TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Introduction to Parallel Computing

Lars Koesterke & Charlie Dey

PCSE

Jan 18, 2018

Why parallel computing?

Assume you have 2 computers

Why/when would you use both of them instead of one?

Benefits?

How to do parallel computing?

If you are using 2 computers, what do you have to do?

Overview

Architectures and Programming Models

Levels of Parallelism

Practical and Theoretical Performance Limits

Other Issues and Challenges

Summary

Architectures and Programming Models

Levels of Parallelism

Practical and Theoretical Performance Limits

Other Issues and Challenges

Summary

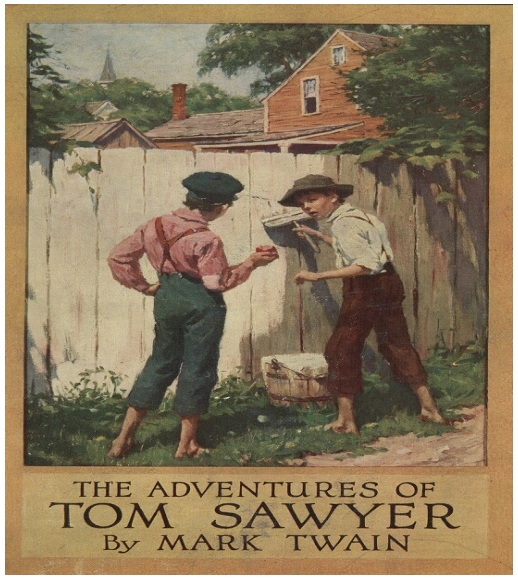
What is Parallel Programming?

More than one paint brush!

Paint the fence faster…

…or paint a bigger fence

Paint brushes = cores



The adventures of Tom Sawyer, by Mark Twain [pseud.]
illustrated by Worth Brehm. Adventures of Tom Sawyer. 1910.
In the public domain. From Beinecke Rare Book & Manuscript Library.
http://brbl-dl.library.yale.edu/vufind/Record/3520172?image_id=1010069

What is Parallel Programming?

More than one mower!

Mow the lawn faster…

…or mow a bigger lawn

Lawn mowers = cores

Brett Chisum 2012 (Augusta National)
Wikipedia Commons
http://www.flickr.com/photos/brettchisum/7051114207

Shared Memory

Wikipedia Commons
http://commons.wikimedia.org/wiki/File:Legpuzzel.JPG

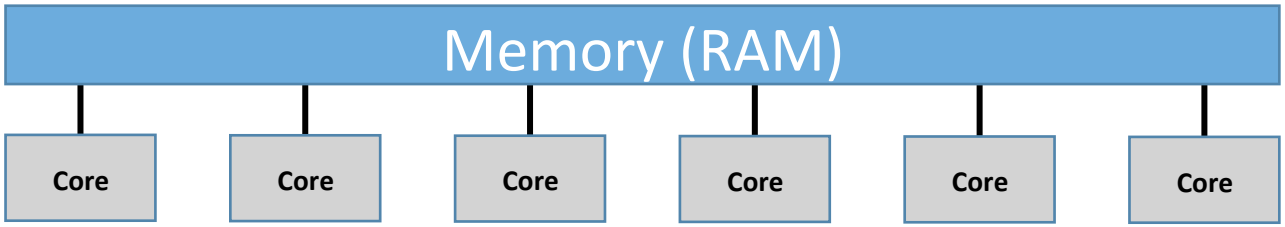
All cores share a common pool of memory (RAM)

The programming challenge is coordination:

- how to avoid competing for access to the same puzzle pieces (in the memory)

Principal programming model: OpenMP

A single executable spawns independent threads and manages threads' access to data



Distributed Memory

Each core* has its own memory (RAM), inaccessible to others

The programming challenge is communication:

- how to share puzzle pieces (send/receive data)

