

CSE 262

Lecture 8

Performance
Communication Avoiding Matrix
Multiplication

Today's lecture

- Performance, measurement and metrics
- Communication performance
- Communication avoiding matrix multiplication

Measures of Performance

- Why do we measure performance?
- Measures of performance
 - ♦ Completion time
 - ♦ Processor time product
Completion time $\times$ # processors
 - ♦ Throughput: amount of work that can be accomplished in a given amount of time
 - ♦ Relative performance: given a reference architecture or implementation
AKA *Speedup*

Parallel Speedup and Efficiency

- How much of an improvement did our parallel algorithm obtain over the serial algorithm?
- Define the *parallel speedup*, S_P

$$S_P = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on } P \text{ processors}}$$

- T_1 is defined as the running time of the “best serial algorithm”
- In general: *not* the running time of the parallel algorithm on 1 processor
- **Definition:** *Parallel efficiency* $E_P = S_P/P$

Performance questions

- You observe the following running times for a parallel program running a fixed workload N
- Assume that the only losses are due to serial sections
- What is the speedup and efficiency on 8 processors?
- What will the running time be on 4 processors?
- What is the maximum possible speedup on an infinite number of processors?
- What fraction of the total running time on 1 processor corresponds to the serial section?
- What fraction of the total running time on 2 processors corresponds to the serial section?



NT	Time
1	10000
2	6000
8	3000

What can go wrong with speedup?

- Not always an accurate way to compare different algorithms....
- .. or the same algorithm running on different machines
- We might be able to obtain a better running time even if we lower the speedup
- If our goal is performance, the bottom line is running time T_p

Superlinear speedup

- We have a *super-linear* speedup when

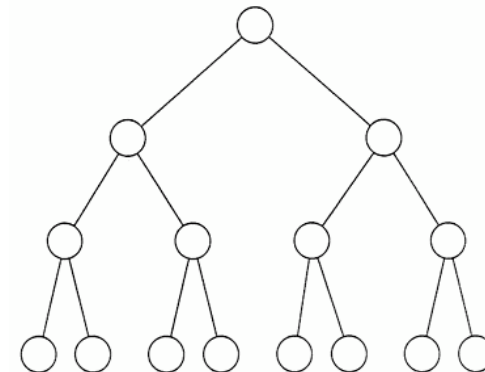
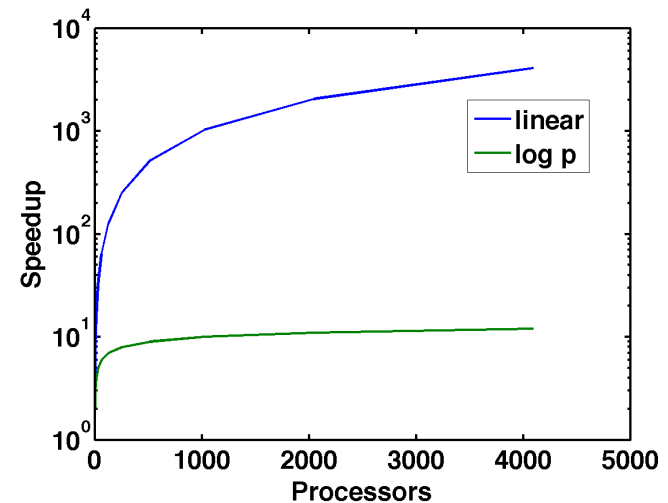
$$S_P > P \Rightarrow E_P > 1$$

- Super-linear speedups are often an artifact of inappropriate measurement technique
- Where there is a super-linear speedup, a better serial algorithm may be lurking

Scalability

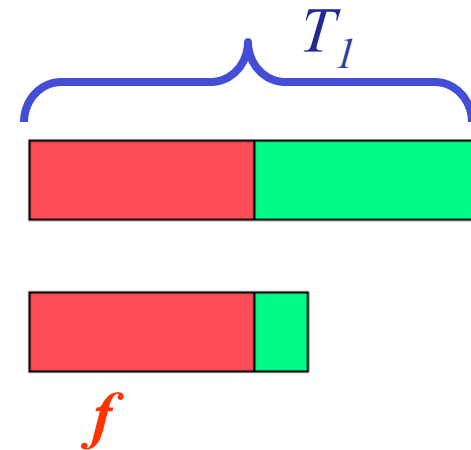
- A computation is **scalable** if performance increases as a “nice function” of the number of processors, e.g. linearly
- In practice scalability can be hard to achieve
 - ▶ Serial sections: code that runs on only one processor
 - ▶ “Non-productive” work associated with parallel execution, e.g. communication
 - ▶ Load imbalance: uneven work assignments over the processors
- Some algorithms present intrinsic barriers to scalability leading to alternatives

for $i=0:n-1$ $sum = sum + x[i]$



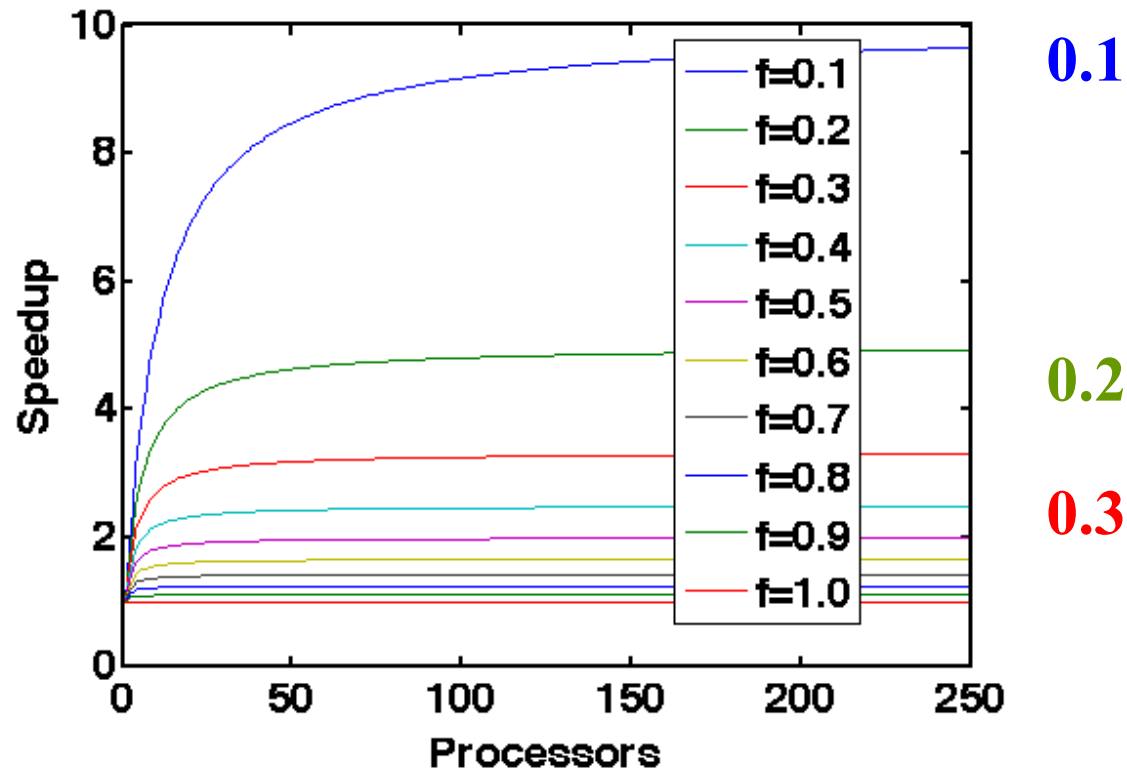
Serial Section

- Limits scalability
- Let f = the fraction of T_1 that runs serially
- $T_1 = f \times T_1 + (1-f) \times T_1$
- $T_p = f \times T_1 + (1-f) \times T_1 / P$
Thus $S_p = 1/[f + (1 - f)/p]$
- As $P \rightarrow \infty$, $S_p \rightarrow 1/f$
- This is known as *Amdahl's Law* (1967)



