Poor Parallel Program Performance?

Bothersome Bad Bugs?

Sloppy, Slow, Software?

Robert J. Harrison

Stony Brook University

When writing any program

- Correctness is the primary concern
  - Parallel programs have creative ways of going wrong

- Why are you writing a parallel program?
  - To use the resources (memory, compute power, ...) of the parallel computer to solve a big problem fast

- For parallel programs, performance is a correctness issue

Getting it right the first time – Design

- What are your objectives?
  - What sort of parallel computer?
  - How big a problem?
  - How fast a solution?
  - How many processors?
  - How portable?

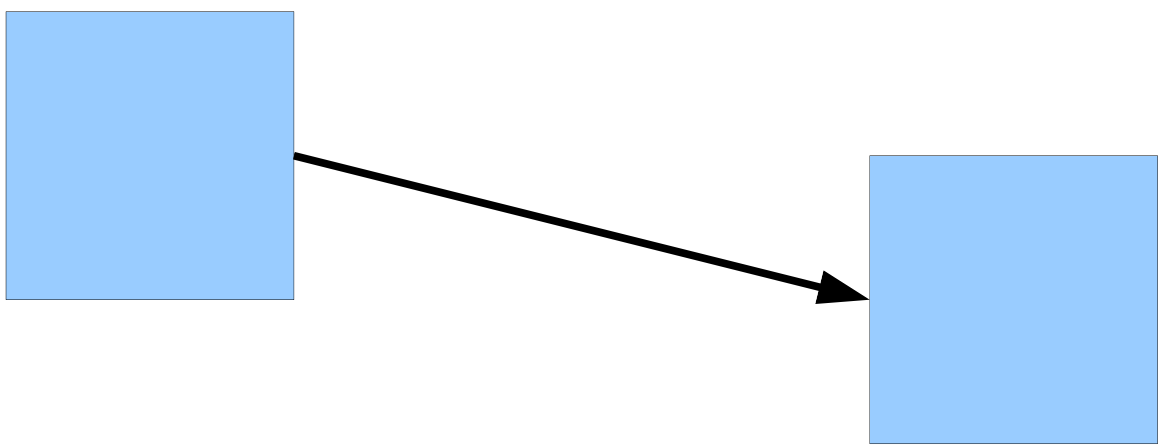
- You can reason about program performance by constructing a performance model

If you have an infinitely fast processor ...

- How fast can you compute?
  - As fast as you can get data to it
- Data motion is more often the bottleneck than how fast computation actually happens
  - FLOPs are free
  - Bandwidth is expensive (power and price)

Latency and bandwidth

- Latency – startup time
  - Cost of operating on zero elements
- Bandwidth – steady state #elements / unit time



$$t(n) = t_0 + n/B$$

$$rate = \frac{n}{t} = \frac{B}{B\,t_0/n + 1} \quad \Rightarrow \quad n_{1/2} = B\,t_0$$

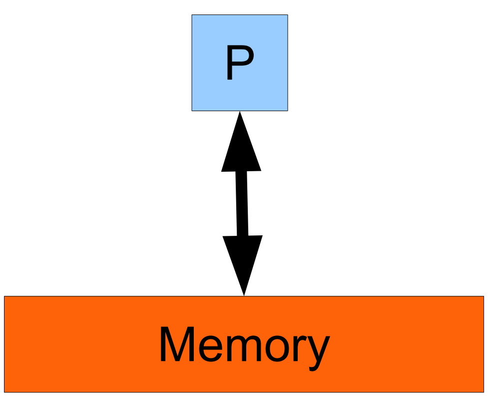
Matrix multiplication on a simple sequential computer - I

```
for i=1 to N
    for j=1 to N
        sum = 0.0
        for k=1 to N
            sum += A[i,k] * B[k,j]
        C[i,j] = sum
```

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

$2N^3$ FLOPs

$2N^3$ memory loads



Compute
8 GFLOP/s
Memory bandwidth
1 GWord/s
Memory latency
0.2 to 2us

A read with unit stride
B read with stride N

Disaster!