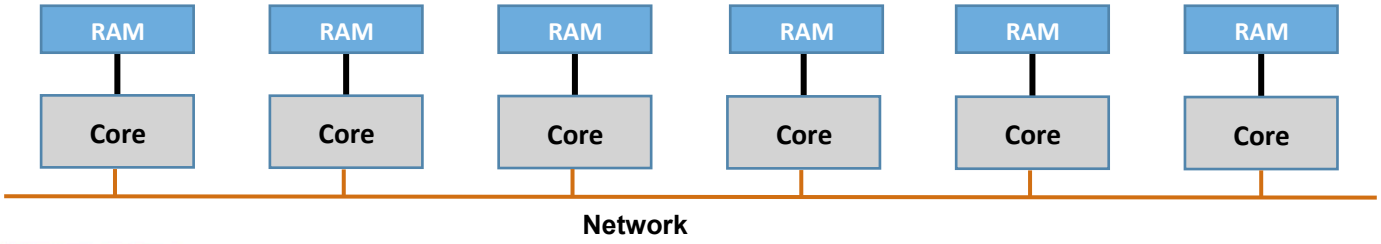
Principal programming model: MPI (Message Passing Interface)

Every assigned core runs a separate copy of the same executable

- a “rank aware” process (also called a task)

Octahedron80 2007 Wikipedia Commons
http://commons.wikimedia.org/wiki/
File:Jigsaw_pieces_with_border.jpg

*we will modify this soon

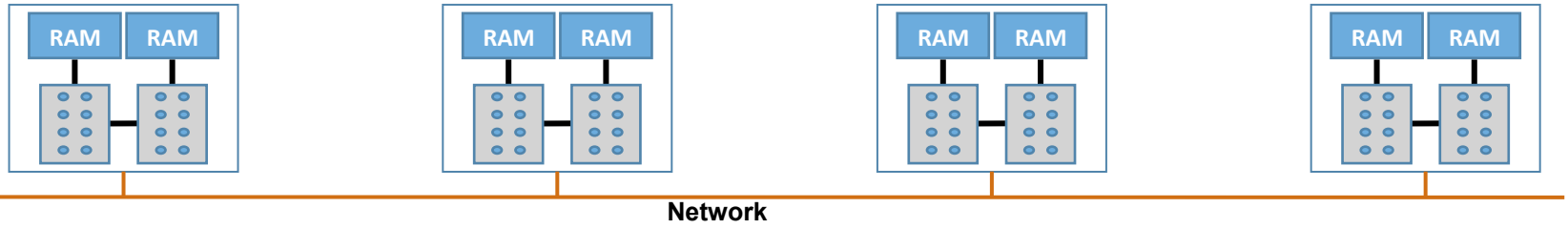


Hybrid Architecture

Most large clusters are hybrids of these models:

- Each node (blade) is a multi-core shared memory computer running its own (Linux) operating system. (Below: multiple “sockets”/node.)
- Many such nodes connected in distributed configuration
- Each core sees only the memory on its own node!

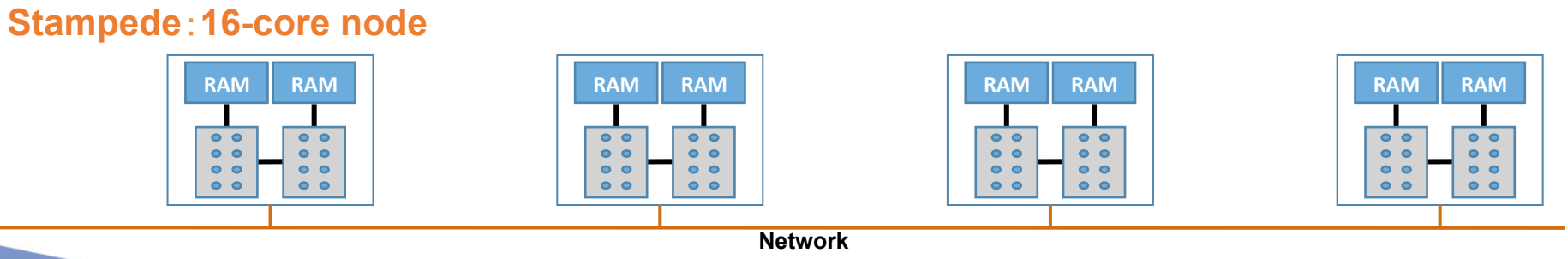
Stampede : 16-core node



Programming Hybrid Architectures

Programming models vary

- Pure OpenMP: confine yourself to one node
- Pure MPI: ignore shared memory
- Hybrid: mix MPI and OpenMP

