WORKSHOP SERIES

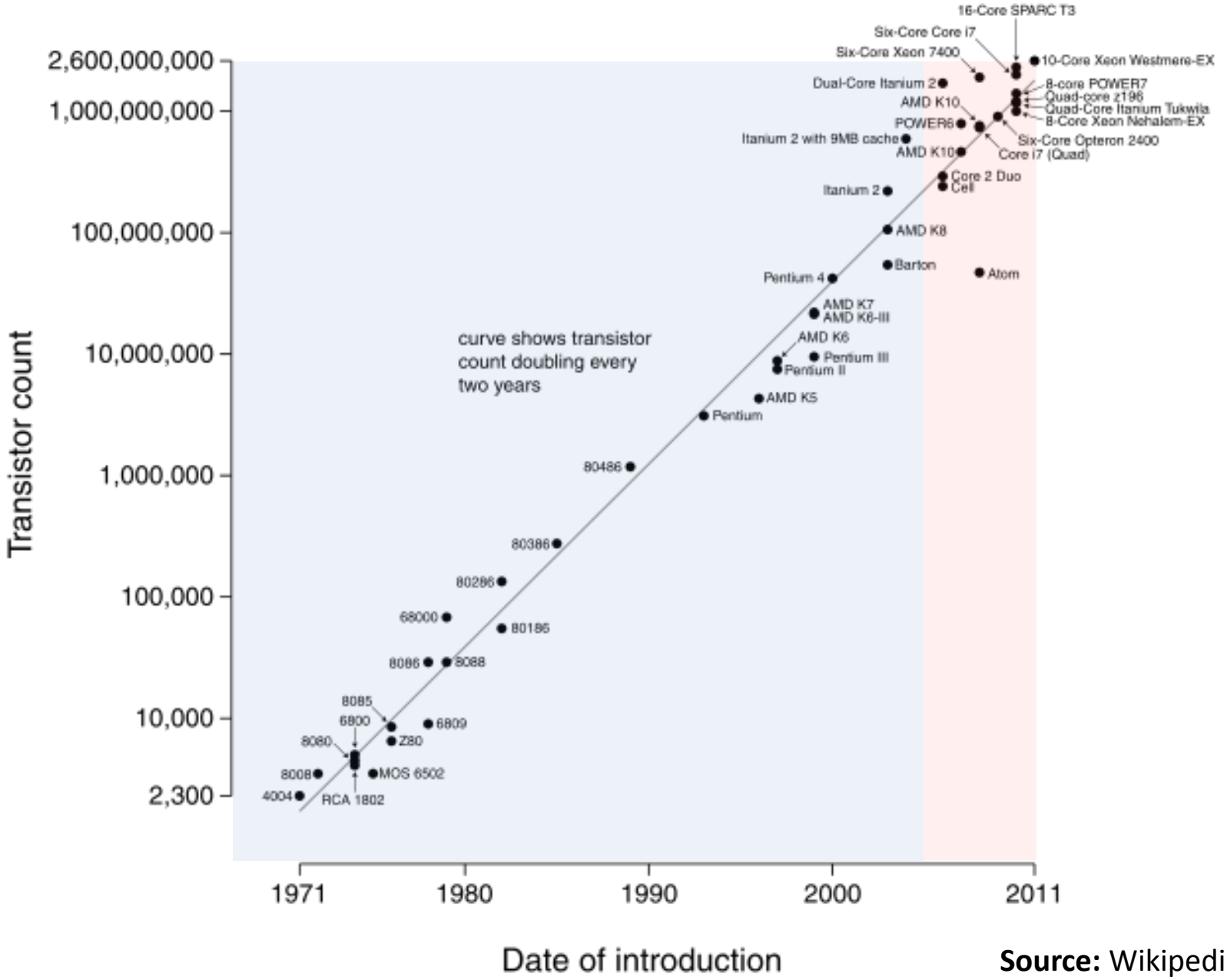
OPENMP/MPI TUTORIAL

GENERAL INTRODUCTION TO PARALLEL COMPUTING

Rezaul Chowdhury
Department of Computer Science
Stony Brook University

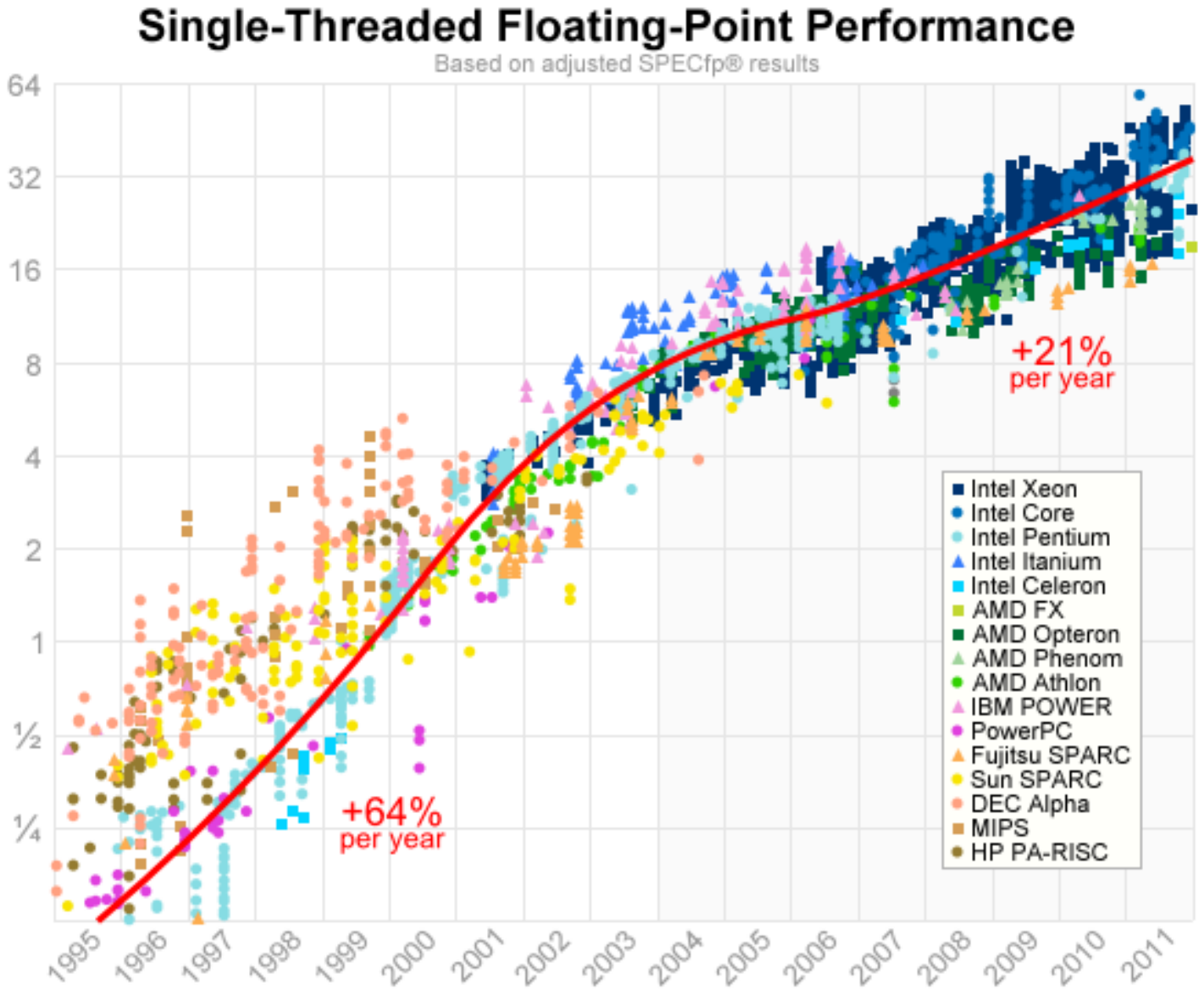
Why Parallelism?

Moore's Law



Source: Wikipedia

Unicore Performance



Source: Jeff Preshing, 2012, http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/

Unicore Performance Has Hit a Wall!

Some Reasons

- Lack of additional ILP
  (Instruction Level Hidden Parallelism)
- High power density
- Manufacturing issues
- Physical limits
- Memory speed

Unicore Performance: No Additional ILP

"Everything that can be invented has been invented."

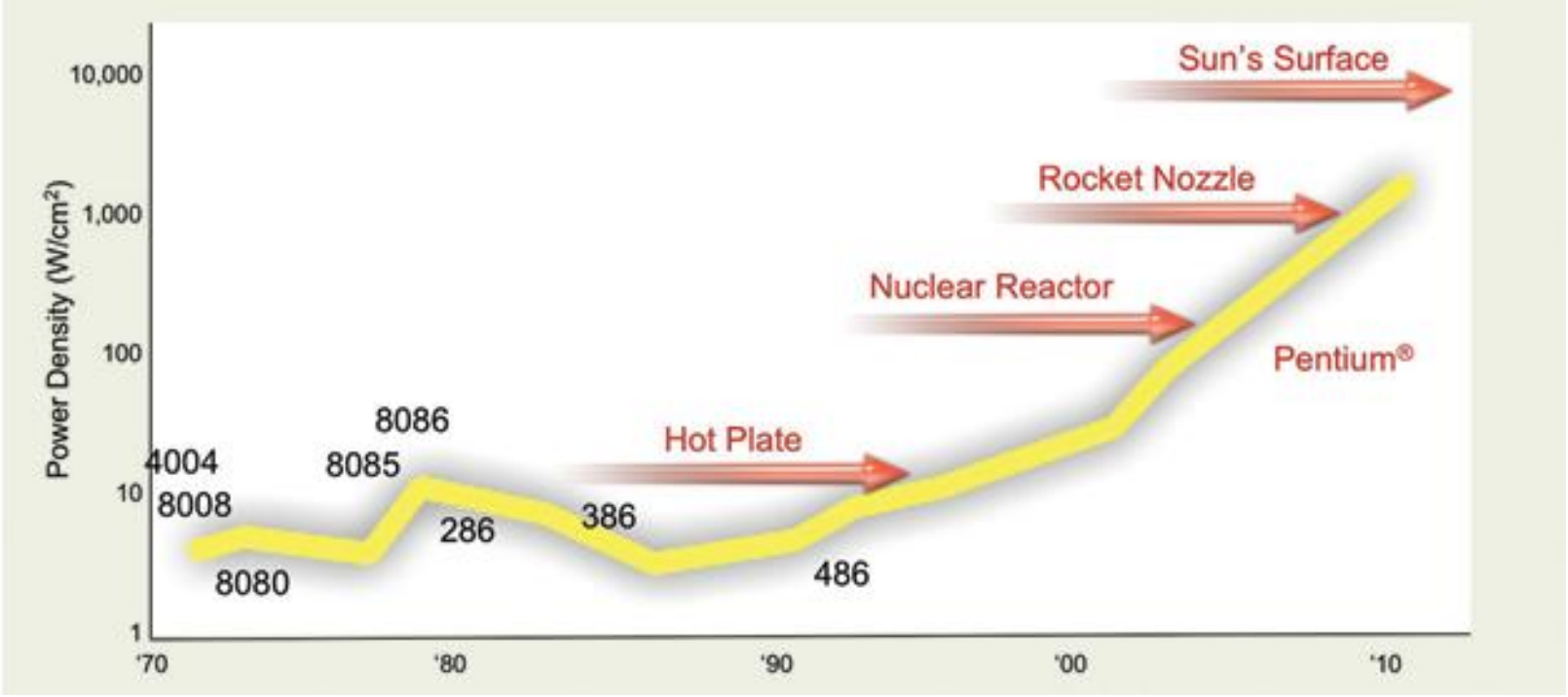
— Charles H. Duell
Commissioner, U.S. patent office, 1899

Exhausted all ideas to exploit hidden parallelism?