Amdahl's law (1967)

- A serial section limits scalability
- Let f = fraction of T_1 that runs serially
- *Amdahl's Law* (1967) : As $P \rightarrow \infty$, $S_P \rightarrow 1/f$



Weak scaling

- Is Amdahl's law pessimistic?
- Observation: Amdahl's law assumes that the workload (W) remains fixed
- But parallel computers are used to tackle more ambitious workloads
- If we increase W with P we have **weak scaling**

f often decreases with W

- We can continue to enjoy speedups
 - ♦ Gustafson's law [1992]
http://en.wikipedia.org/wiki/Gustafson's_law
www.scl.ameslab.gov/Publications/Gus/FixedTime/FixedTime.pdf

Computing scaled speedup

- Instead of asking what the speedup is, we ask: “how long a parallel program would run on a single processor ?”
- Let $T_P = 1$
- $f' =$ fraction of serial time spent on the parallel program
- $T_1 = f' + (1-f') \times P = S'_P =$ scaled speedup
- Scaled speedup is linear in P

Isoefficiency

- Consequence of Gustafson's observation is that we increase N with P
- Kumar: We can maintain constant efficiency so long as we increase N appropriately
- The *isoefficiency* function specifies the growth of N in terms of P
- If N is linear in P , we have a scalable computation
- Problem: the amount of memory per core is shrinking

Today's lecture

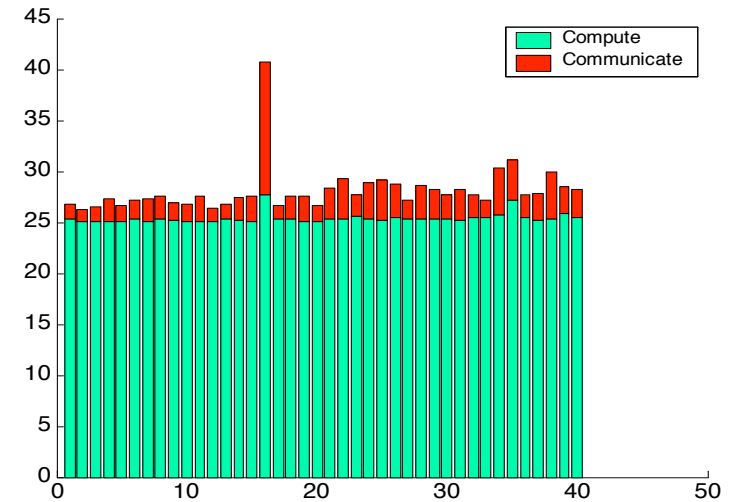
- Performance metrics
- **Performance Measurement**
- Communication performance
- Communication avoiding matrix multiplication

Challenges to measuring performance

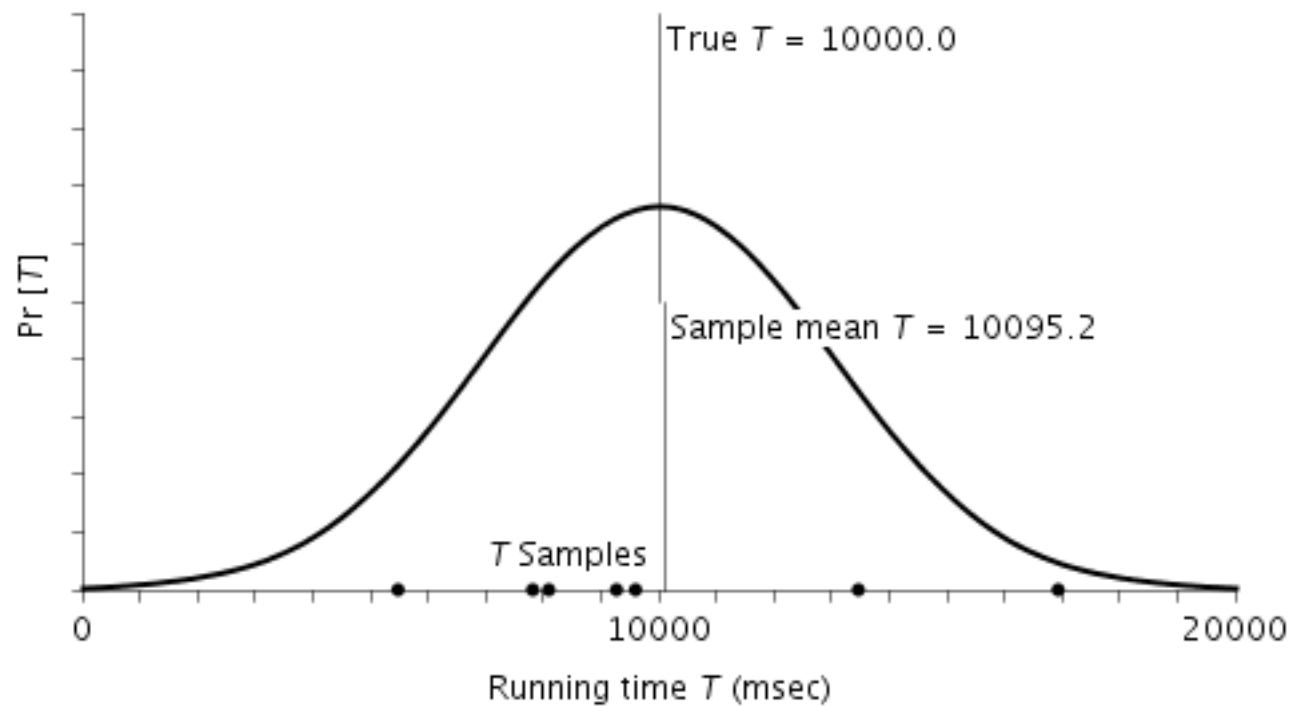
- Reproducibility
 - ◆ Transient system operating conditions
 - ◆ Differing systems or program configuration
- Measurements are imprecise
 - ◆ “Heisenberg uncertainty principle:”
measurement technique may affect performance
 - ◆ Overheads and inaccuracy
- Explain anomalous behavior, but ignore anomalies that are not significant
- Cost of measuring a full run is prohibitive
 - ◆ Ignore startup code if you plan to run for a much longer time in production

Measurement collection

- Report the *best* timings
 - ▶ Repeat results $\times 3$ to 5 until at least 2 measures agree to within... 5%, 10%
 - ▶ Report the minimum time
- Also report outliers
- A scatter plot or error bar can be useful