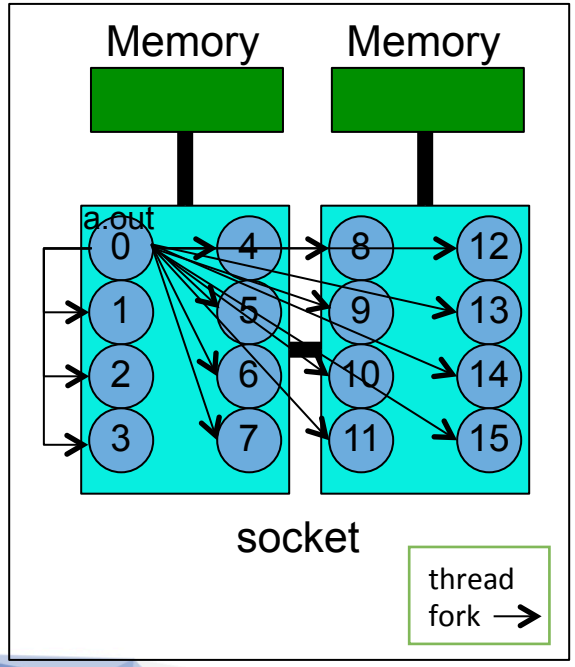


Programming Hybrid Architectures

Pure OpenMP

./a.out

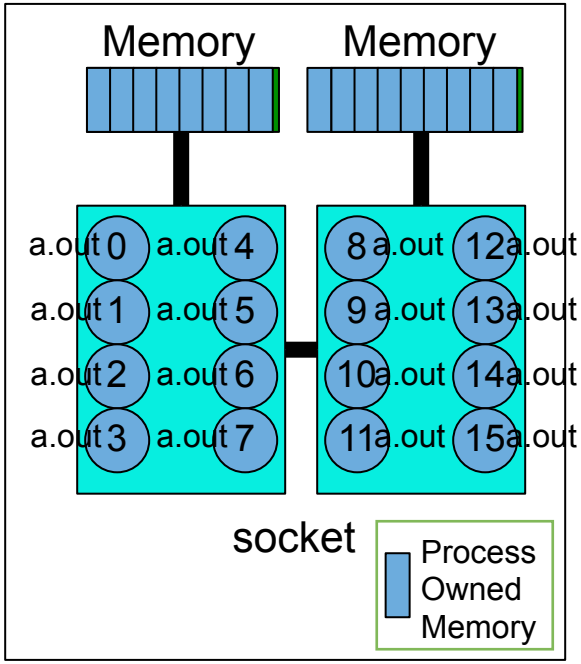
launch 1 a.out



Pure MPI

mpirun a.out

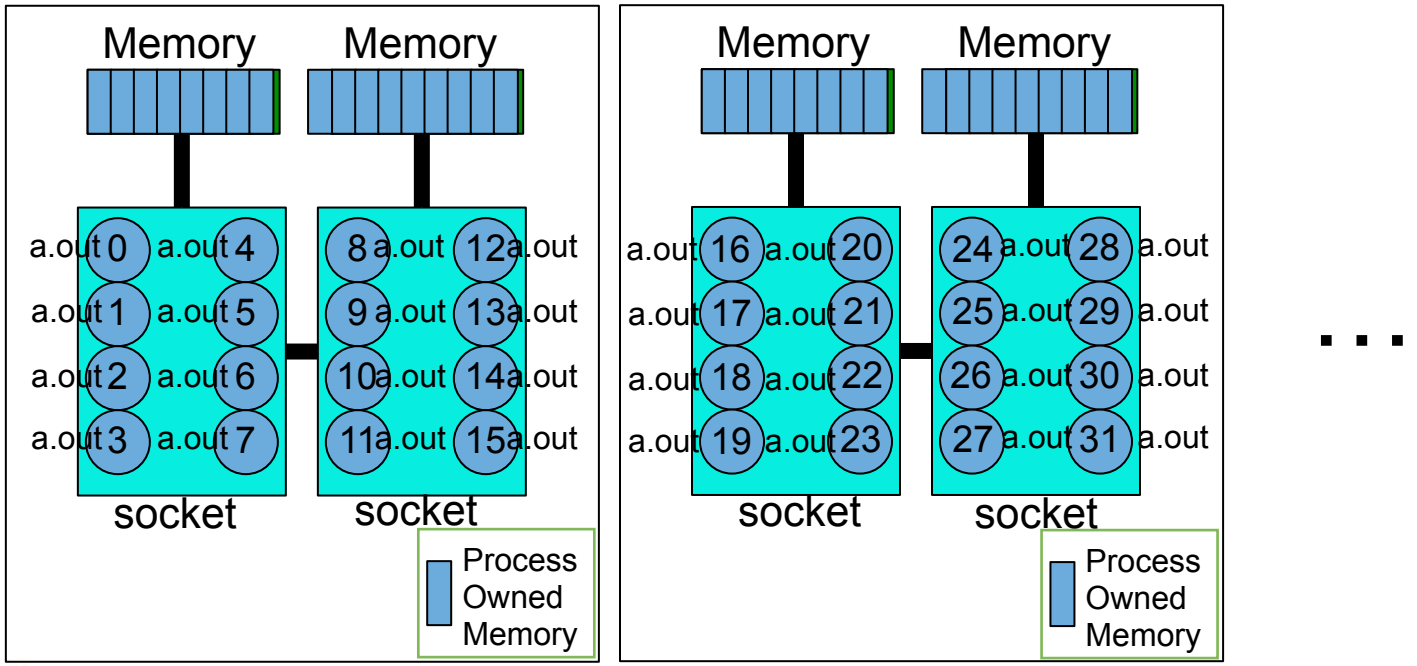
launch 16 a.out's



Multi-node Pure MPI

ibrun a.out
mpirun a.out

launch 16xN
a.out's



...

Why Do Parallel Computing?

Limits of single CPU computing

- processor speed, maximum performance
- total memory
- limited I/O capability