- Multiple simultaneous instructions
- Instruction Pipelining
- Out-of-order instructions
- Speculative execution
- Branch prediction
- Register renaming, etc.

Unicore Performance: High Power Density

- Dynamic power, $P_d \propto V^2 f C$
  - *V = supply voltage*
  - *f = clock frequency*
  - *C = capacitance*
- But $V \propto f$
- Thus $P_d \propto f^3$



Source: Patrick Gelsinger, Intel Developer Forum, Spring 2004 (Simon Floyd)

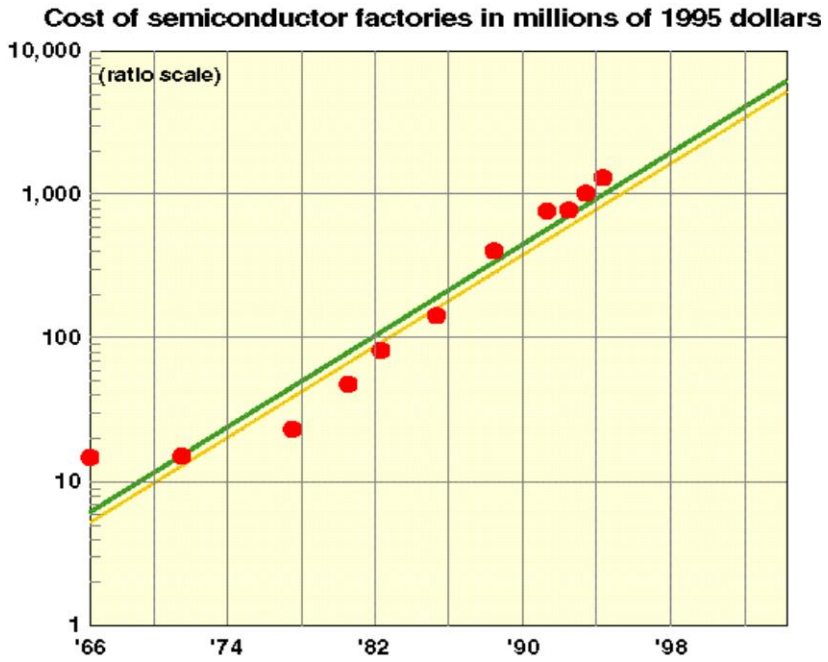
Unicore Performance: Manufacturing Issues

- Frequency, $f \propto 1 / s$
  - *s = feature size (transistor dimension)*
- Transistors / unit area $\propto 1 / s^2$
- Typically, die size $\propto 1 / s$
- So, what happens if feature size goes down by a factor of *x*?
  - Raw computing power goes up by a factor of x^4 !
  - Typically most programs run faster by a factor of x^3 without any change!

Source: Kathy Yelick and Jim Demmel, UC Berkeley

Unicore Performance: Manufacturing Issues

- Manufacturing cost goes up as feature size decreases
  - Cost of a semiconductor fabrication plant doubles every 4 years (Rock's Law)
- CMOS feature size is limited to 5 nm (at least 10 atoms)



Source: Kathy Yelick and Jim Demmel, UC Berkeley

Unicore Performance: Physical Limits

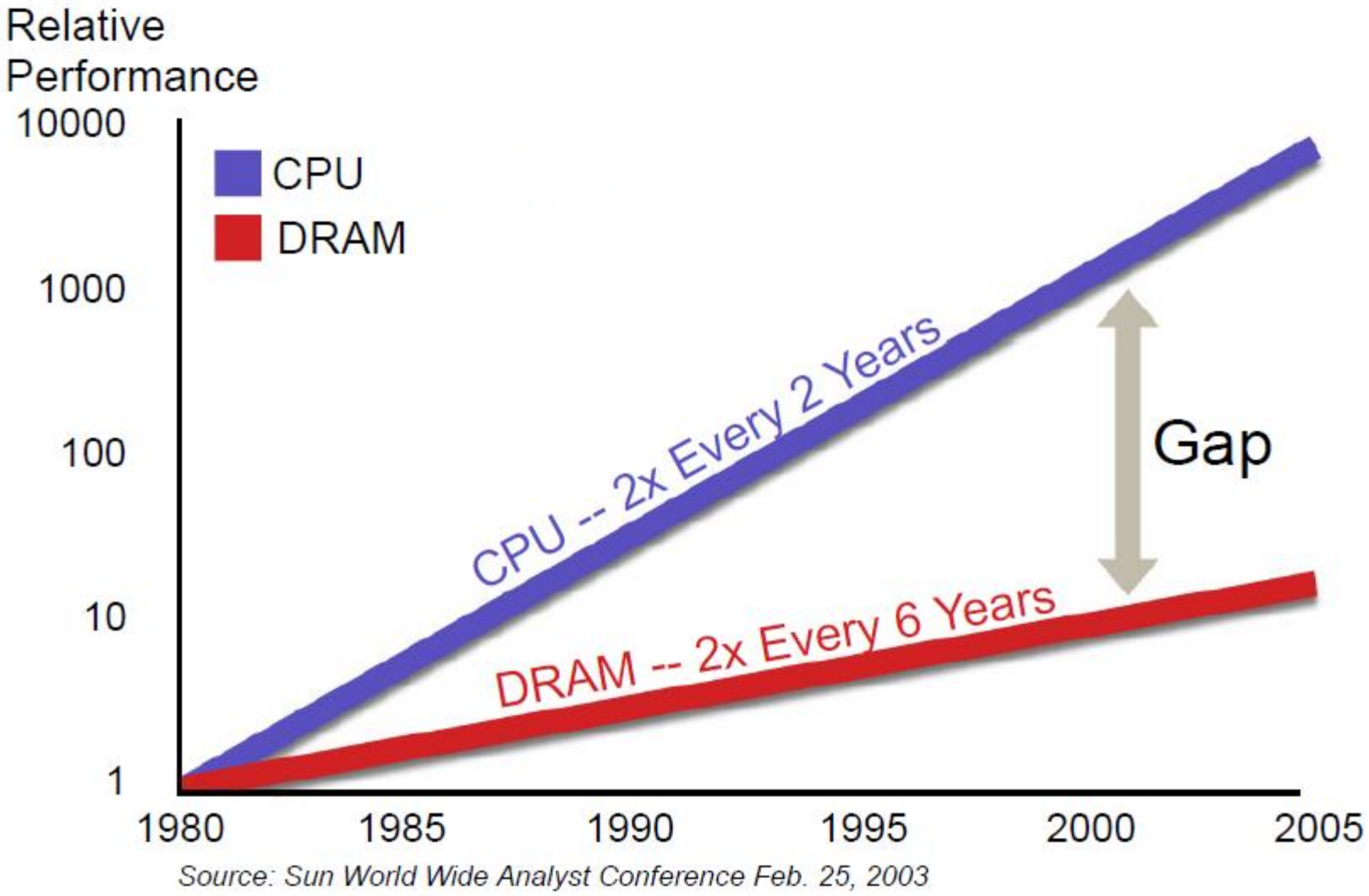
Execute the following loop on a serial machine in 1 second:

$$for\ (\ i = 0;\ i < 10^{12};\ ++i\ )$$
$$z[\ i\ ] = x[\ i\ ] + y[\ i\ ];$$

- We will have to access 3×10^{12} data items in one second
- Speed of light is, $c \approx 3 \times 10^{8}$ m/s
- So each data item must be within c / $3 \times 10^{12} \approx 0.1$ mm from the CPU on the average
- All data must be put inside a 0.2 mm × 0.2 mm square
- Each data item (≥ 8 bytes) can occupy only 1 $Å^2$ space!
  (size of a small atom!)

Source: Kathy Yelick and Jim Demmel, UC Berkeley

Unicore Performance: Memory Wall



Source: Rick Hetherington, Chief Technology Officer, Microelectronics, Sun Microsystems

Unicore Performance Has Hit a Wall!

Some Reasons

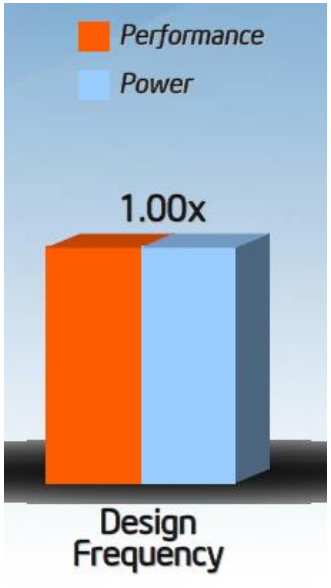
- Lack of additional ILP
  (Instruction Level Hidden Parallelism)
- High power density
- Manufacturing issues
- Physical limits
- Memory speed

"Oh Sinnerman, where you gonna run to?"

— Sinnerman (recorded by Nina Simone)

Where You Gonna Run To?