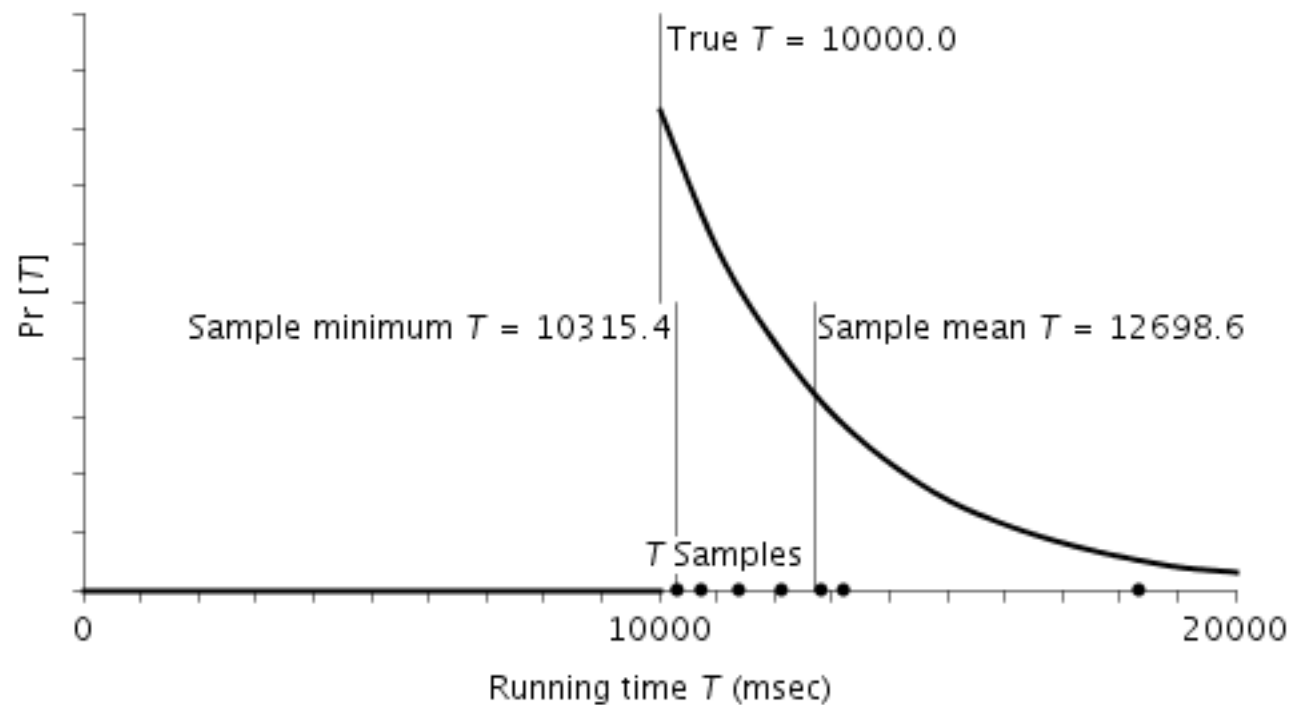


Why do we take the minimum time?



Alan Kaminsky. *Building Parallel Programs: SMPs, Clusters, and Java*. Copyright © 2010 Course Technology

Measurement errors are not distributed symmetrically



Timing collection

- Measures of time
 - ▶ Elapsed, or “wall clock” time
 - ▶ CPU time = system + user time
 - ▶ Overhead, resolution, and quantization effects
- Measurement tools
 - ▶ Can be platform dependent, especially library routines
 - ▶ Unix `time` command does a reasonable job for long-running programs
 - ▶ `gettimeofday()`

Enable others to reproduce your results

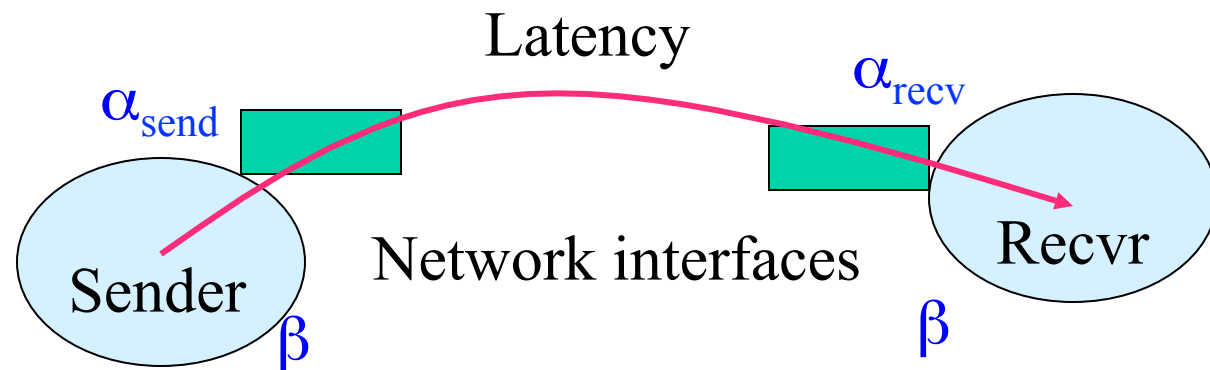
- Builds confidence within a community
- Report where you ran, software versions, processor, etc.
 - ▶ `uname -a`
 - ▶ Linux lilliput 2.6.35-30-server #61-Ubuntu SMP Tue Oct 11 18:09:44 UTC 2011 x86_64 GNU/Linux
 - ▶ `gcc --version`
 - ▶ gcc version 4.4.5 (Ubuntu/Linaro 4.4.4-14ubuntu5)
 - ▶ `icpc --version`
 - ▶ icpc (ICC) 12.0.2 20110112
 - ▶ `nvcc --version`
 - ▶ Cuda compilation tools, release 4.0, V0.2.1221
 - ▶ Access processor configuration information
 - ▶ Device # 0 has 30 cores
 - ▶ Device # 1 has 4 cores
 - ▶ Choosing device 0
 - ▶ Device is a GeForce GTX 285, capability: 1.3
 - ▶ CUDA Driver version: 2030, runtime version: 2030

Today's lecture

- Performance metrics
- Performance Measurement
- **Communication performance**
- Communication avoiding matrix multiplication

Message passing: where does the time go?

- Communication performance can be a major factor in determining application performance
- Under ideal conditions...
 - ♦ There is a pending receive waiting for an incoming message, which is transmitted directly to and from the user's message buffer
 - ♦ There is no other communication traffic
- Assume a contiguous message
- LogP model (Culler et al, 1993)