Parallel computing allows one to:

- solve problems that don't fit on a single CPU
- solve problems that can't be solved in a reasonable time

So we can solve…

- faster
- larger problems
- more cases

Architectures and Programming Models

Levels of Parallelism

Practical and Theoretical Performance Limits

Other Issues and Challenges

Summary

Needle(s) in the Haystack(s)

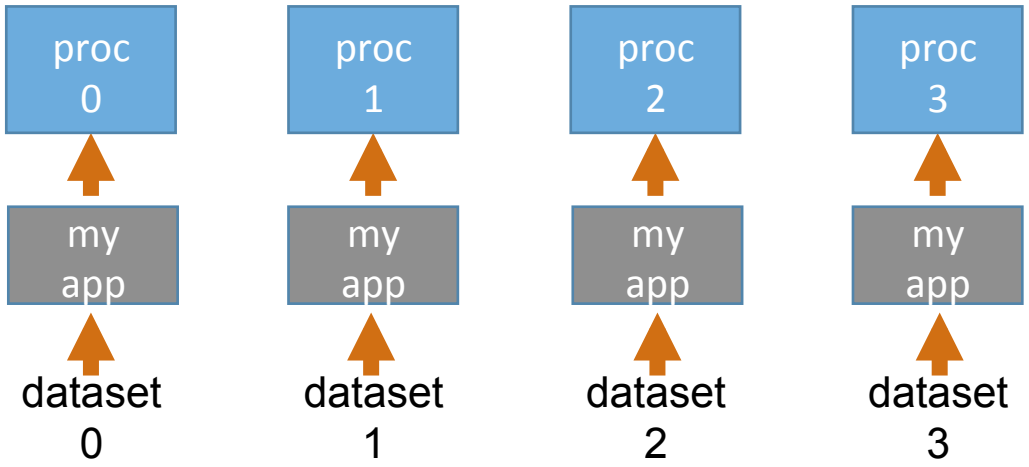
First approach: think top-down and coarse-grained