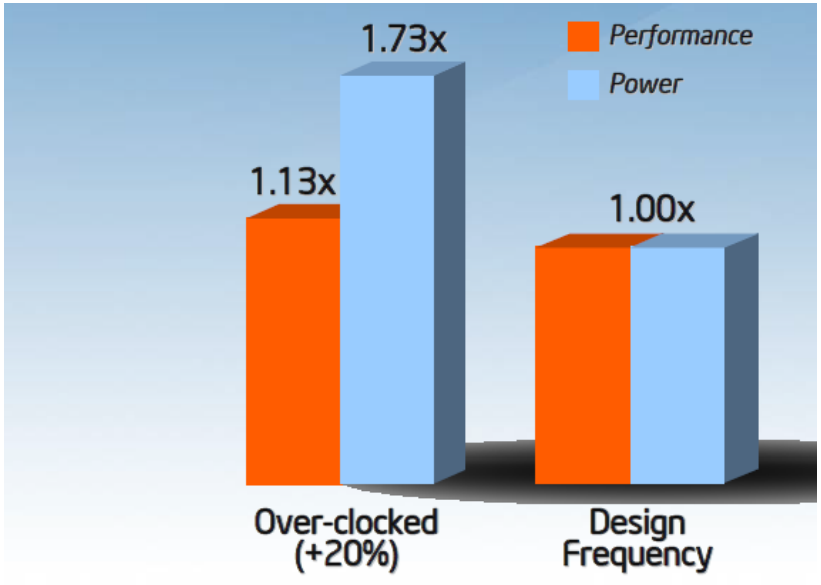
- Changing f by 20% changes performance by 13%
- So what happens if we overclock by 20%?



Source: Andrew A. Chien, Vice President of Research, Intel Corporation

Where You Gonna Run To?

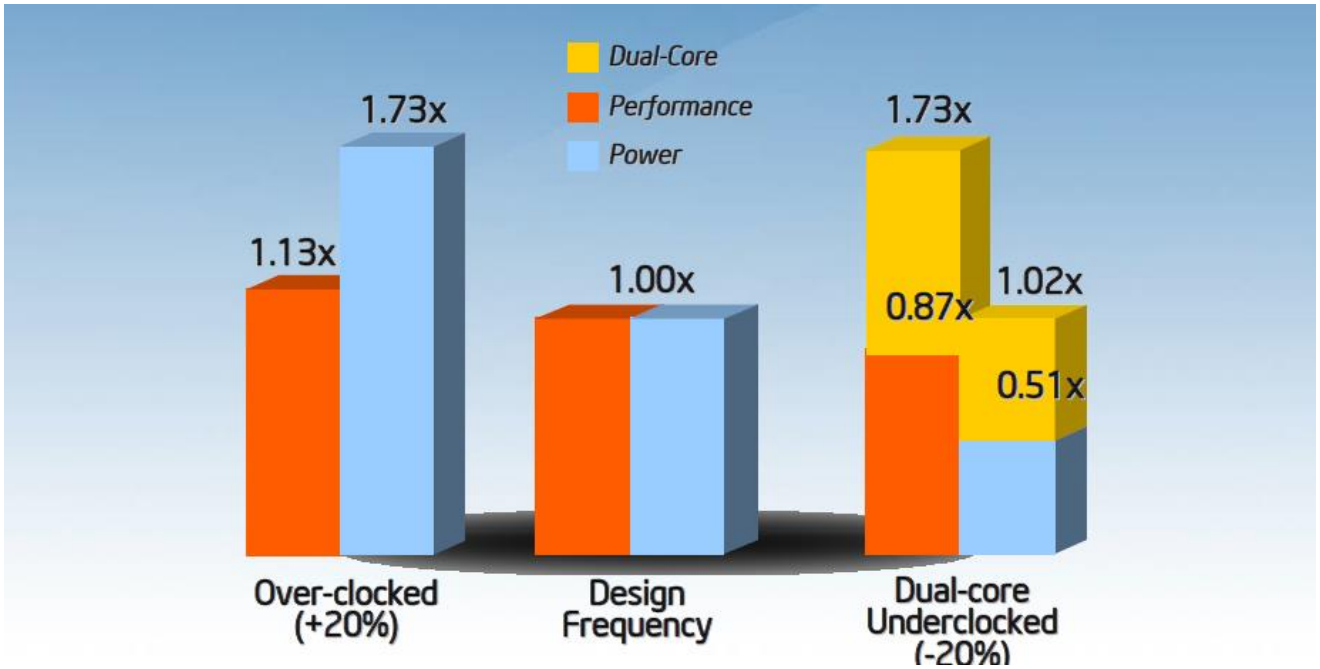
- Changing f by 20% changes performance by 13%
- So what happens if we overclock by 20%?
- And underclock by 20%?



Source: Andrew A. Chien, Vice President of Research, Intel Corporation

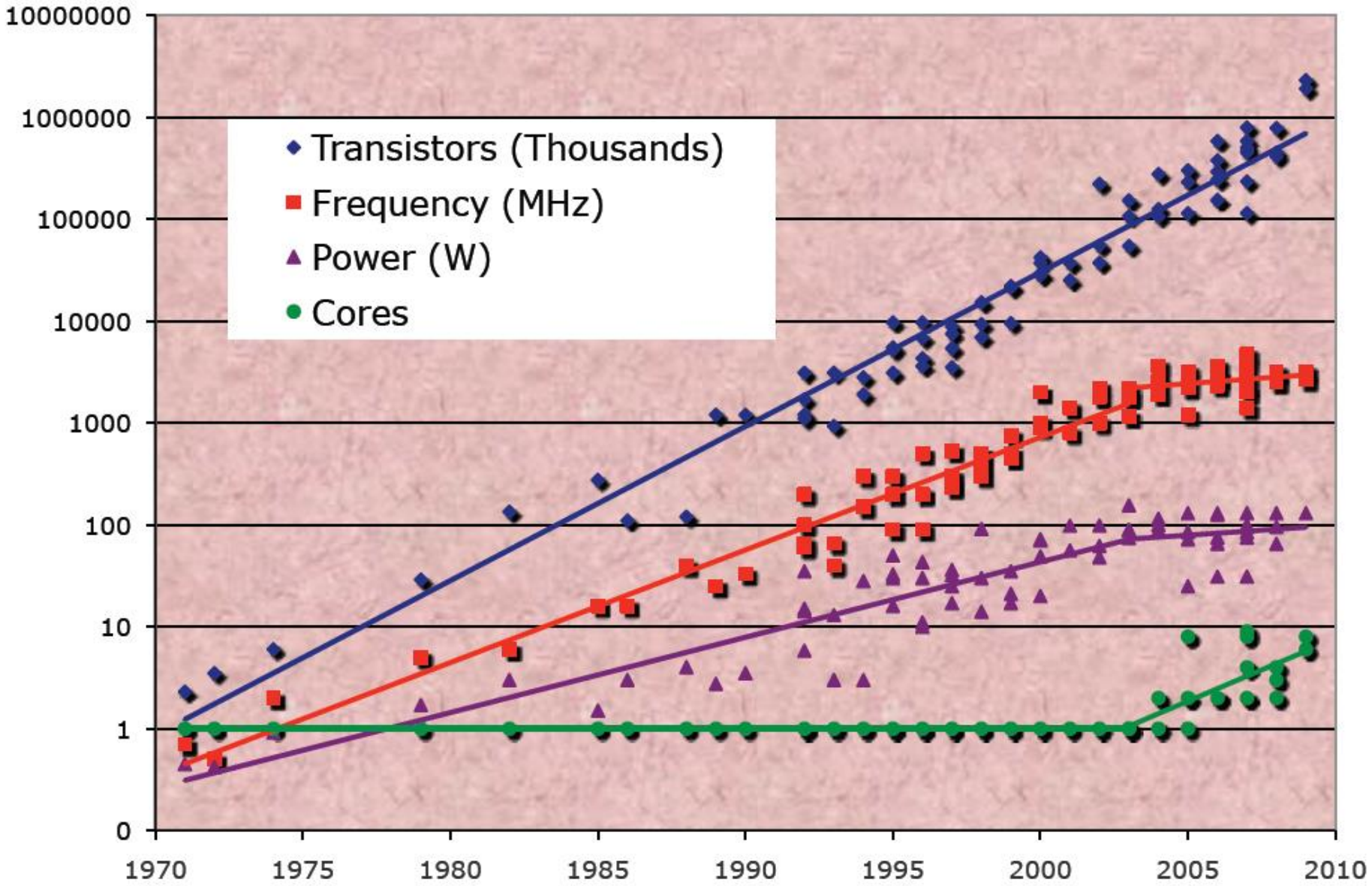
Where You Gonna Run To?

- Changing f by 20% changes performance by 13%
- So what happens if we overclock by 20%?
- And underclock by 20%?