Communication performance

- The so-called α β model is often good enough

- Message passing time = $\alpha + \beta^{-1} n$

α = message startup time

β_{∞} = peak bandwidth (bytes per second)

n = message length

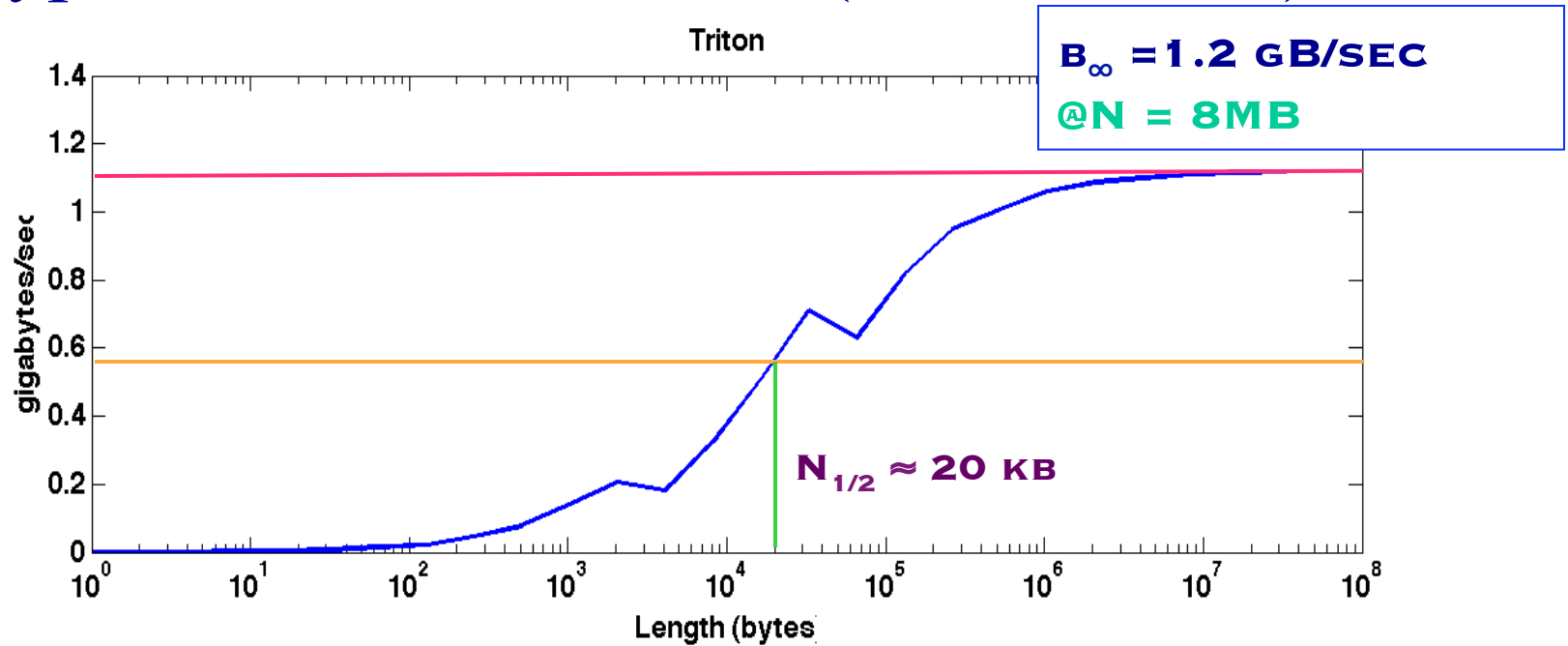
- “Short” messages: startup term dominates

$$\alpha \gg \beta^{-1} n$$

- “Long” messages: bandwidth term dominates

$$\beta^{-1} n \gg \alpha$$

Typical bandwidth curve (SDSC Triton)



$$\alpha = 3.2 \mu\text{sec}$$

Long Messages: $\beta^{-1} \infty n \gg \alpha$



Half power point

- $T(n)$ = time to send a message of length n
- Let $\beta(n)$ = the effective bandwidth

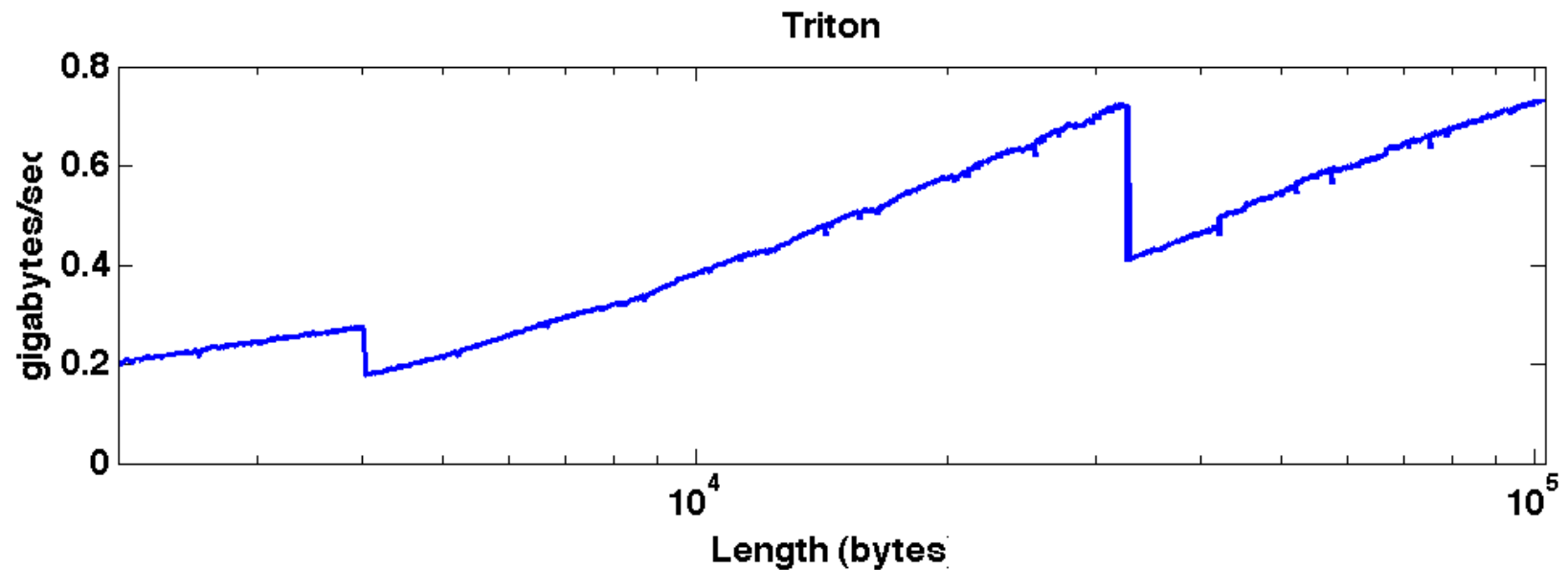
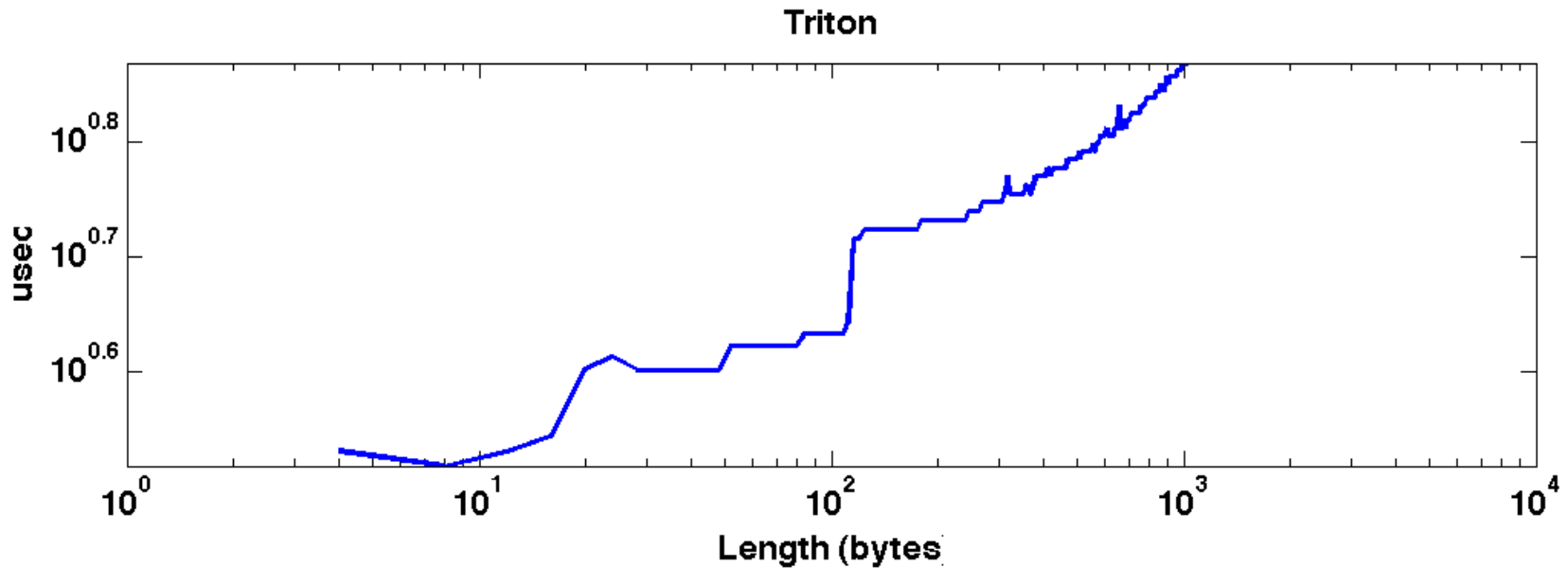
$$\beta^{-1}(n) = n / T(n)$$
- We define the *half power point* $n_{1/2}$ as the message size need to achieve $\frac{1}{2} \beta_{\infty}$