Partition the work into essentially independent tasks

Paul Allison 2007
http://www.geograph.org.uk/photo/602033
Wikipedia Commons
http://commons.wikimedia.org/wiki/
File:Hay_Bales_-_geograph.org.uk_-_602033.jpg

P.N.Alhucemas (Oruteta) 2009
Wikipedia Commons
http://commons.wikimedia.org/wiki/File:Almiar_(1).JPG

Coarse-Grained Parallelism (task-based)



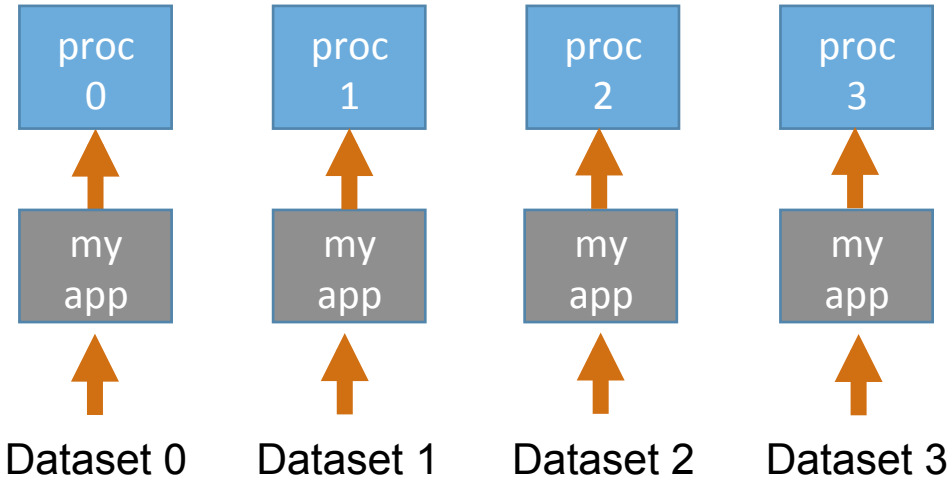
Assign tasks (e.g. MPI processes) to processors (nodes, cores, …)

The same code operates on different data

Logic within the program may differ across processors

How much communication, coordination, synchronization?

Massive (Embarrassing) Parallelism



High degree of independence

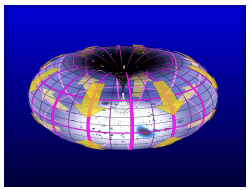
Important example: parameter sweeps

Little to no coordination, communication

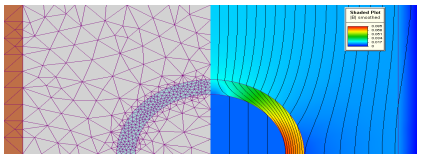
Domain Decomposition

Key issues:

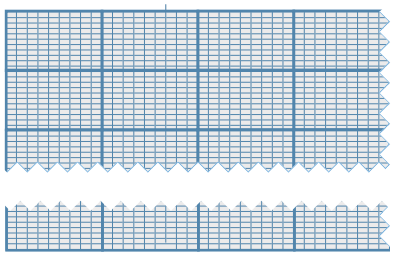
- Dependencies across ghost (halo/transition/boundary) regions
- Communication
- Load balancing
- More



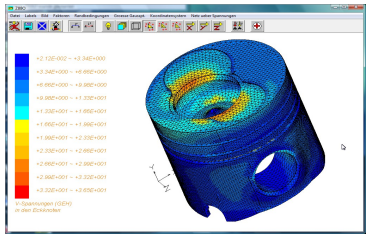
BryanBrandenburg.net 2012
Wikipedia Commons
http://commons.wikimedia.org/wiki/File:Bryan_Brandenburg_Big_Bang_Big_Bagel_Theory_Howard_Boom.jpg



Ethan Hein 2008
http://www.flickr.com/photos/ethanhein/2352707753/

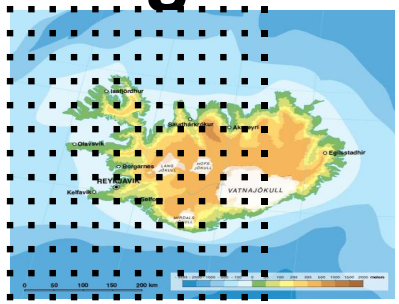


Doug James 2013



Bal 79 on Wikipedia Commons 2008
http://commons.wikimedia.org/wiki/File:Z88v13_1.jpg

Need: Weather Modeling and Forecasting



For modeling a hurricane region:

- Region of interest is 1000 X 1000 miles, with height of 10 miles.
- Partition into segments of 0.1 x 0.1 x 0.1 miles: 10^10 grid points
- Simulate 2 days, with 30-minute time steps: 100 total time steps

Assume the computations at each grid point require 100 instructions.

