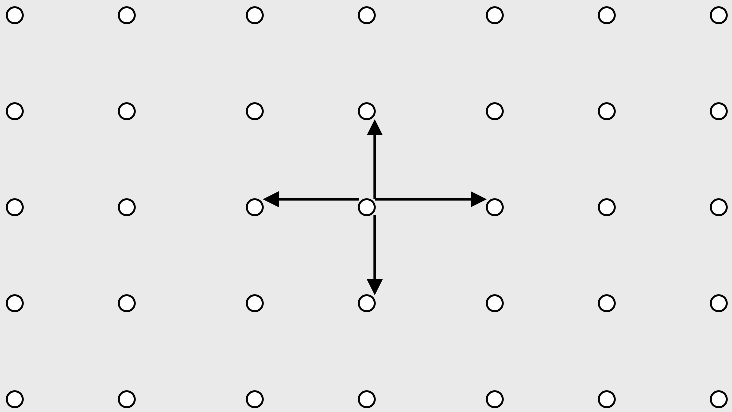
- The parallel algorithm is effective because:

$$\alpha(n,p) \rightarrow 0 \text{ as } n \rightarrow \infty \text{ for fixed } p$$

Finite Differences: Problem

- Consider a finite difference iterative method applied to a 2D grid where:

$$X_{i,j}^{t+1} = \omega \cdot (X_{i,j-1}^{t} + X_{i,j+1}^{t} + X_{i-1,j}^{t} + X_{i+1,j}^{t}) + (1-\omega) \cdot X_{i,j}^{t}$$

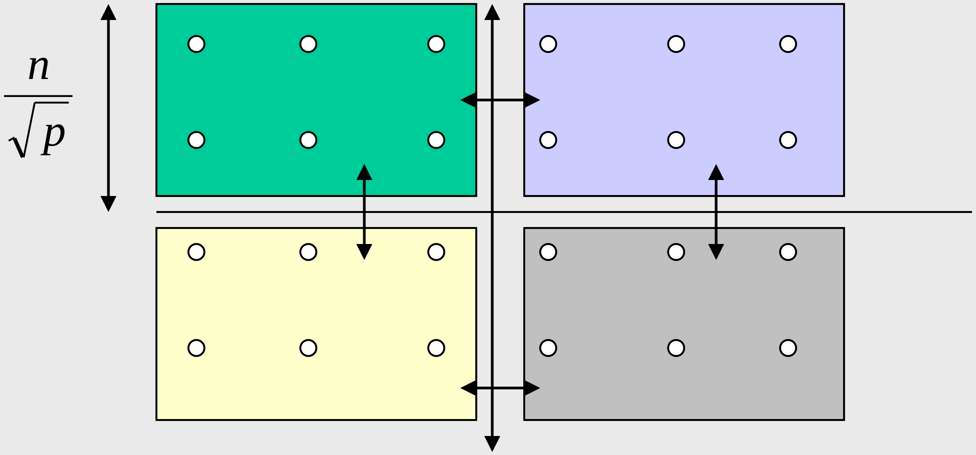


Finite Differences: Serial Algorithm

```
finitediff()
{
    for (t=0;t<T;t++) {
        for (i=0;i<n;i++) {
            for (j=0;j<n;j++) {
                x[i,j]=w_1*(x[i,j-1]+x[i,j+1]+x[i-1,j]+x[i+1,j]+w_2*x[i,j];
            }
        }
    }
}
```

Finite Differences: Parallel Algorithm

- Each processor computes on a sub-grid of $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ points
- Synch between processors after every iteration ensures correct values being used for subsequent iterations



Finite Differences: Parallel Algorithm

```
finitediff()
{
    row_id=my_processor_row_id();
    col_id=my_processor_col_id();
    p=numbre_of_processors();
    sp=sqrt(p);
    rows=cols=ceil(n/sp);
    row_start=row_id*rows;
    col_start=col_id*cols;
    for (t=0;t<T;t++) {
        for (i=row_start;i<min(row_start+rows,n);i++) {
            for (j=col_start;j<min(col_start+cols,n);j++) {
                x[i,j]=w_1*(x[i,j-1]+x[i,j+1]+x[i-1,j]+x[i+1,j]+w_2*x[i,j];
            }
            barrier();
        }
    }
}
```