$$\frac{1}{2} \beta^{-1}_{\infty} = n_{1/2} / T(n_{1/2}) \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_{\infty}$$
- In theory, this occurs when $\alpha = \beta^{-1}_{\infty} n_{1/2}$

$$\Rightarrow n_{1/2} = \alpha \beta_{\infty}$$
- Generally not a good predictor of $n_{1/2}$
- For SDSC's Triton Cluster
 - ♦ $\alpha \approx 3.2 \mu s$, $\beta_{\infty} \approx 1.2 \text{ Gbytes/sec} \Rightarrow n_{1/2} \approx 3.6 \text{ KB}$
 - ♦ The actual value of $n_{1/2} \approx 20 \text{ KB}$
- Measurements from the Ring Program (available on Bang, stampede soon)

Length (Bytes)	Bandwidth (GB/sec)	Time (us)
1	0.31	3.247
2	0.62	3.219
4	1.24	3.216
8	2.47	3.244
16	4.91	3.258
32	8.3	3.855
64	15.81	4.047
128	25.28	5.062
256	48.25	5.305
512	86.25	5.936
1024	142.8	7.168
2048	209.3	9.786
4096	188.8	21.7
8192	334.7	24.48
16384	519.2	31.56
32768	718.6	45.6
65536	702.7	93.26
131072	897.1	146.1
262144	1039	252.4
524288	1124	466.4
1048576	1177	890.8
2097152	1201	1747
4194304	1216	3449
8388608	1223	6858

Short and intermediate message lengths



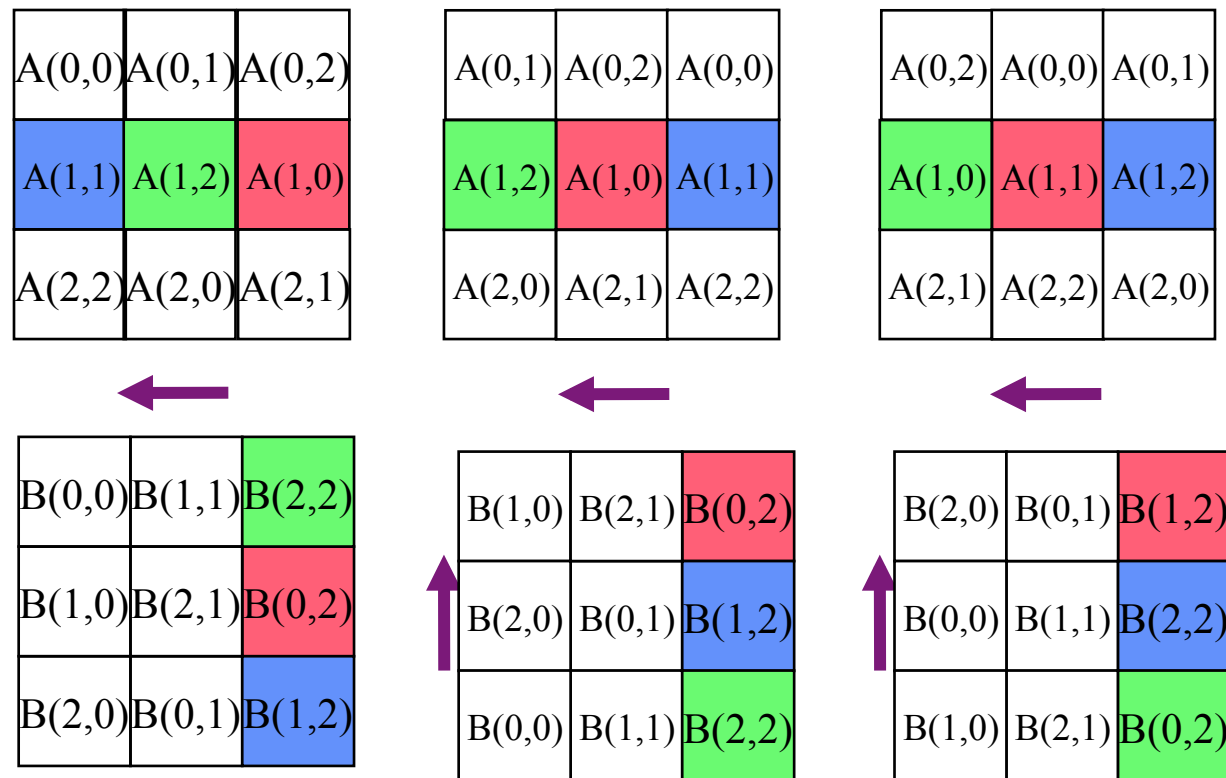
Today's lecture

- Performance metrics
- Performance Measurement
- Communication performance
- **Communication avoiding matrix multiplication**