- A single time step then requires 10^12 instructions.
- For two days we need 10^14 instructions
- For serial computer with 10^8 instructions/sec, this takes 10^6 seconds (10 days!) to predict next 2 days!!

THIS REQUIRES PARALLELISM FOR PERFORMANCE TO PREDICT

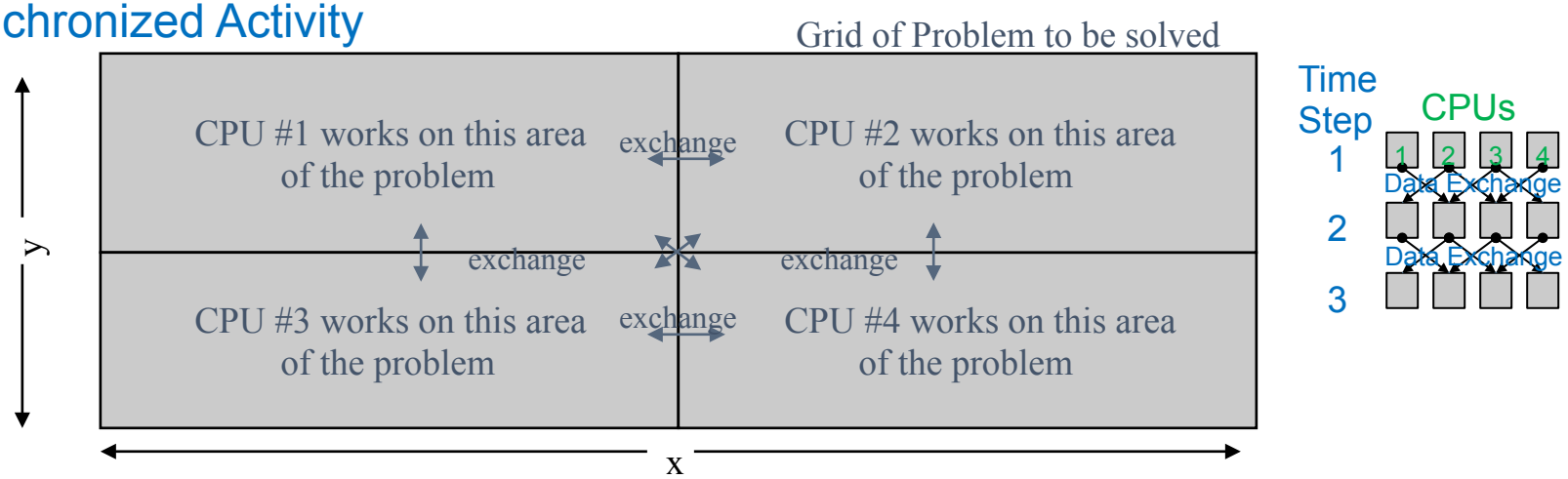
- Also requires lots of memory which implies parallelism

The major weather forecast centers (US, Europe, Asia) have supercomputers with 1000s of processors.