Finite Differences:Analysis

- The sequential algorithm performs 6 operations(2 multiplications, 4 additions) every iteration per point on the grid. Hence, for an n*n grid and T iterations, the number of operations executed in the sequential algorithm is:

$$T_s = 6n^2 t_0$$

2 multiplications

x[i,j]=w_1(x[i,j-1]+x[i,j+1]+x[i-1,j]+x[i+1,j]+w_2*x[i,j];*

4 additions

Finite Differences:Analysis

- The parallel algorithm uses p processors with static blockwise scheduling. Each processor computes on an m*m sub-grid allocated to each processor in a blockwise manner
- The expression for m is given by $m \le \left\lceil \frac{n}{\sqrt{p}} \right\rceil$ The runtime for the parallel algorithm is:

$$T_p = 6m^2 t_0 = 6\left(\left\lceil \frac{n}{\sqrt{p}} \right\rceil\right)^2 t_0$$

Finite Differences:Analysis

- The barrier synch needed for each iteration can be optimally performed in log(p) steps
- The total runtime for the parallel algorithm for the computation is:

$$T_p = 6m^2 t_0 = 6(\left\lceil \frac{n}{\sqrt{p}} \right\rceil)^2 t_0 + \log(p)(t_0 + t_c) = 6\frac{n^2}{p} t_0 + \log(p)(t_0 + t_c)$$

- The speedup of the parallel algorithm is:

$$S_p = \frac{T_s}{T_p} = \frac{6n^2}{6\frac{n^2}{p} + \log(p)(1 + t_c / t_0)}$$

Finite Differences:Analysis

- The Amdahl's fraction for this parallel algorithm can be determined by rewriting the previous equation as:

$$S_p = \frac{p}{1 + \frac{pc\log(p)}{6n^2}} \Rightarrow S_p = \frac{p}{1 + (p-1)\alpha(n,p)}$$

- Hence, the Amdahl's fraction α(n.p) is:

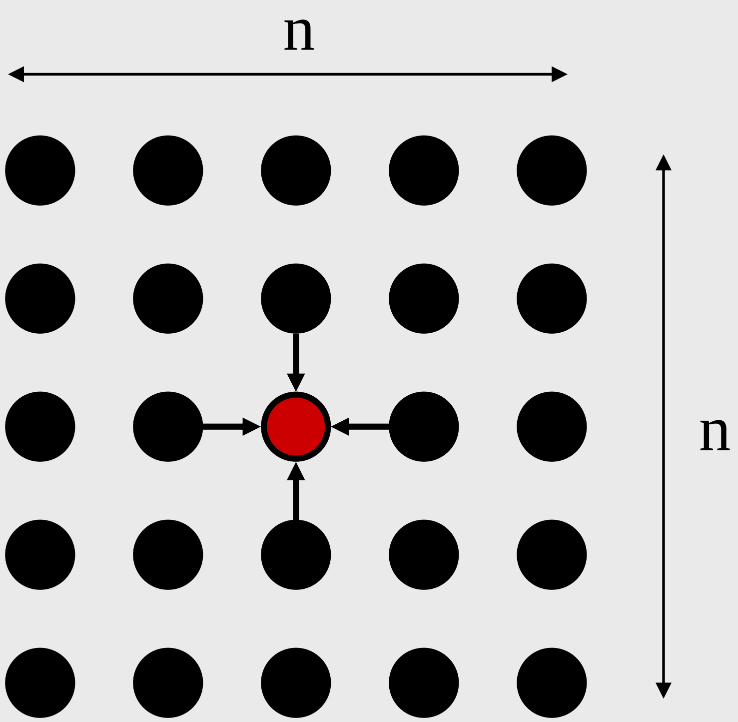
$$\alpha(n,p) = \frac{pc\log(p)}{(p-1)6n^2}$$

- We finally note that

$$\alpha(n,p) \to 0 \text{ as } n \to \infty \text{for fixed p}$$

- Hence, the parallel algorithm is effective

Equation Solver




```
procedure solve (A)
    …
    while(!done) do
        diff = 0;
        for i = 1 to n do
            for j = 1 to n do
                temp = A[i, j];
                A[i, j] = …
                diff += abs(A[i,j] – temp);
            end for
        end for
        if (diff/(n*n) < TOL) then done =1 ;
    end while
end procedure
```

A[i,j] = 0.2 * (A[i, j] + A[i, j-1] + A[i-1, j] + a[i, j+1] + a[i+1, j])