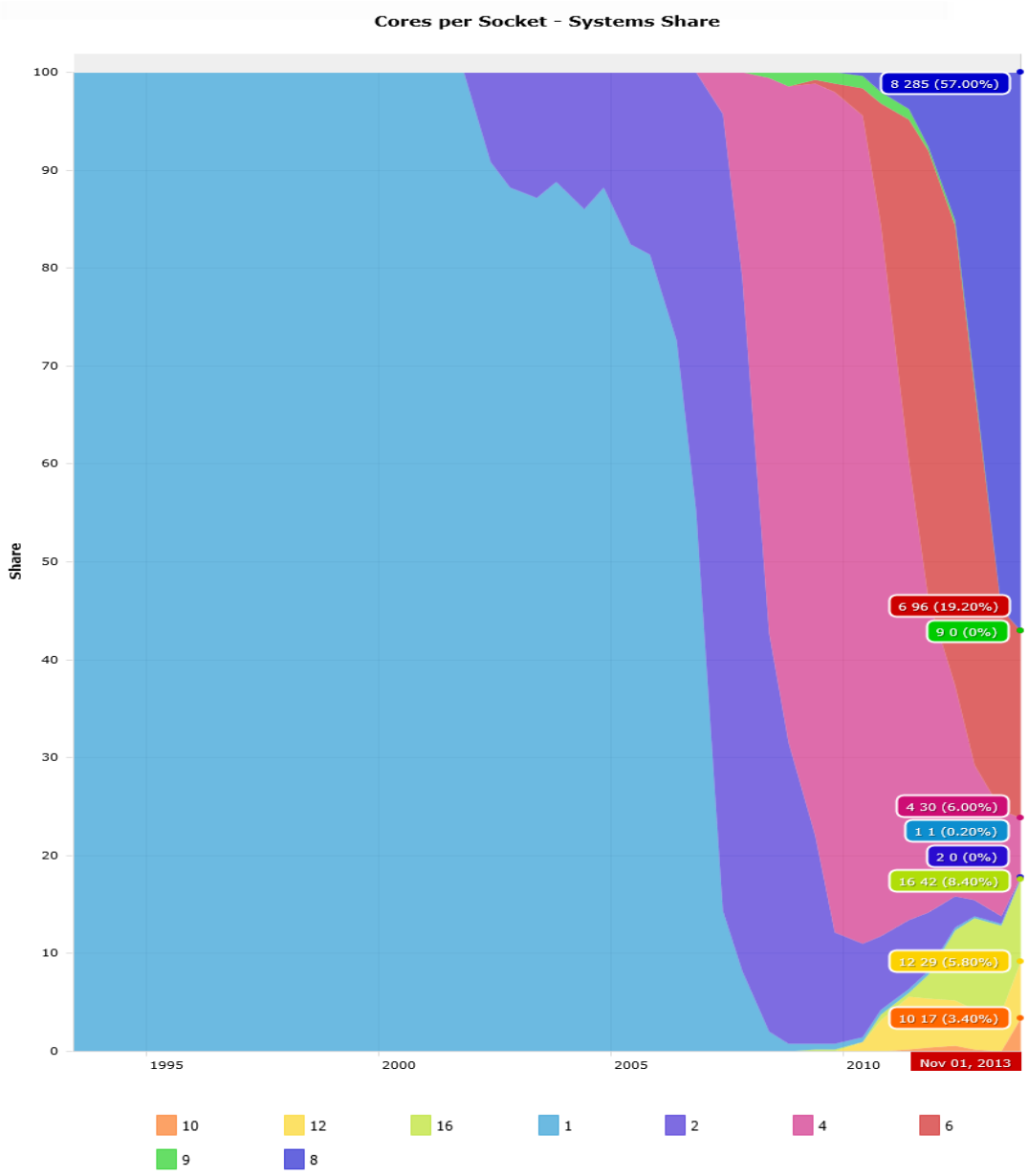
Source: Andrew A. Chien, Vice President of Research, Intel Corporation

Moore's Law Reinterpreted



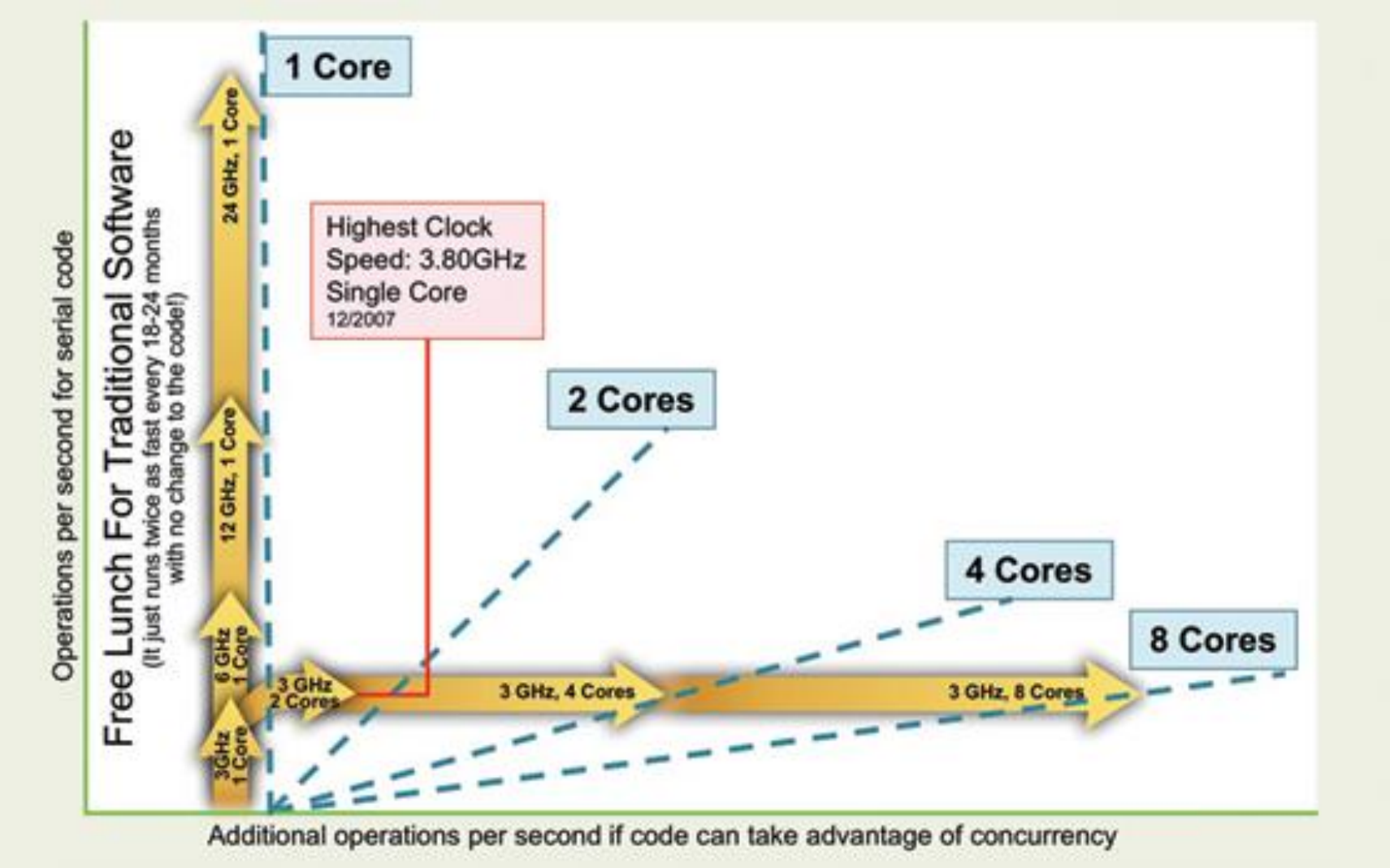
Source: Report of the 2011 Workshop on Exascale Programming Challenges

Top 500 Supercomputing Sites (Cores / Socket)



Source: www.top500.org

No Free Lunch for Traditional Software

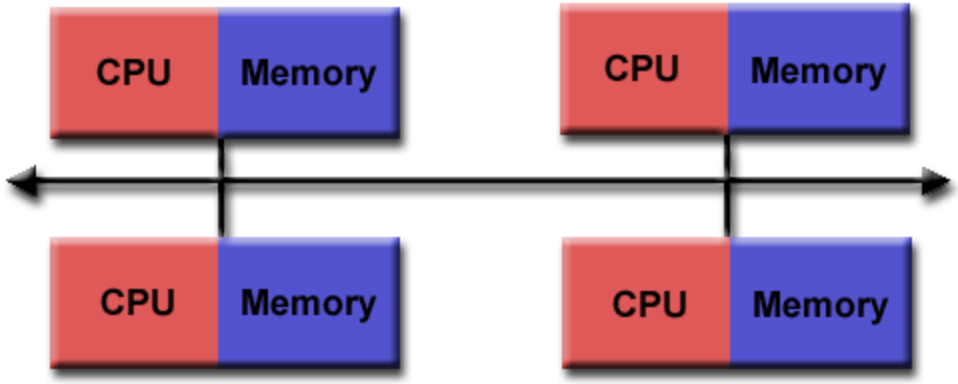


Source: Simon Floyd, Workstation Performance: Tomorrow's Possibilities (Viewpoint Column)

A Useful Classification of Parallel Computers

Parallel Computer Memory Architecture (Distributed Memory)

- Each processor has its own local memory — no global address space
- Changes in local memory by one processor have no effect on memory of other processors

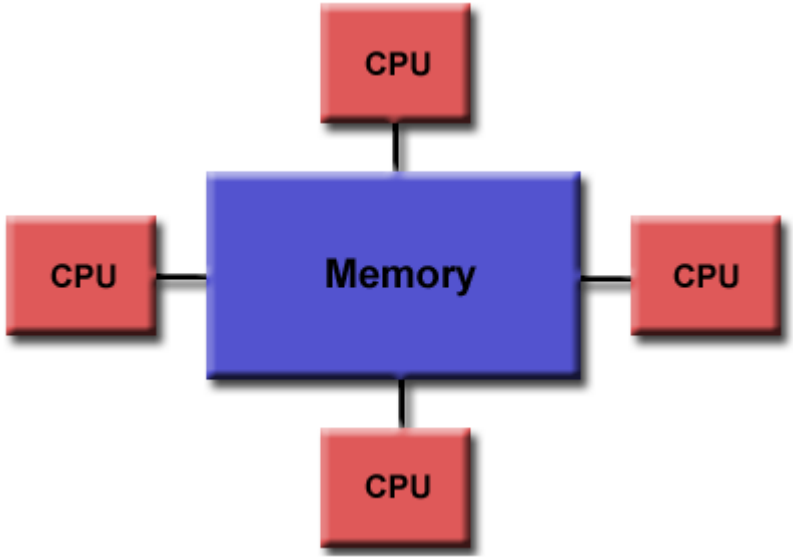


Source: Blaise Barney, LLNL

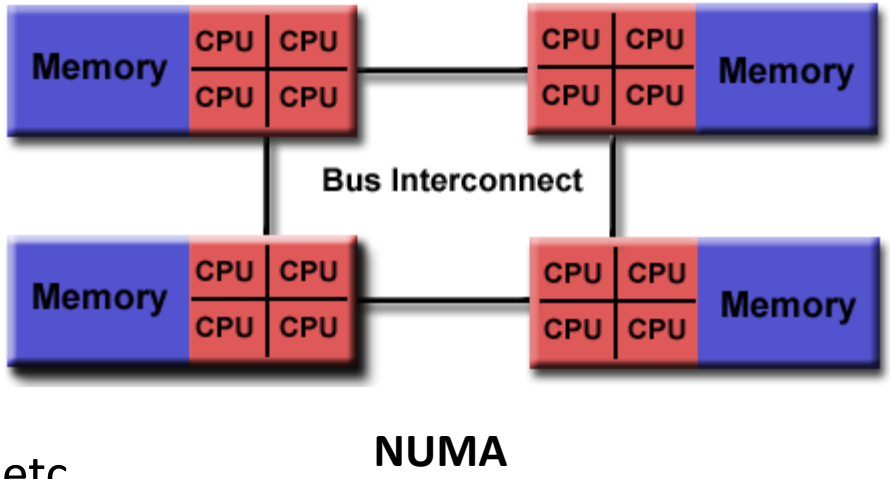
- Communication network to connect inter-processor memory
- Programming
  - Message Passing Interface (MPI)
  - Many once available: PVM, Chameleon, MPL, NX, etc.