Working with a Decomposed Domain

Parallel computing: using multiple processors or computers to work together on a common task. It involves:

- Concurrent Execution of Instructions
- Dependencies (realized as data production or movement)
- Synchronized Activity

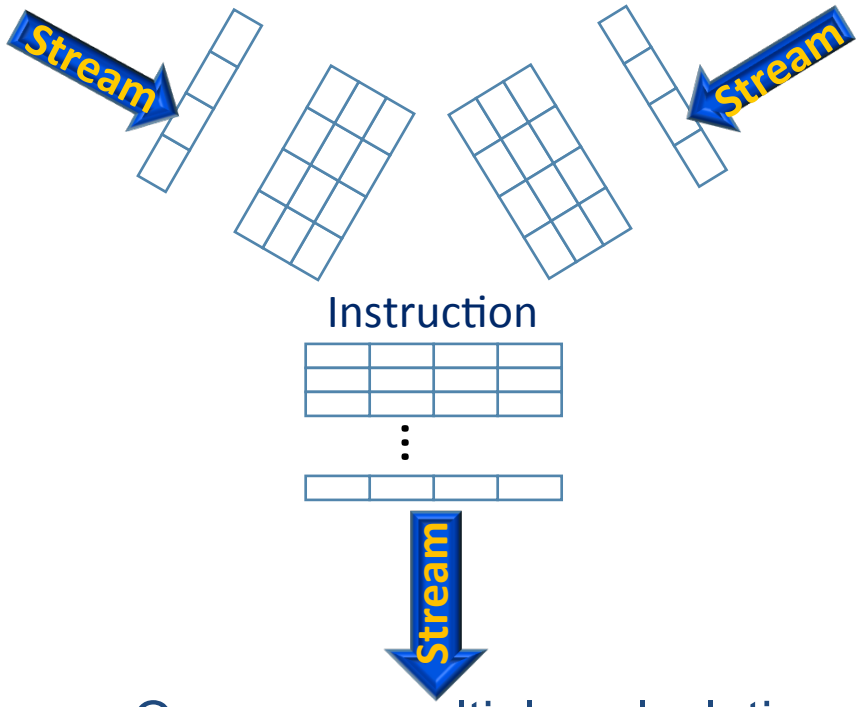


- Parallel Activity exists at many levels.

Fine-Grained Parallelism: Vectorization

C. Holmes 2009
Wikipedia Commons
http://www.flickr.com/photos/inventorchris2/7723117886/

One combine,
multiple rows of wheat



One core, multiple calculations

Fine-Grained Parallelism: Vectorization

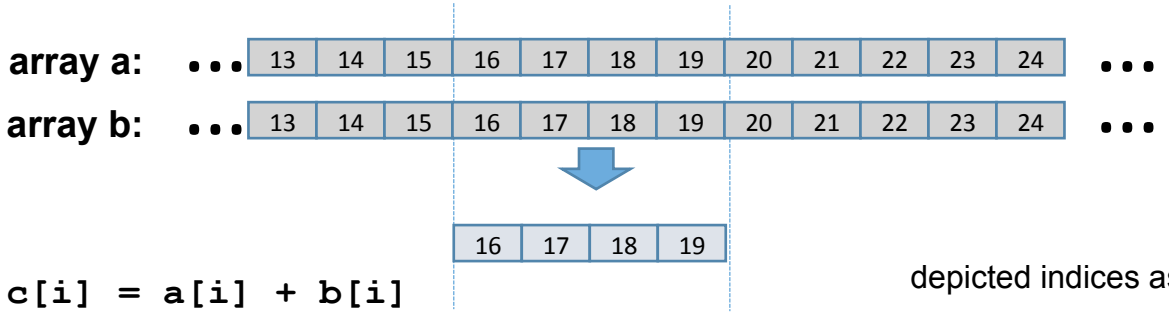
Tight, long inner loops with a few familiar array calculations:

```
/* C-style loop */

for (int i=0; i<n; ++i )
    c[i] = a[i] + b[i];


!Fortran arrays
c = a + b

!or Fortran loop
do i=1, n
  c(i) = a(i) + b(i)
enddo
```

Vectorization

Vector units (Stampede2)

- Each core has two 512-bit vector units
- Support for FMA: floating point 'multiply and add'
- Each unit can produce up to 16 double precision (DP) results/cycle
- Together they can produce up to 32 DP (64 SP) results/cycle



depicted indices assume 0-based aligned arrays

Medium-Grained Parallelism

```
...
compute gradients
solve for momentum
compute fluxes
solve for pressure correction
update pressure field
...
```

Your algorithm undoubtedly consists of a number of steps…

Some of those steps include loops that may be candidates for parallel execution…

```
for each cell i
  findCritPoints( i )
next i
```