Workloads

- Basic properties
  - Memory requirement : $O(n^2)$
  - Computational complexity : $O(n^3)$, assuming the number of iterations to converge to be $O(n)$
- Assume speedups equal to # of p
- Grid size
  - Fixed-size : fixed
  - Fixed-time :

$$n^3 \times p = k^3 \;\therefore k = \sqrt[3]{p} \times n$$

  - Memory-bound :

$$n^2 \times p = k^2 \;\therefore k = \sqrt{p} \times n$$

Memory Requirement of Equation Solver

Fixed-size : $\frac{n^2}{p}$ $\downarrow$

Fixed-time: $\frac{k^2}{p} = \frac{(n \times \sqrt[3]{p})^2}{p} = \frac{n^2}{\sqrt[3]{p}}$, $n^3 \times p = k^3$ $\downarrow$

Memory-bound : $n^2 \times p$ $\uparrow$

Time Complexity of Equation Solver

Fixed-size: $\dfrac{n^3}{p}$ ↓

Fixed-time: n^3

Memory-bound: $\dfrac{(n\sqrt{p})^3}{p} = n^3\sqrt{p}$ ↑

Sequential time complexity

$k^3 = (n \times \sqrt{p})^3$, $n^2 \times p = k^2$

Concurrency

Concurrency is proportional to the number of grid points

Fixed-size : n^2

Fixed-time: $k^2 = (n \times \sqrt[3]{p})^2 = n^2 \times \sqrt[3]{p^2}$ ↑, $n^3 \times p = k^3$

Memory-bound: $n^2 \times p = k^2$ ↑

Communication to Computation Ratio

Fixed-size :

$$CCR = \frac{\sqrt{\frac{n^2}{p}}}{\frac{n^2}{p}} = \frac{1}{\sqrt{\frac{n^2}{p}}} = \frac{\sqrt{p}}{n}$$

Memory-bound :

$$CCR = \frac{\sqrt{\frac{k^2}{p}}}{\frac{k^2}{p}} = \frac{1}{\sqrt{\frac{k^2}{p}}} = \frac{1}{\sqrt{\frac{(n\sqrt{p})^2}{p}}} = \frac{1}{n}$$

Fixed-time :

$$CCR = \frac{\sqrt{\frac{k^2}{p}}}{\frac{k^2}{p}} = \frac{1}{\sqrt{\frac{k^2}{p}}} = \frac{1}{\sqrt{\frac{(n\sqrt[3]{p})^2}{p}}} = \frac{\sqrt[6]{p}}{n}$$

Scalability

- The Need for New Metrics
  - Comparison of performances with different workload
  - Availability of massively parallel processing
- Scalability

  Ability to maintain parallel processing gain when both problem size and system size increase

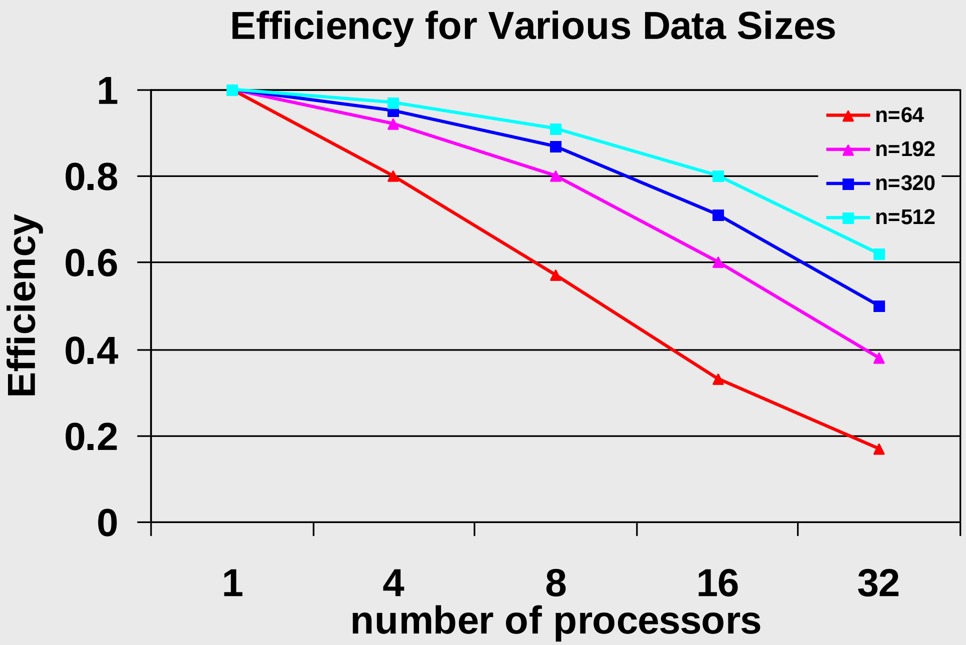
Parallel Efficiency

$$E_p = \frac{S_p}{p}$$

- The achieved fraction of total potential parallel processing gain
  - Assuming linear speedup *p* is ideal case
- The ability to maintain efficiency when problem size increase
- Isoefficiency