Parallel Computer Memory Architecture (Shared Memory)

- All processors access all memory as global address space
- Changes in memory by one processor are visible to all others
- Two types
  - Uniform Memory Access (UMA)
  - Non-Uniform Memory Access (NUMA)
- Programming
  - Open Multi-Processing (OpenMP)
  - Cilk/Cilk++ and Intel Cilk Plus
  - Intel Thread Building Block (TBB), etc.



UMA

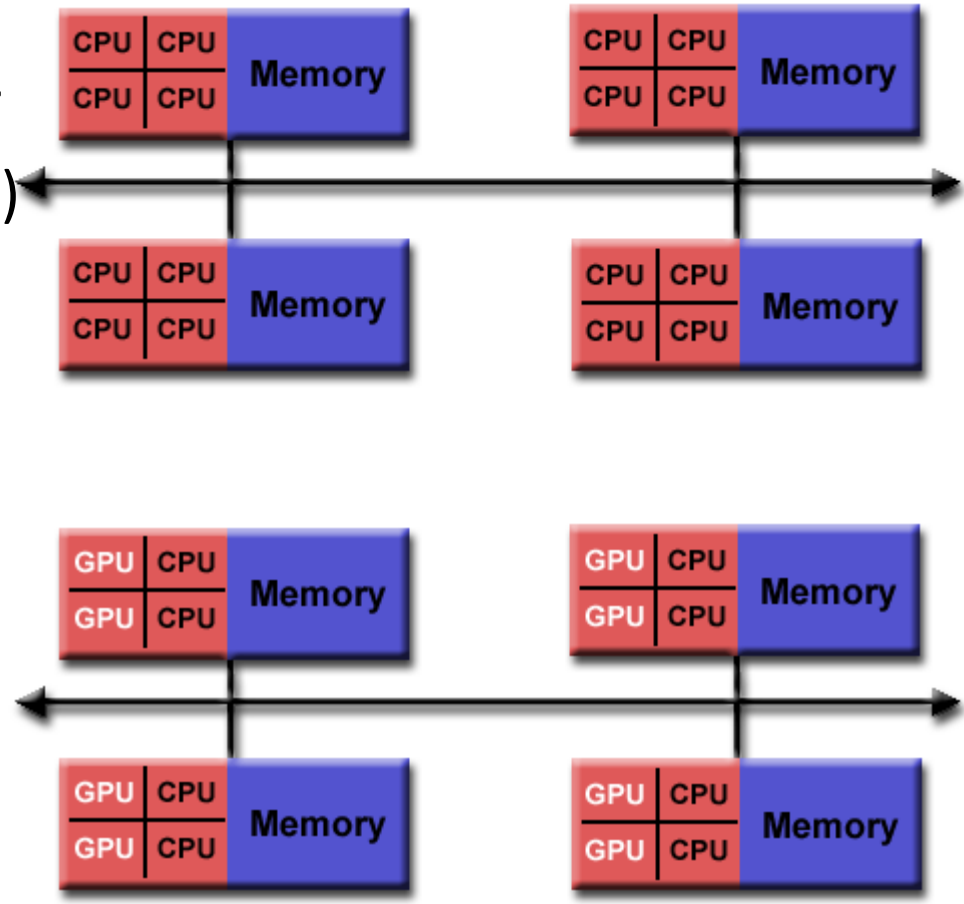


NUMA

Source: Blaise Barney, LLNL

Parallel Computer Memory Architecture (Hybrid Distributed-Shared Memory)

- The share-memory component can be a cache-coherent SMP or a Graphics Processing Unit (GPU)
- The distributed-memory component is the networking of multiple SMP/GPU machines
- Most common architecture for the largest and fastest computers in the world today
- Programming
  - OpenMP / Cilk + CUDA / OpenCL + MPI, etc.



Source: Blaise Barney, LLNL

Types of Parallelism

Nested Parallelism

$$ {}^{n}C_{r} = {}^{n-1}C_{r-1} + {}^{n-1}C_{r} $$

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = comb( n - 1, r - 1 );
  y = comb( n - 1, r );

  return ( x + y );
}
```

Serial Code

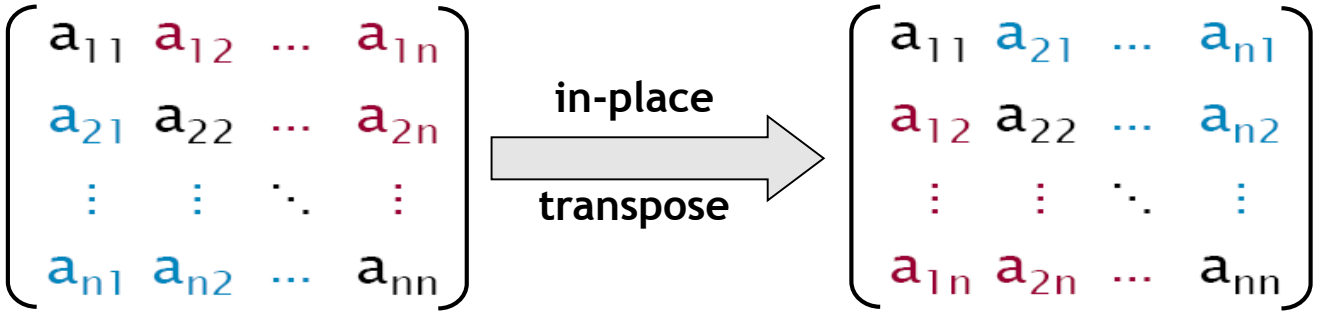
Grant permission to execute the called (spawned) function in parallel with the caller.

Control cannot pass this point until all spawned children have returned.

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n - 1, r - 1 );
  y = comb( n - 1, r );

  sync;

  return ( x + y );
}
```

Parallel Code

Loop Parallelism




```
for ( int i = 1; i < n; ++i )
   for ( int j = 0; j < i; ++j )
      {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
      }
```

Allows all iterations of the loop to be executed in parallel.

Can be converted to spawns and syncs using recursive divide-and-conquer.

Serial Code

```
parallel for ( int i = 1; i < n; ++i )
   for ( int j = 0; j < i; ++j )
      {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
      }
```

Parallel Code

Analyzing Parallel Algorithms

Speedup

Let T_p = running time using p identical processing elements

Speedup, $S_p = \frac{T_1}{T_p}$

Theoretically, $S_p \leq p$

Perfect or *linear* or *ideal* speedup if $S_p = p$

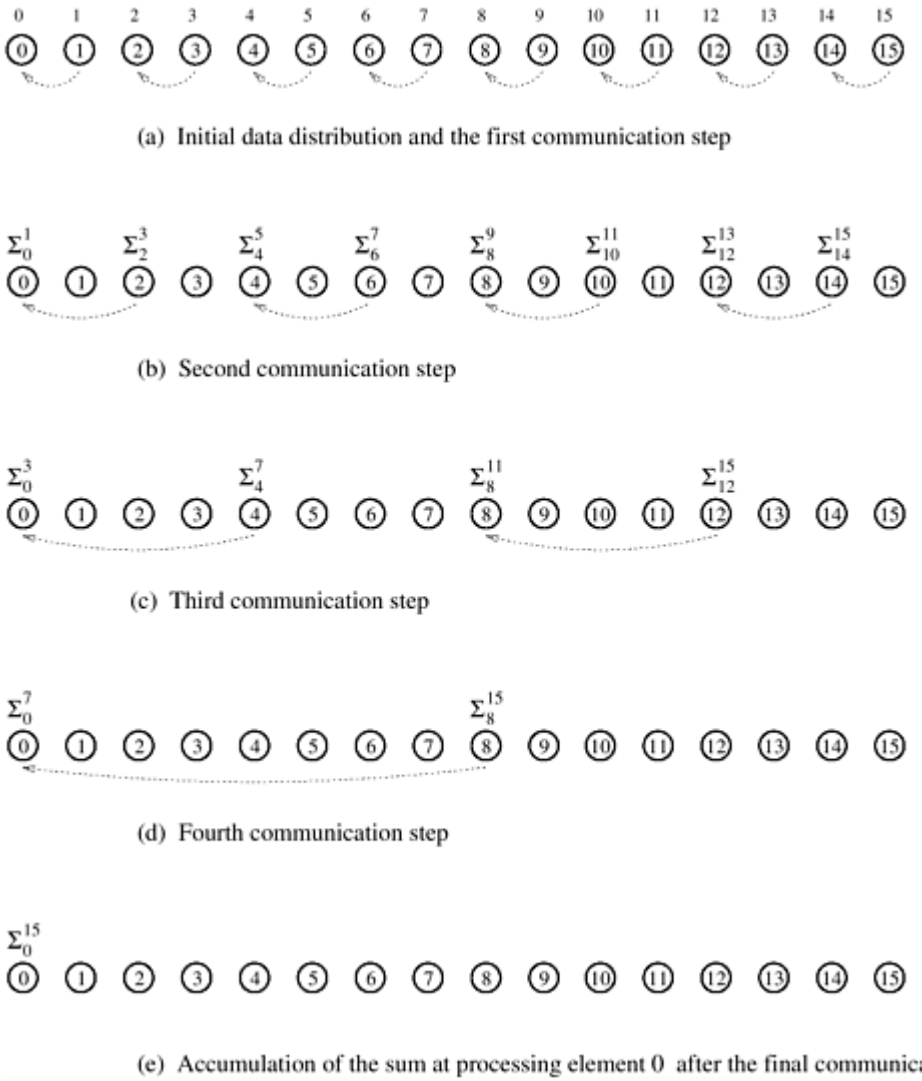
Speedup

Consider adding n numbers using n identical processing elements.

Serial runtime, $T = \Theta(n)$

Parallel runtime, $T_n = \Theta(\log n)$

Speedup, $S_n = \frac{T_1}{T_n} = \Theta\left(\frac{n}{\log n}\right)$



Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Parallelism & Span Law

We defined, T_p = runtime on p identical processing elements

Then span, T_∞ = runtime on an infinite number of identical processing elements

Parallelism, $P = \frac{T_1}{T_\infty}$

Parallelism is an upper bound on speedup, i.e., $S_p \leq P$

Span Law

$$T_p \geq T_\infty$$

Work Law

The cost of solving (or work performed for solving) a problem:

On a Serial Computer: is given by T_1

On a Parallel Computer: is given by pT_p

Work Law

$$T_p \geq \frac{T_1}{p}$$

Bounding Parallel Running Time (T_p)

A *runtime/online scheduler* maps tasks to processing elements dynamically at runtime.