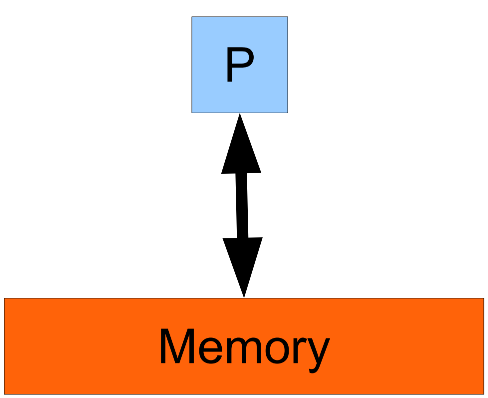
Matrix multiplication on a simple sequential computer - II

```
for i=1 to N
    for k=1 to N
        aik = A[i,k]
        for j=1 to N
            C[i,j] += aik * B[k,j]
```

$$c_{i\,j} = \sum_{k=1}^{N} a_{i\,k} b_{k\,j}$$

$$2N^3 \quad \text{FLOPs}$$

$$3N^3 \quad \text{memory load} + \text{store}$$



Compute
8 GFLOP/s
Memory bandwidth
1 GWord/s
Memory latency
0.2 to 2us

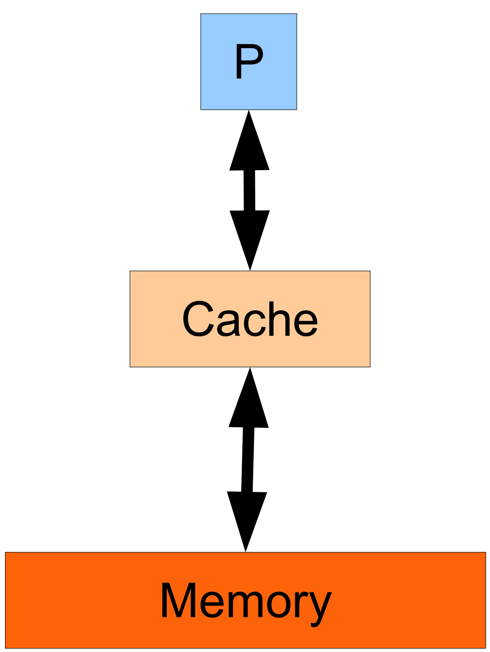
B read with unit stride
C written with unit stride

12.5% of peak speed

Matrix multiplication on a simple sequential computer - III

```
for i=1 to N
    for k=1 to N
        aik = A[i,k]
        for j=1 to N
            C[i,j] += aik * B[k,j]
```

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$



Compute
 8 GFLOP/s
Cache bandwidth
 8 GWord/s
Cache latency
 0.5ns
Memory bandwidth
 1 GWord/s
Memory latency
 0.2 to 2us

$$2N^3 \quad \text{FLOPs}$$

$$3N^3 \quad \text{memory load} + \text{store}$$

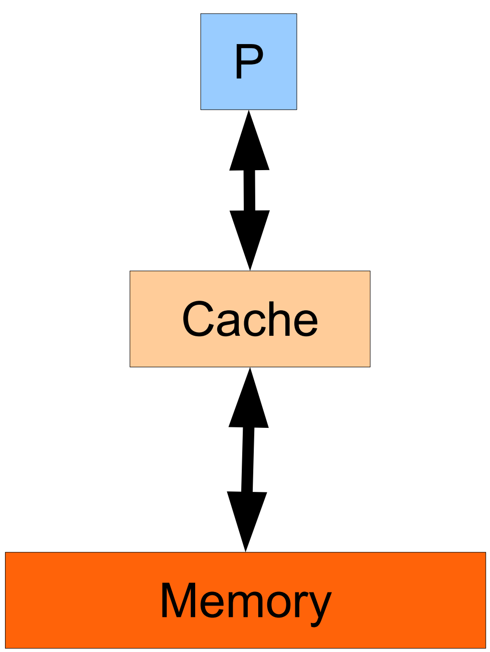
B read with unit stride
C written with unit stride