Maintain Efficiency (isoefficiency)

- Efficiency of adding n numbers in parallel



E=1/(1+2plogp/n)

  - For an efficiency of 0.80 on 4 procs, n=64
  - For an efficiency of 0.80 on 8 procs, n=192
  - For an efficiency of 0.80 on 16 procs, n=512

- Ideally Scalable

$$T(m \times p, m \times W) = T(p, W)$$

  - T: execution time
  - W: work executed
  - P: number of processors used
  - m: scale up m times
  - work: flop count based on the best practical serial algorithm
- Fact:

$$T(m \times p, m \times W) = T(p, W)$$

if and only if

The Average Unit Speed Is Fixed

– Definition:

The *average unit speed* is the achieved speed divided by the number of processors

– Definition (Isospeed Scalability):

An algorithm-machine combination is scalable if the achieved average unit speed can remain constant with increasing numbers of processors, provided the problem size is increased proportionally

- Isospeed Scalability (Sun & Rover, 91)
  - W: work executed when p processors are employed
  - W': work executed when p' > p processors are employed to maintain the average speed

$$Scalability = \psi(p, p') = \frac{p' \cdot W}{p \cdot W'}$$

  - Ideal case

$$W' = \frac{p' \cdot W}{p}, \qquad \psi(p, p') = 1$$

  - Scalability in terms of time

$$\psi(p, p') = \frac{T_p(W)}{T_{p'}(W')} = \frac{\text{time with work } W \text{ on } p \text{ processors}}{\text{time with work } W' \text{ on } p' \text{ processors}}$$

- Isospeed Scalability (Sun & Rover)
  - W: work executed when p processors are employed
  - W': work executed when p' > p processors are employed to maintain the average speed

$$Scalability = \psi(p, p') = \frac{p' \cdot W}{p \cdot W'}$$

  - Ideal case

$$W' = \frac{p' \cdot W}{p}, \qquad \psi(p, p') = 1$$

- X. H. Sun, and D. Rover, "Scalability of Parallel Algorithm-Machine Combinations," *IEEE Trans. on Parallel and Distributed Systems*, May, 1994 (Ames TR91)

The Relation of Scalability and Time

- More scalable leads to smaller time
  - Better initial run-time and higher scalability lead to superior run-time
  - Same initial run-time and same scalability lead to same scaled performance
  - Superior initial performance may not last long if scalability is low
- Range Comparison

• X.H. Sun, "Scalability Versus Execution Time in Scalable Systems," *Journal of Parallel and Distributed Computing*, Vol. 62, No. 2, pp. 173-192, Feb 2002.

Range Comparison Via Performance Crossing Point

Assume Program I is oz times slower than program 2 at the initial state

Begin (Range Comparison)

$p' = p;$

Repeat

$p' = p' + 1;$

Compute the scalability of program 1 $\Phi(p,p')$;

Compute the scalability of program 2 $\Psi(p,p')$;

Until ($\Phi(p,p') > \alpha \Psi(p,p')$ or p'= the limit of ensemble size)

If $\Phi(p,p') > \alpha \Psi(p,p')$ **Then**

p is the smallest scaled crossing point;

program 2 is superior at any ensemble size $p^{\dagger}$, $p \leq p^{\dagger} < p'$