A *greedy scheduler* never leaves a processing element idle if it can map a task to it.

Theorem [Graham'68, Brent'74]: For any greedy scheduler,

$$T_p \leq \frac{T_1}{p} + T_\infty$$

Corollary: For any greedy scheduler,

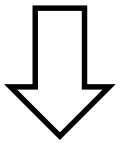
$$T_p \leq 2T_p^*,$$

where T_p^* is the running time due to optimal scheduling on *p* processing elements.

Analyzing Parallel Matrix Multiplication

Parallel Iterative MM

```
Iter-MM ( Z, X, Y )          { X, Y, Z are n × n matrices,
                               where n is a positive integer }
1.  for i ← 1 to n do
2.      for j ← 1 to n do
3.          Z[ i ][ j ] ← 0
4.          for k ← 1 to n do
5.              Z[ i ][ j ] ← Z[ i ][ j ] + X[ i ][ k ] · Y[ k ][ j ]
```



```
Par-Iter-MM ( Z, X, Y )      { X, Y, Z are n × n matrices,
                               where n is a positive integer }
1.  parallel for i ← 1 to n do
2.      parallel for j ← 1 to n do
3.          Z[ i ][ j ] ← 0
4.          for k ← 1 to n do
5.              Z[ i ][ j ] ← Z[ i ][ j ] + X[ i ][ k ] · Y[ k ][ j ]
```

Parallel Iterative MM

```
Par-Iter-MM ( Z, X, Y )        { X, Y, Z are n × n matrices,
                                  where n is a positive integer }
1.  parallel for i ← 1 to n do
2.      parallel for j ← 1 to n do
3.          Z[ i ][ j ] ← 0
4.          for k ← 1 to n do
5.              Z[ i ][ j ] ← Z[ i ][ j ] + X[ i ][ k ] · Y[ k ][ j ]
```

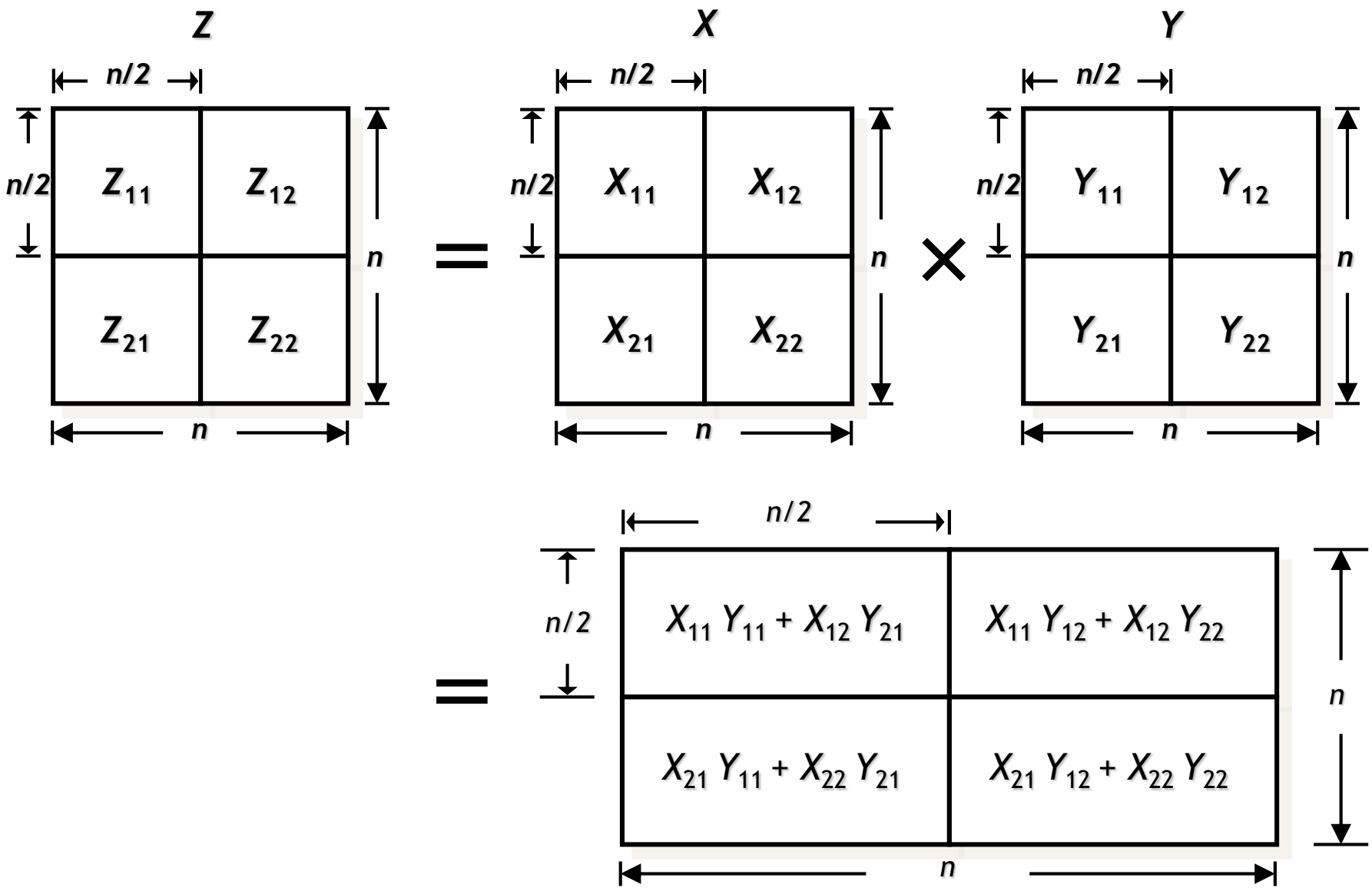
Work: $T_1(n) = \Theta(n^3)$

Span: $T_\infty(n) = \Theta(n)$

Parallel Running Time: $T_p(n) = \mathrm{O}\left(\frac{T_1(n)}{p} + T_\infty(n)\right) = \mathrm{O}\left(\frac{n^3}{p} + n\right)$

Parallelism: $\frac{T_1(n)}{T_\infty(n)} = \Theta(n^2)$

Parallel Recursive MM



Parallel Recursive MM

```
Par-Rec-MM ( Z, X, Y )     { X, Y, Z are n × n matrices,
                               where n = 2^k for integer k ≥ 0 }
 1. if n = 1 then
 2.     Z ← Z + X · Y
 3. else
 4.     spawn Par-Rec-MM ( Z11, X11, Y11 )
 5.     spawn Par-Rec-MM ( Z12, X11, Y12 )
 6.     spawn Par-Rec-MM ( Z21, X21, Y11 )
 7.           Par-Rec-MM ( Z21, X21, Y12 )
 8.     sync
 9.     spawn Par-Rec-MM ( Z11, X12, Y21 )
10.     spawn Par-Rec-MM ( Z12, X12, Y22 )
11.     spawn Par-Rec-MM ( Z21, X22, Y21 )
12.           Par-Rec-MM ( Z22, X22, Y22 )
13.     sync
14. endif
```

Parallel Recursive MM

```
Par-Rec-MM ( Z, X, Y )    { X, Y, Z are n × n matrices,
                              where n = 2^k for integer k ≥ 0 }
 1. if n = 1 then
 2.     Z ← Z + X · Y
 3. else
 4.     spawn Par-Rec-MM ( Z11, X11, Y11 )
 5.     spawn Par-Rec-MM ( Z12, X11, Y12 )
 6.     spawn Par-Rec-MM ( Z21, X21, Y11 )
 7.           Par-Rec-MM ( Z21, X21, Y12 )
 8.     sync
 9.     spawn Par-Rec-MM ( Z11, X12, Y21 )
10.     spawn Par-Rec-MM ( Z12, X12, Y22 )
11.     spawn Par-Rec-MM ( Z21, X22, Y21 )
12.           Par-Rec-MM ( Z22, X22, Y22 )
13.     sync
14. endif
```

Work:

$$T_1(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 8T_1\left(\frac{n}{2}\right) + \Theta(1), & \text{otherwise.} \end{cases}$$

$$= \Theta(n^3)$$

Span:

$$T_\infty(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 2T_\infty\left(\frac{n}{2}\right) + \Theta(1), & \text{otherwise.} \end{cases}$$