Recalling Cannon's algorithm

- $\sqrt{p}$ shift and multiply-add steps
- Each processor forms the partial product of local A & B and adds into the accumulated sum in C

$$C[1,2] = A[1,0]*B[0,2] + A[1,1]*B[1,2] + A[1,2]*B[2,2]$$



Cost of Cannon's Algorithm

forall $i=0$ to $\sqrt{p}-1$

 CShift-left $A[i; :]$ by i

// $T = \alpha + \beta n^2/p$

forall $j=0$ to $\sqrt{p}-1$

 Cshift-up $B[:, j]$ by j

// $T = \alpha + \beta n^2/p$

for $k=0$ to $\sqrt{p}-1$

forall $i=0$ to $\sqrt{p}-1$ **and** $j=0$ to $\sqrt{p}-1$

$C[i,j] += A[i,j]*B[i,j]$

// $T = 2*n^3/p^{3/2}$

 CShift-left $A[i; :]$ by 1

// $T = \alpha + \beta n^2/p$

 Cshift-up $B[:, j]$ by 1

// $T = \alpha + \beta n^2/p$

end forall

end for

$$T_p = 2n^3/p + 2(\alpha(1+\sqrt{p}) + (\beta n^2)(1+\sqrt{p})/p)$$

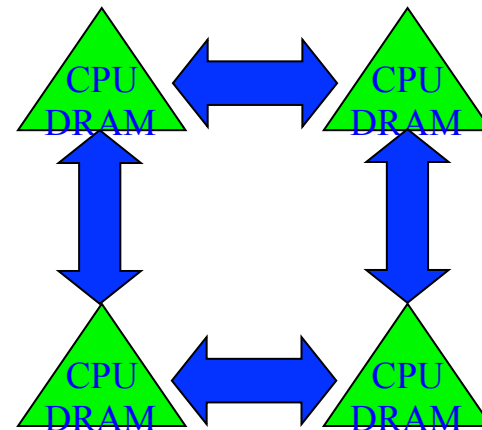
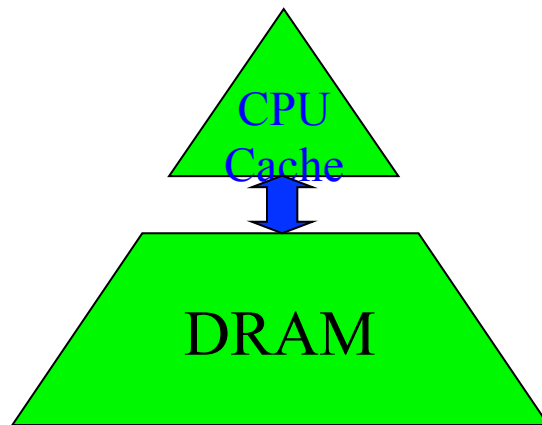
$$E_p = T_1/(pT_p) = (1 + \alpha p^{3/2}/n^3 + \beta \sqrt{p}/n)^{-1}$$

$$\approx (1 + O(\sqrt{p}/n))^{-1}$$

$$E_p \rightarrow 1 \text{ as } (n/\sqrt{p}) \text{ grows [sqrt of data / processor]}$$

Can we improve on Cannon's algorithm?

- Relative to arithmetic speeds, communication is becoming more costly with time
- Communication can be data motion on or off-chip, across address spaces
- We seek algorithms that increase the amount of work (flops) relative to data moved



Communication lower bound

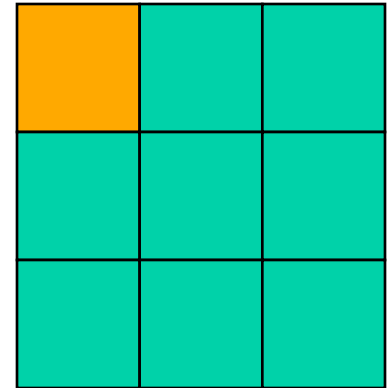
for Matrix Multiplication and other direct linear algebra

- Let M = Size of fast memory/processor, e.g. cache
- # words moved per processor
 $\Omega(\text{\#flops(per processor)} / \sqrt{M})$
- # messages sent per processor
 $\Omega(\text{\#flops(per processor)} / M^{3/2})$
- Consider dense matrix multiply
 - 1 copy of the data, $M \approx n^2 / P$
 - Lower bounds are $\Omega(n^2 / \sqrt{P})$ and $\Omega(\sqrt{P})$
- Realized by Cannon's algorithm

Canon's Algorithm - optimality

- General result
 - ♦ If each processor has M words of local memory ...
 - ♦ ... at least 1 processor must transmit $\Omega (\# \text{ flops} / M^{1/2})$ words of data
- If local memory $M = O(n^2/p)$...
 - ♦ at least 1 processor performs $f \geq n^3/p$ flops
 - ♦ ... lower bound on number of words transmitted by at least 1 processor

$$\begin{aligned}\Omega ((n^3/p) / \sqrt{(n^2/p)}) &= \Omega ((n^3/p) / \sqrt{M}) \\ &= \Omega (n^2 / \sqrt{p})\end{aligned}$$

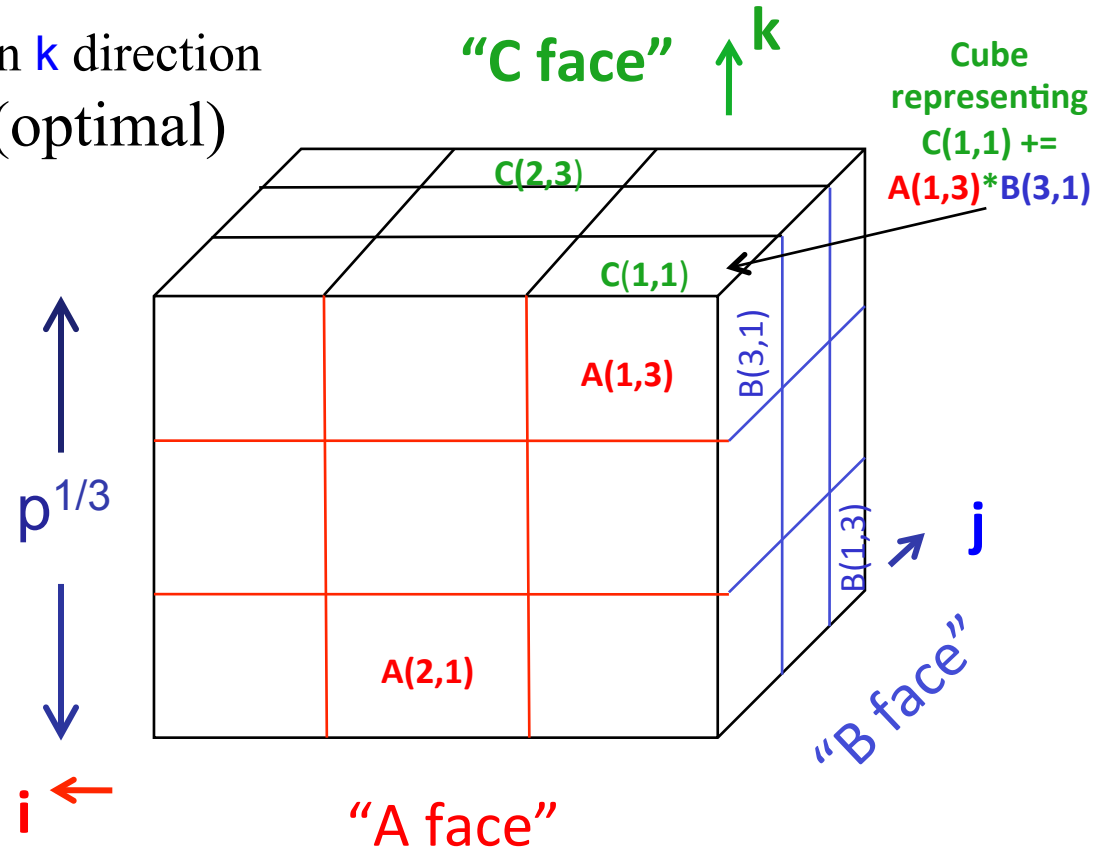
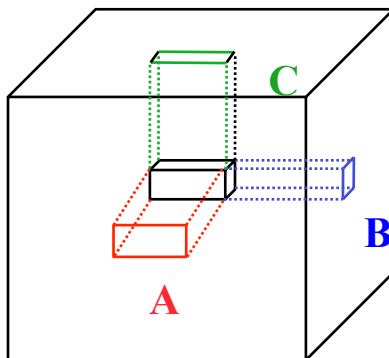