66% speed ... iff B&C small enough to fit in cache

Matrix multiplication on a simple sequential computer - IV

```
for tiles of i
    for tiles of j
        for k=1 to N
            for i in tile
                aik = A[i,k]
                for j in tile
                    C[i,j] += aik*B[k,j]
```

$$c_{i\,j} = \sum_{k=1}^{N} a_{i\,k} b_{k\,j}$$



Choose tile size T so that 2*T*T fits into cache
Usually must do this empirically

Memory motion and compute take the same time

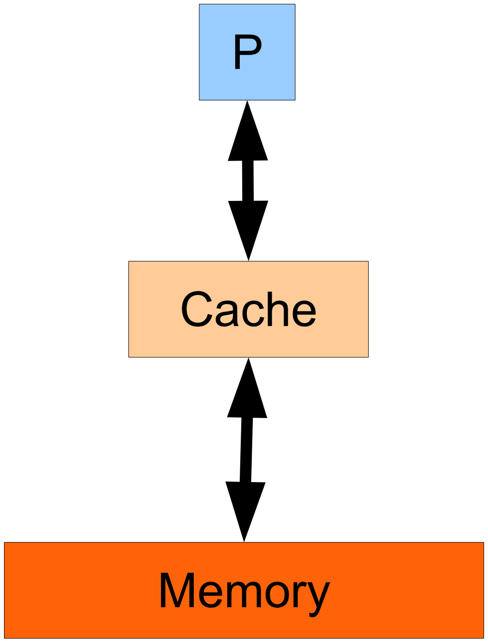
Assuming compute and memory motion can be overlapped we are now running at 66% peak speed

To get peak speed must keep C in registers

Matrix multiplication on a simple sequential computer - V

```
for tiles of i
    for tiles of j
        for k=1 to N
            for i in tile
                aik = A[i,k]
                load C[i,*]into R[*]
                for j in tile
                    R[j] += aik*B[k,j]
                store R[*] into C[i,*]
```

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$



Now we are running at full speed

Depressing conclusion is that we must tile for every level of the memory hierarchy

This applies equally to distributed memory computers

Parallel Algorithms

- All of the complexity of sequential algorithms
  - Plus ...
- Concurrency
  - Gotta keep everyone equally busy
- Additional levels in the memory hierarchy
  - Cache and memory of other processors
- Contention
  - Literally processes fighting over wires

Weak and strong scaling

- Strong scaling
  - Fixed problem size as increase number of processors

- Weak scaling
  - Increase the problem size as increase number of processes

Speedup, efficiency, iso-efficiency

$$S(P) = \frac{t(1)}{t(P)} \qquad S_{\text{ideal}}(P) = P$$

$$\epsilon(P) = \frac{S(P)}{P} = \frac{t(1)}{P\,t(P)} \leq 1 \qquad \epsilon_{\text{ideal}}(P) = 1$$

$$t(P, N, \epsilon) =$$

isoefficiency : running time of problem size N required to get specified efficiency on P processors