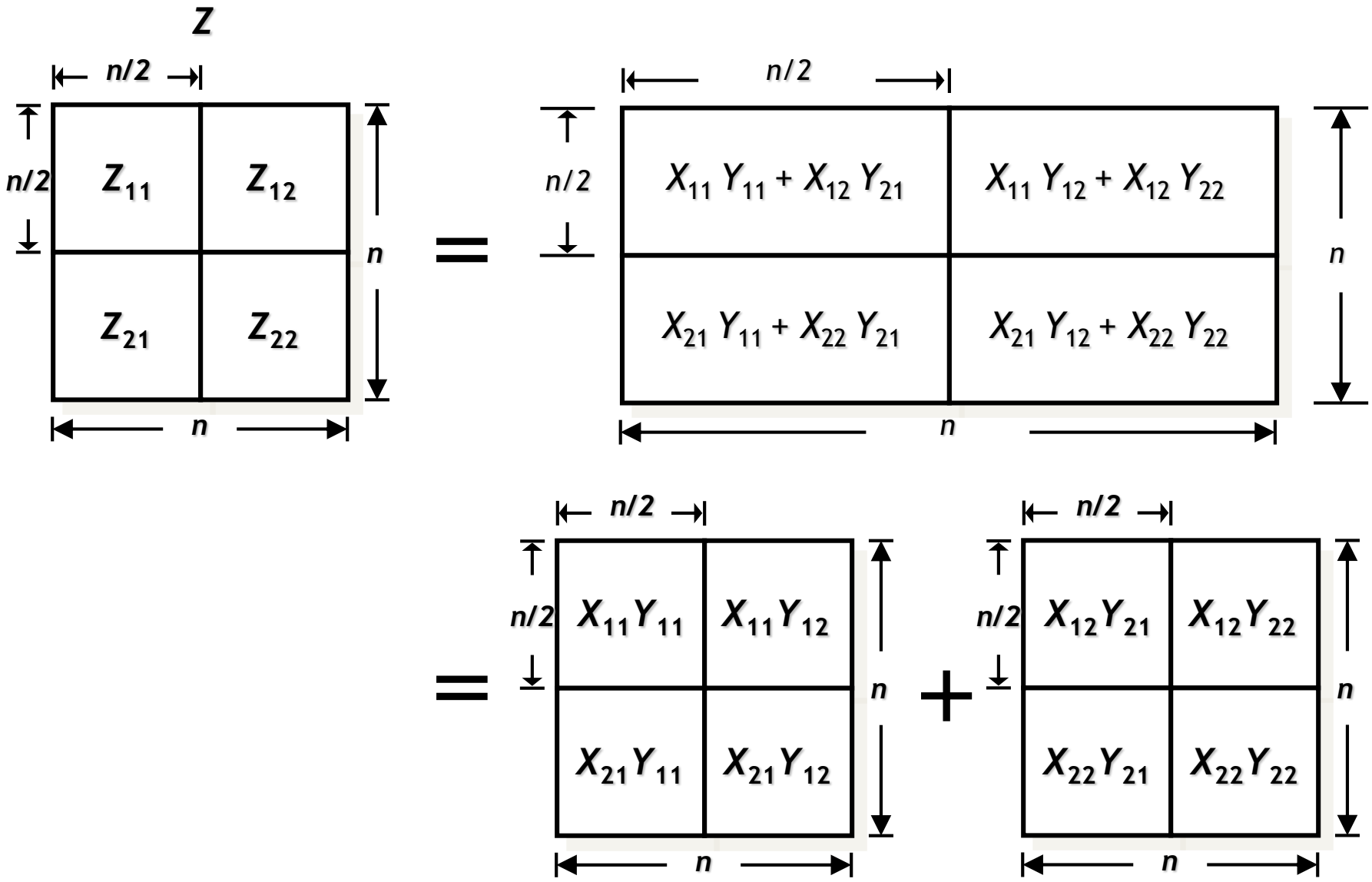
$$= \Theta(n)$$

Parallelism: $\frac{T_1(n)}{T_\infty(n)} = \Theta(n^2)$

Additional Space:

$$s_\infty(n) = \Theta(1)$$

Recursive MM with More Parallelism



Recursive MM with More Parallelism

```
Par-Rec-MM2 ( Z, X, Y )      { X, Y, Z are n × n matrices,
                               where n = 2^k for integer k ≥ 0 }
 1. if n = 1 then
 2.     Z ← Z + X · Y
 3. else          { T is a temporary n × n matrix }
 4.     spawn Par-Rec-MM2 ( Z11, X11, Y11 )
 5.     spawn Par-Rec-MM2 ( Z12, X11, Y12 )
 6.     spawn Par-Rec-MM2 ( Z21, X21, Y11 )
 7.     spawn Par-Rec-MM2 ( Z21, X21, Y12 )
 8.     spawn Par-Rec-MM2 ( T11, X12, Y21 )
 9.     spawn Par-Rec-MM2 ( T12, X12, Y22 )
10.     spawn Par-Rec-MM2 ( T21, X22, Y21 )
11.           Par-Rec-MM2 ( T22, X22, Y22 )
12.     sync
13.     parallel for i ← 1 to n do
14.         parallel for j ← 1 to n do
15.             Z[ i ][ j ] ← Z[ i ][ j ] + T[ i ][ j ]
16. endif
```

Recursive MM with More Parallelism

```
Par-Rec-MM2 ( Z, X, Y )      { X, Y, Z are n × n matrices,
                               where n = 2^k for integer k ≥ 0 }
 1. if n = 1 then
 2.     Z ← Z + X · Y
 3. else          { T is a temporary n × n matrix }
 4.     spawn Par-Rec-MM2 ( Z11, X11, Y11 )
 5.     spawn Par-Rec-MM2 ( Z12, X11, Y12 )
 6.     spawn Par-Rec-MM2 ( Z21, X21, Y11 )
 7.     spawn Par-Rec-MM2 ( Z21, X21, Y12 )
 8.     spawn Par-Rec-MM2 ( T11, X12, Y21 )
 9.     spawn Par-Rec-MM2 ( T12, X12, Y22 )
10.     spawn Par-Rec-MM2 ( T21, X22, Y21 )
11.           Par-Rec-MM2 ( T22, X22, Y22 )
12.     sync
13.     parallel for i ← 1 to n do
14.         parallel for j ← 1 to n do
15.             Z[ i ][ j ] ← Z[ i ][ j ] + T[ i ][ j ]
16. endif
```

Work:

$$T_1(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 8T_1\left(\frac{n}{2}\right) + \Theta(n^2), & \text{otherwise.} \end{cases}$$

$$= \Theta(n^3)$$

Span:

$$T_\infty(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ T_\infty\left(\frac{n}{2}\right) + \Theta(\log n), & \text{otherwise.} \end{cases}$$

$$= \Theta(\log^2 n)$$

Parallelism: $\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n^3}{\log^2 n}\right)$

Additional Space:

$$s_\infty(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 8s_\infty\left(\frac{n}{2}\right) + \Theta(n^2), & \text{otherwise.} \end{cases}$$

$$= \Theta(n^3)$$

Distributed-Memory Naïve Matrix Multiplication

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

```
Iter-MM ( X, Y, Z, n )
1.  for i ← 1 to n do
2.      for j ← 1 to n do
3.          for k ← 1 to n do
4.              z_ij ← z_ij + x_ik × y_kj
```

Distributed-Memory Naïve Matrix Multiplication

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

Suppose we have $p = n \times n$ processors, and processor P_{ij} is responsible for computing z_{ij}.

Let's assume that one master processor initially holds both X and Y.

Each processor in the group $\{P_{i,1}, P_{i,2}, \ldots, P_{i,n}\}$ will require row i of X.

Similarly, for other rows of X, and all columns of Y.

Each P_{ij} computes z_{ij} and sends back to master.

Distributed-Memory Naïve Matrix Multiplication

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$$

$$\begin{bmatrix} z_{11} & z_{12} & \cdots & z_{1n} \\ z_{21} & z_{22} & \cdots & z_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \cdots & z_{nn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix} \times \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{bmatrix}$$

Let t_s be the startup time of a message, and

t_w be the per-word transfer time.

The communication complexity of broadcasting m units of data to a group of size n is $(t_s + mt_w) \log n$.

Communication complexity of sending one unit of data back to master is $(t_s + t_w)$.

Hence, $t_{comm} \leq 2n(t_s + nt_w) \log n \ + n^2(t_s + t_w)$.

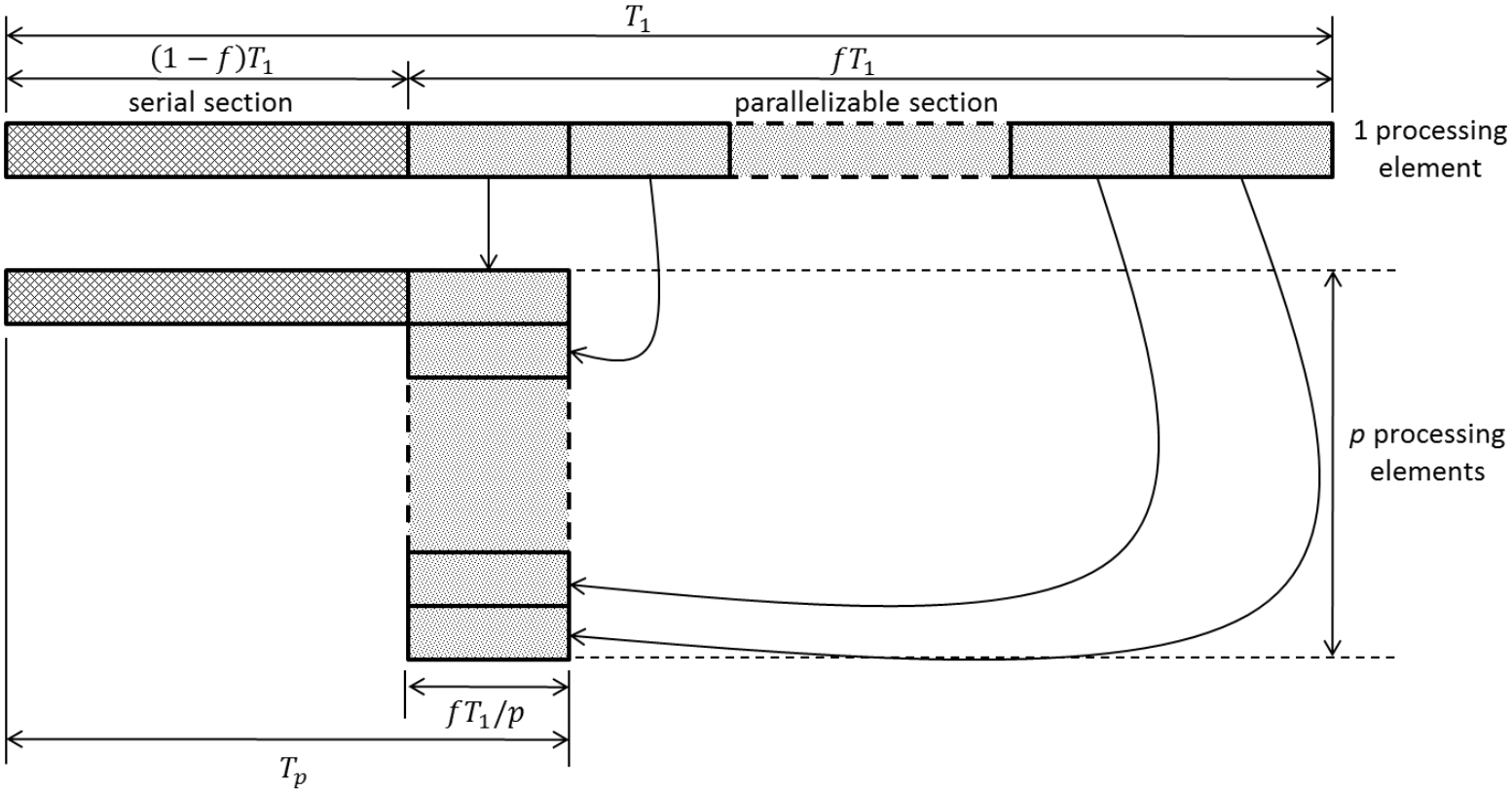
The $\log n$ factor vanishes because of pipelining

Also $t_{comp} = 2n$.