New communication lower bounds – direct linear algebra [Ballard & Demmel '11]

- Let M = amount of fast memory per processor
- Lower bounds
 - ♦ # words moved by at least 1 processor
 $\Omega(\# \text{ flops} / M^{1/2})$
 - ♦ # messages sent by at least 1 processor
 $\Omega(\# \text{ flops} / M^{3/2})$
- Holds not only for Matrix Multiply but many other “direct” algorithms in linear algebra, sparse matrices, some graph theoretic algorithms
- Identify 3 values of M
 - ♦ 2D (Cannon’s algorithm)
 - ♦ 3D (Johnson’s algorithm)
 - ♦ 2.5D (Ballard and Demmel)

Johnson's 3D Algorithm

- 3D processor grid: $p^{1/3} \times p^{1/3} \times p^{1/3}$
 - Bcast A (B) in j (i) direction ($p^{1/3}$ redundant copies)
 - Local multiplications
 - Accumulate (Reduce) in k direction
- Communication costs (optimal)
 - Volume = $O(n^2/p^{2/3})$
 - Messages = $O(\log(p))$
- Assumes space for $p^{1/3}$ redundant copies
- Trade memory for communication

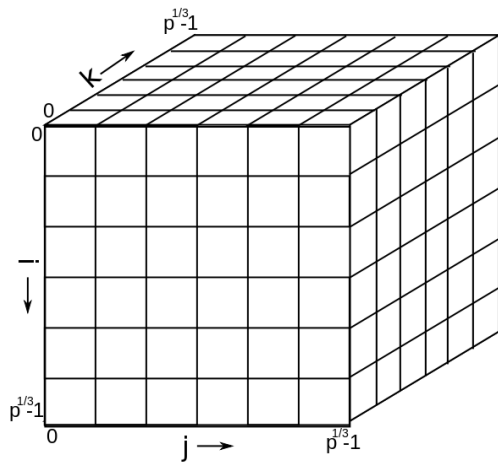


Source: Edgar Solomonik

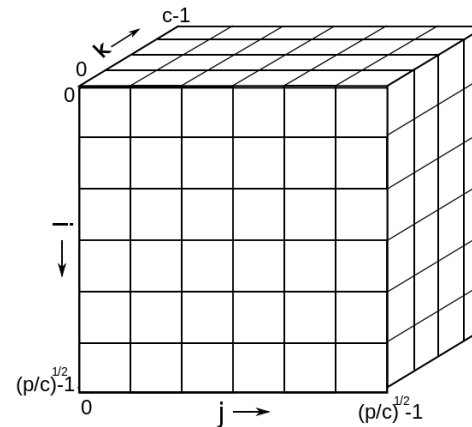
2.5D Algorithm

- What if we have space for only $1 \leq c \leq p^{1/3}$ copies ?
- P processor on a $(P/c)^{1/2} \times (P/c)^{1/2} \times c$ mesh $M = \Omega(c \cdot n^2/p)$
- Communication costs : lower bounds
 - ♦ Volume = $\Omega(n^2 / (cp)^{1/2})$; Set $M = c \cdot n^2/p$ in $\Omega(\# \text{ flops} / M^{1/2})$
 - ♦ Messages = $\Omega(p^{1/2} / c^{3/2})$; Set $M = c \cdot n^2/p$ in $\Omega(\# \text{ flops} / M^{3/2})$
 - ♦ sends $c^{1/2}$ times fewer words, $c^{3/2}$ times fewer messages
- 2.5D algorithm “interpolates” between 2D & 3D algorithms

3D



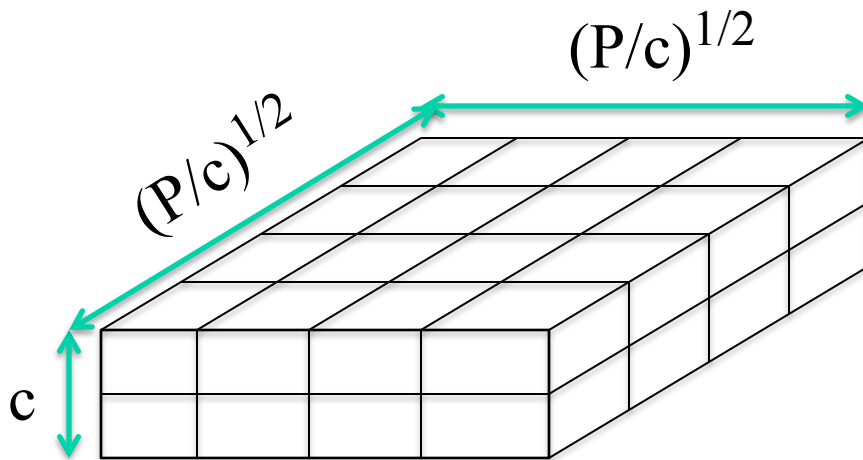
2.5D



Source: Edgar Solomonik

2.5D Algorithm

- Assume can fit cn^2/P data per processor, $c > 1$
- Processors form $(P/c)^{1/2} \times (P/c)^{1/2} \times c$ grid