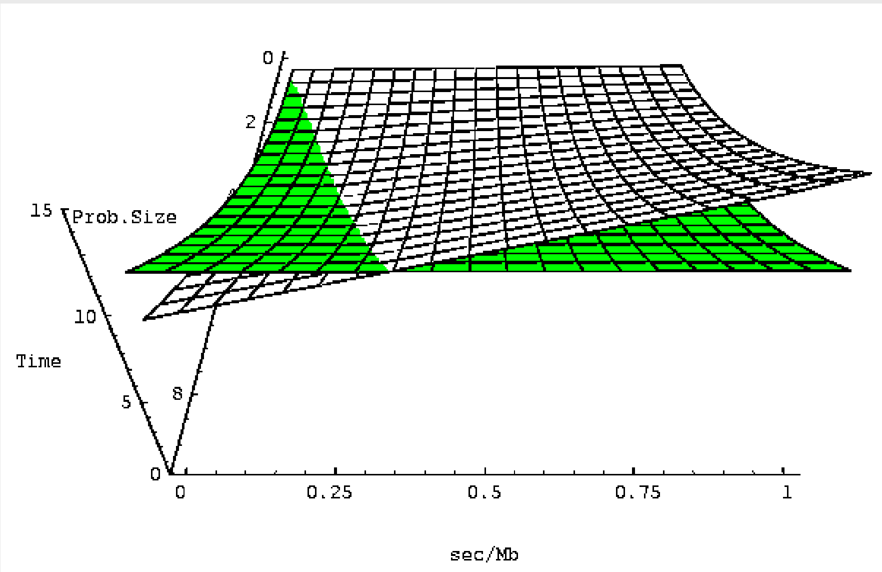
Else

program 2 is superior at any ensemble size $p^{\dagger}$, $p \leq p^{\dagger} \leq p'$

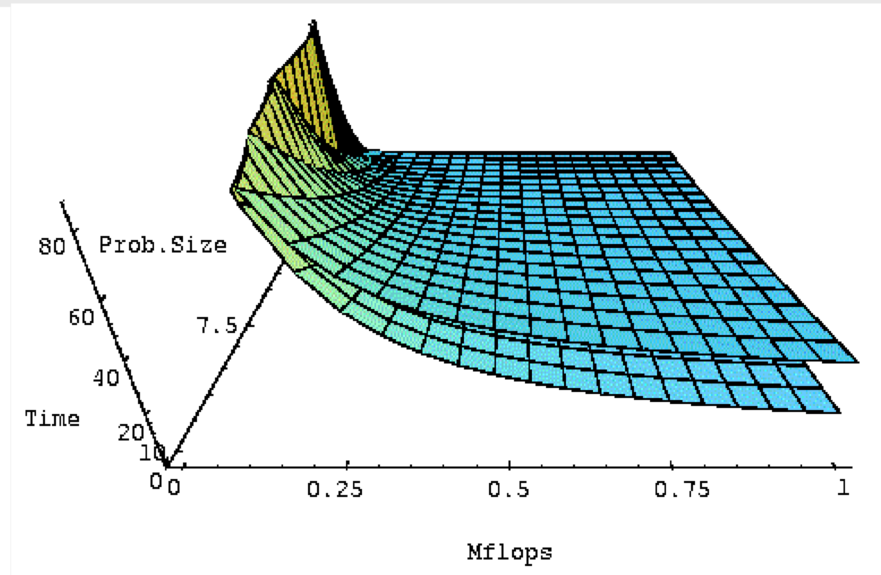
End {if}

End {Range Comparison}

- Range Comparison



Influence of Communication Speed



Influence of Computing Speed

- X.H. Sun, M. Pantano, and Thomas Fahringer, "Integrated Range Comparison for Data-Parallel Compilation Systems," *IEEE Trans. on Parallel and Distributed Processing*, May 1999.

Summary

- Relation between Iso-speed scalability and iso-efficiency scalability
  – Both measure the ability to maintain parallel efficiency defined as

$$E_p = \frac{S_p}{p}$$

  – Where iso-efficiency's speedup is the traditional speedup defined as

$$S_p = \frac{\text{Uniprocessor Execution Time}}{\text{Parallel Execution Time}}$$

  – Iso-speed's speedup is the generalized speedup defined as

$$S_p = \frac{\text{Parallel Speed}}{\text{Sequential Speed}}$$

  – If the the sequential execution speed is independent of problem size, iso-speed and iso-efficiency is equivalent
  – Due to memory hierarchy, sequential execution performance varies largely with problem size

Summary

- Predict the sequential execution performance becomes a major task due to advanced memory hierarchy
  - Memory-LogP model is introduced for data access cost
- New challenge in distributed computing
- Generalized iso-speed scalability
- Generalized performance tool: GHS

- K. Cameron and X.-H. Sun, "Quantifying Locality Effect in Data Access Delay: Memory logP," *Proc. of 2003 IEEE IPDPS 2003*, Nice, France, April, 2003.
- X.-H. Sun and M. Wu, "Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling," *Proc. of 2003 IEEE IPDPS 2003*, Nice, France, April, 2003.