Finally, $T_p = t_{comp} + t_{comm}$.

Scaling Laws

Scaling of Parallel Algorithms (Amdahl's Law)



Suppose only a fraction f of a computation can be parallelized.

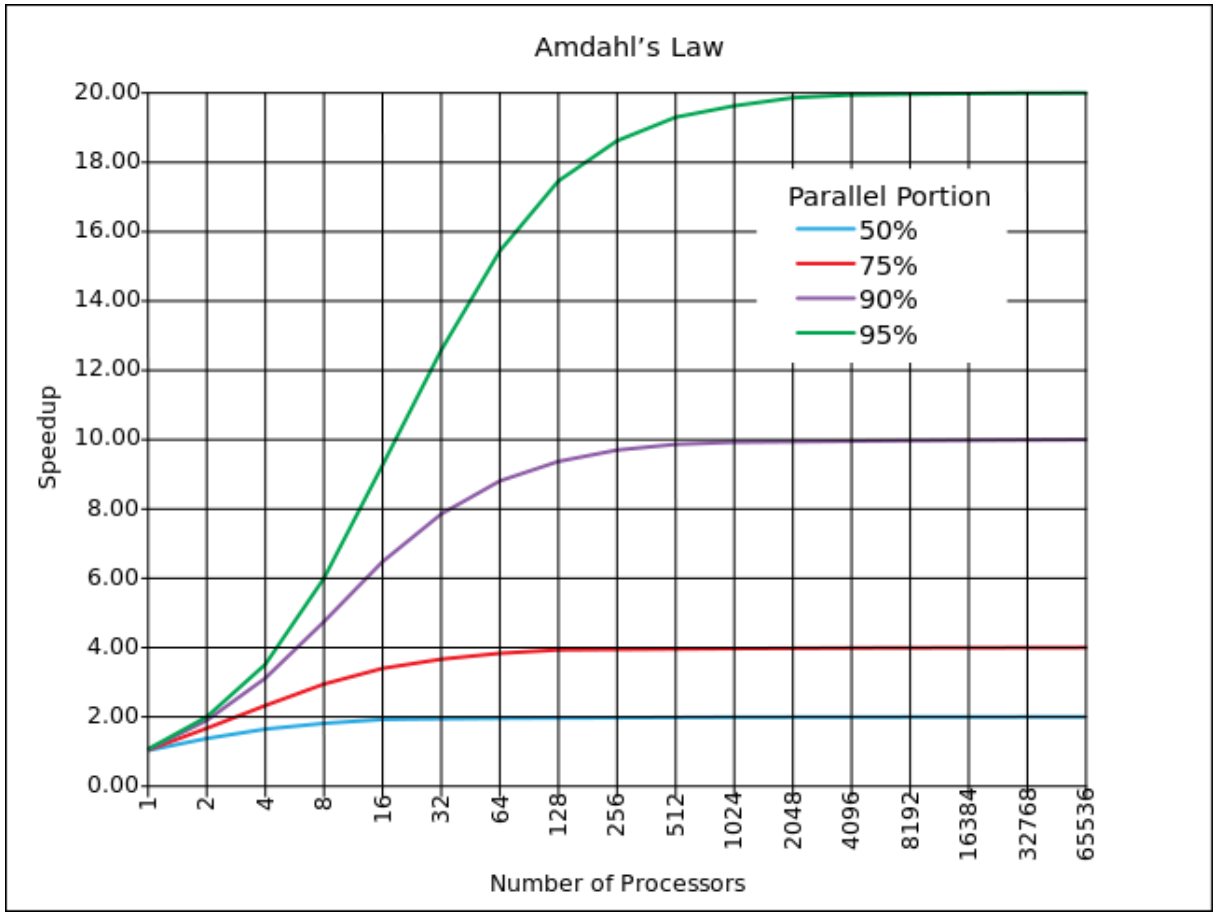
Then parallel running time, $T_p \geq (1-f)T_1 + f\frac{T_1}{p}$

Speedup, $S_p = \frac{T_1}{T_p} \leq \frac{p}{f+(1-f)p} = \frac{1}{(1-f)+\frac{f}{p}} \leq \frac{1}{1-f}$

Scaling of Parallel Algorithms (Amdahl's Law)

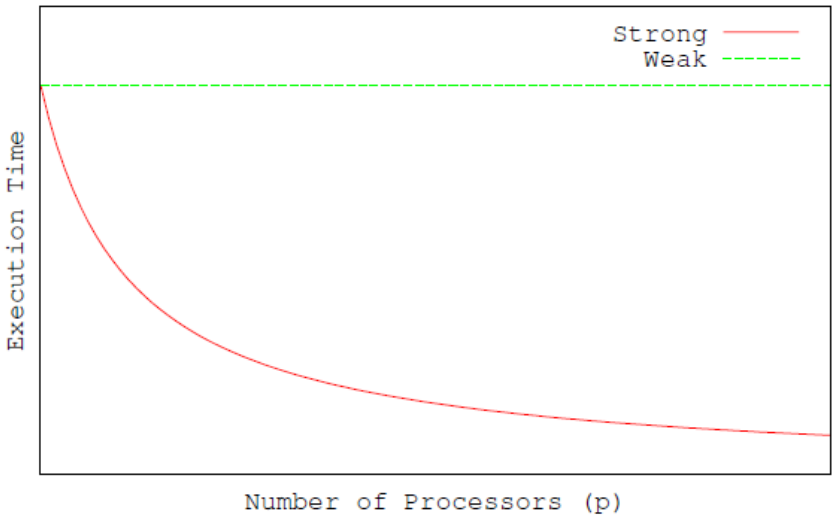
Suppose only a fraction f of a computation can be parallelized.

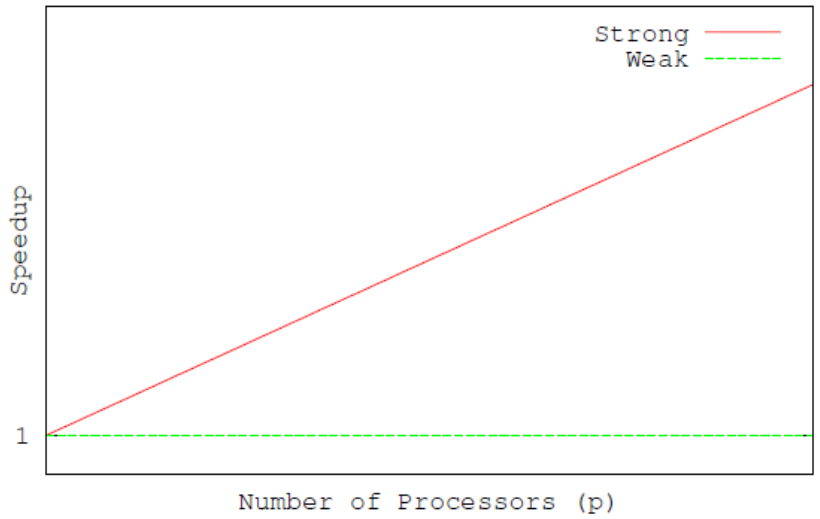
Speedup, $S_p = \frac{T_1}{T_p} \leq \frac{1}{(1-f)+\frac{f}{p}} \leq \frac{1}{1-f}$



Source: Wikipedia

Strong Scaling vs. Weak Scaling





Source: Martha Kim, Columbia University

Strong Scaling

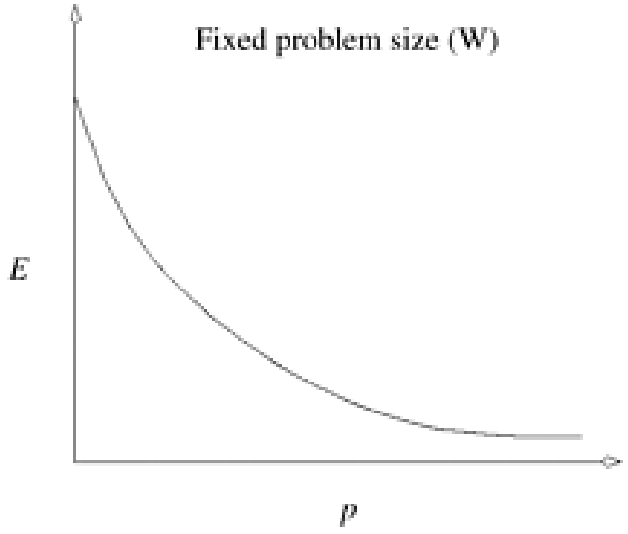
How T_p (or S_p) varies with p when the problem size is fixed.

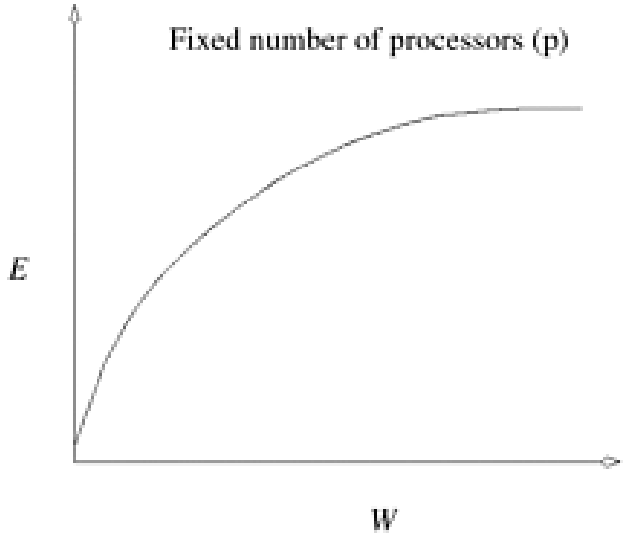
Weak Scaling

How T_p (or S_p) varies with p when the problem size per processing element is fixed.

Scalable Parallel Algorithms

Efficiency, $E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$





Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

A parallel algorithm is called *scalable* if its efficiency can be maintained at a fixed value by simultaneously increasing the number of processing elements and the problem size.

Scalability reflects a parallel algorithm's ability to utilize increasing processing elements effectively.

WORKSHOP SERIES

OPENMP/MPI TUTORIAL

"We used to joke that
"parallel computing is the future, and always will be,"
but the pessimists have been proven wrong."

— Tony Hey

Now Have Fun!