... for perfect weak scaling is a constant

... for perfect strong scaling is a decreasing function of P

Amdahl's Law

- Assume

$$t(P) = t_{seq} + \frac{t_{par}}{P}$$

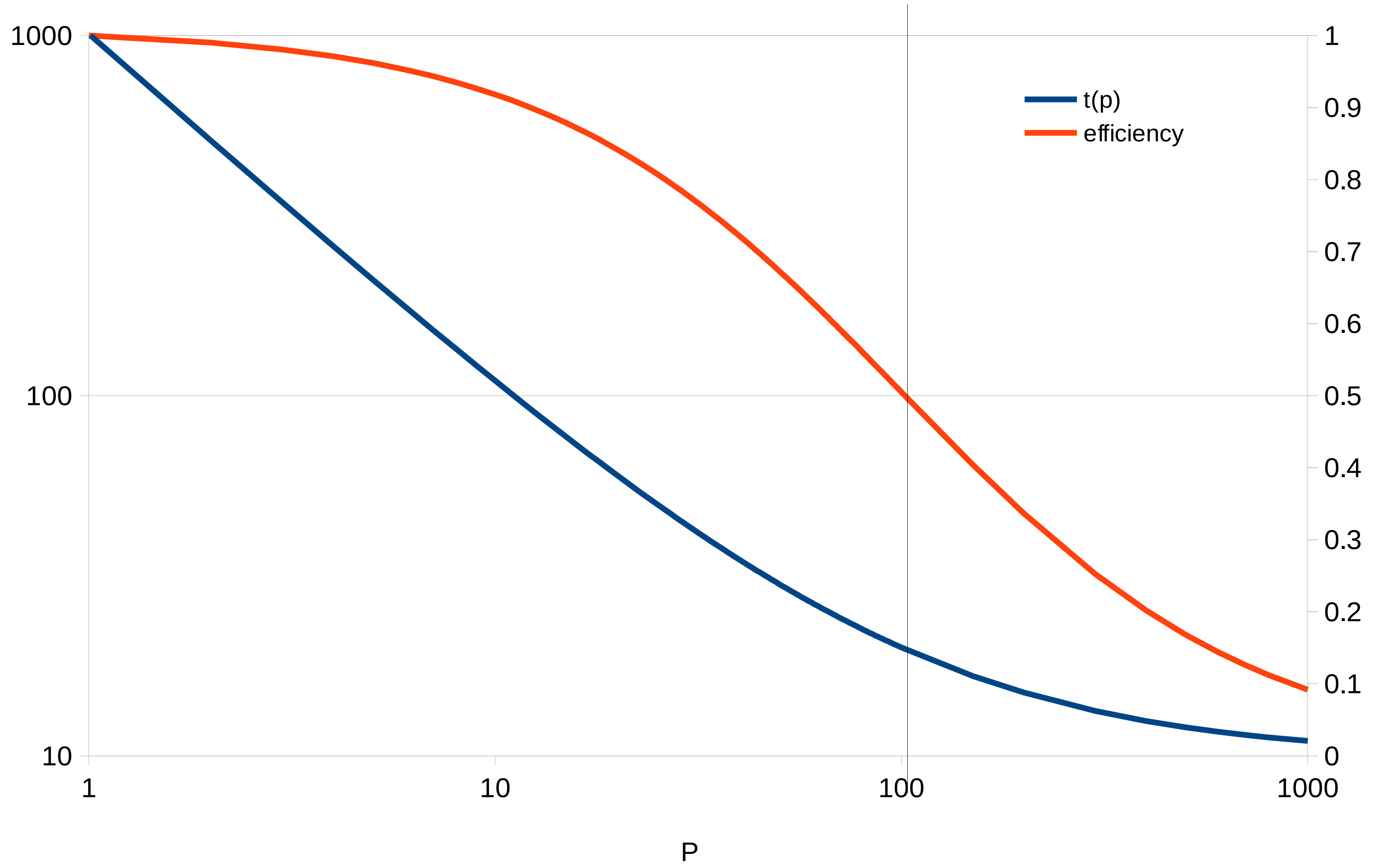
  - Sequential component (or equivalently a part that each process must repeat)
  - Perfectly parallel component
- Maximum speed up is

$$S(P) = \frac{t(0)}{t(P)} = \frac{t_{seq} + t_{par}}{t_{seq} + \frac{t_{par}}{P}}$$

$$S(\infty) \rightarrow \frac{t_{par}}{t_{seq}}$$

50% efficiency when $P = \dfrac{t_{par}}{t_{seq}}$

Amdahl's law in action



$$t(p) = 10 + \frac{1000}{P}$$

Contention

- Multiple processes trying to
  - Read the same memory location or bank
  - Send a message over the same wire
  - Compute on the same processor
- Execution is serialized
  - Equivalent to reducing execution rate
  - E.g., P processes trying simultaneously to send data over a link of bandwidth B
    - Each process sees bandwidth B/P

New failure modes

- Race conditions
  - Read before/after write
  - Concept of atomicity
- Dead/live lock
- Hangs
- Exhausting buffer space
- Oversubscribed resources
  - Exponential slowdown

Safe programming models

- Some models are designed to eliminate some types of errors
  - Communicating sequential processes (CSP, Hoare)
  - MPI safe message passing
  - Bulk synchronous program (BSP)
  - OpenMP data parallel