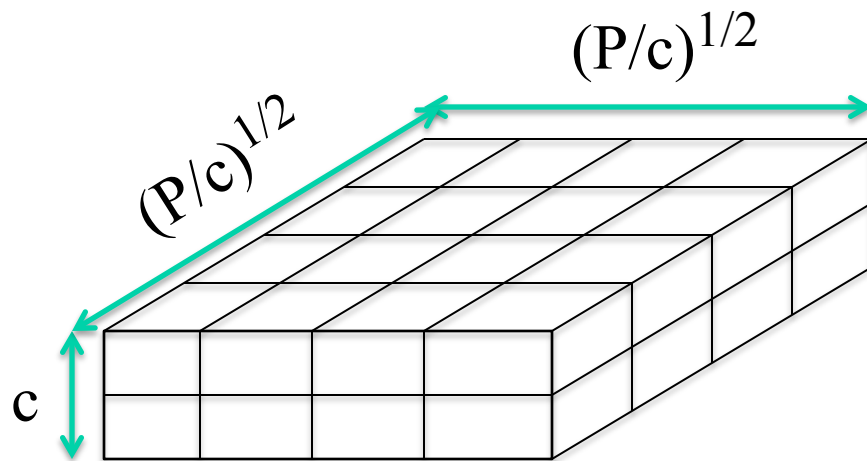


Example: $P = 32$, $c = 2$

Source Jim Demmel

2.5D Algorithm

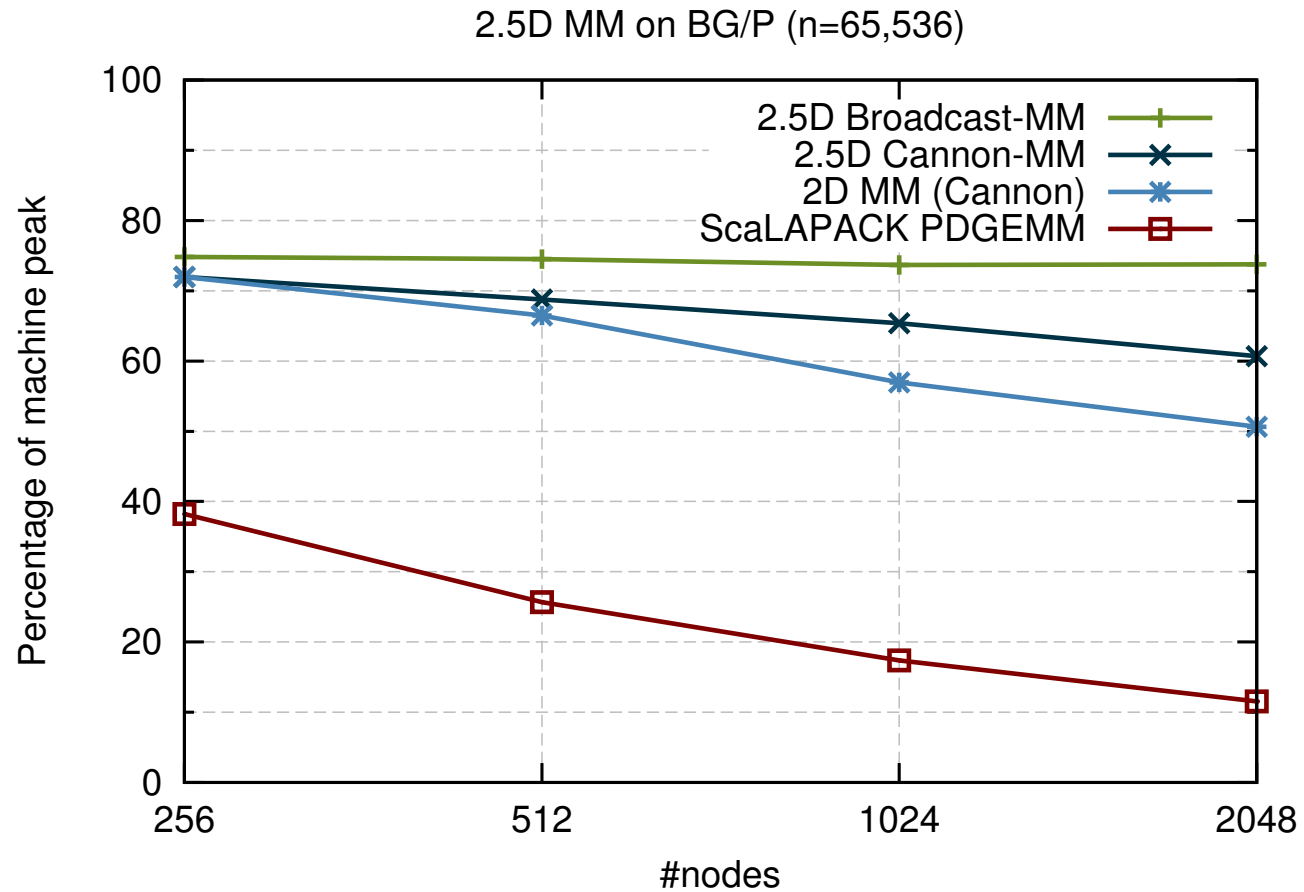
- Assume can fit cn^2/P data per processor, $c > 1$
- Processors form $(P/c)^{1/2} \times (P/c)^{1/2} \times c$ grid



Initially $P(i,j,0)$ owns $A(i,j)$ & $B(i,j)$
each of size $n(c/P)^{1/2} \times n(c/P)^{1/2}$

- (1) $P(i,j,0)$ broadcasts $A(i,j)$ and $B(i,j)$ to $P(i,j,k)$
- (2) Processors at level k perform $1/c$ -th of SUMMA,
i.e. $1/c$ -th of $\sum_m A(i,m) * B(m,j)$
- (3) Sum-reduce partial sums $\sum_m A(i,m) * B(m,j)$ along k -axis so that $P(i,j,0)$ owns $C(i,j)$

Performance on Blue Gene P



Implications for scaling (parallel case)

- To ensure that communication is not the bottleneck, we must balance the relationships among various performance attributes
 - ♦ $\gamma M^{1/2} > \approx \beta$: time to add two rows of locally stored square matrix $>$ reciprocal bandwidth
 - ♦ $\gamma M^{3/2} > \approx \alpha$: time to multiply 2 locally stored square matrices $>$ latency
- Machine parameters:
 - ♦ γ = seconds per flop (multiply or add)
 - ♦ β = reciprocal bandwidth (time)
 - ♦ α = latency (time)
 - ♦ M = local (fast) memory size
 - ♦ P = number of processors
- $\text{Time} = \gamma * \text{\#flops} + \beta * \text{\#flops}/M^{1/2} + \alpha * \text{\#flops}/M^{3/2}$

2.5D Algorithm

- Interpolate between 2D (Cannon) and 3D
 - ♦ c copies of A & B
 - ♦ Perform $p^{1/2}/c^{3/2}$ Cannon steps on each copy of A&B
 - ♦ Sum contributions to C over all c layers
- Communication costs (not quite **optimal**, but not far off)

- ♦ Volume:

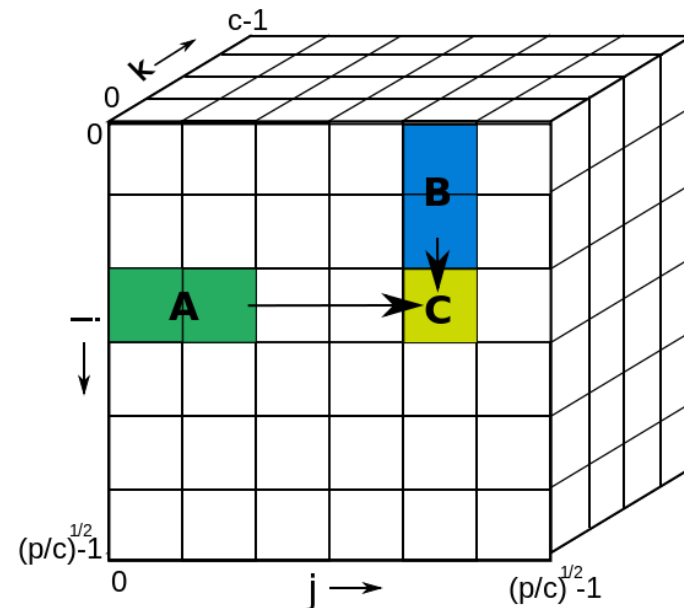
$$O(n^2 / (cp)^{1/2})$$

$$[\Omega(n^2 / (cp)^{1/2})]$$

- ♦ Messages:

$$O(p^{1/2} / c^{3/2} + \log(c))$$

$$[\Omega(p^{1/2} / c^{3/2})]$$



Source: Edgar Solomonik