Medium-Grained Parallelism

```
...

compute gradients

solve for momentum

compute fluxes

solve for pressure correction

update pressure field

...
```

One possible approach:

assign **OpenMP threads** to loop iterations (sometimes as simple as adding a one-line directive)…

```
for each cell i
  findCritPoints( i )
next i
```

Multi-Level Parallelism

It is not the only way to complete the work

Does not necessarily require hybrid MPI-OpenMP

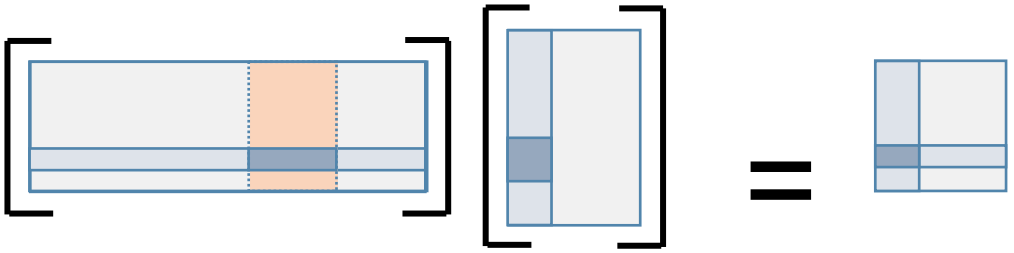
But a common hybrid approach looks like this:

- Coarse-grained parallelism across nodes via MPI (tasks manage blocks, slabs, sub-domains)
- Medium-grained parallelism on the nodes via OpenMP (threads manage middle loops over mesh points, slices, elements, particles)
- Vectorization for the tight inner loops within each thread

Linear Algebra Kernels

Off-the-shelf optimization and parallelism from mature, robust libraries; e.g.

- Intel Math Kernel Library (MKL) – robust support for Skylake and KNL
- PETSc – dense and sparse object-oriented solvers
- Lots of others (I'm probably skipping your favorite…)



Architectures and Programming Models

Levels of Parallelism

Practical and Theoretical Performance Limits

Other Issues and Challenges

Summary

Measuring Scalability (strong scalability)

The problem size stays fixed, but the number of processing elements are increased.

- Speedup on p processors:

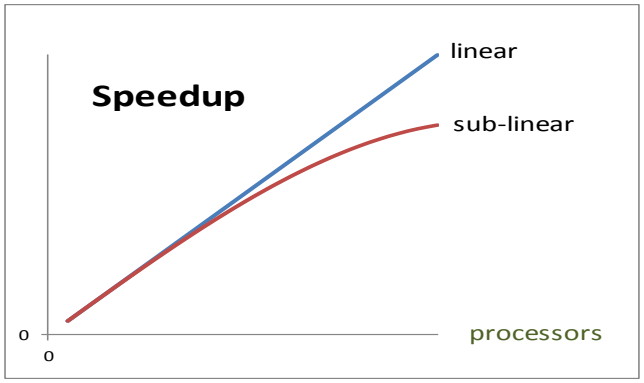
$$S_p = \frac{Time\ on\ one\ processor}{Time\ on\ p\ processors}$$

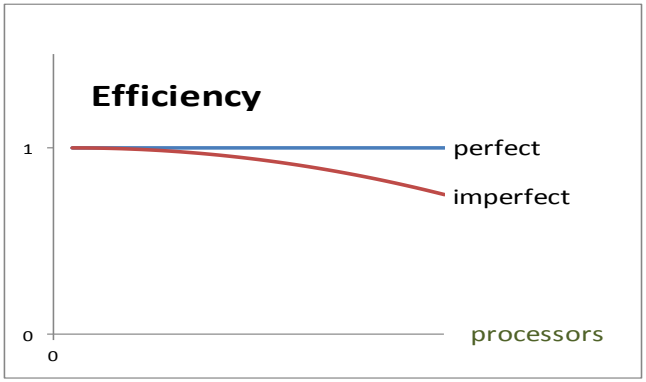
Sp=p is perfect (ideal)

- Efficiency on p processors:

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

Ep=1 is perfect (ideal)





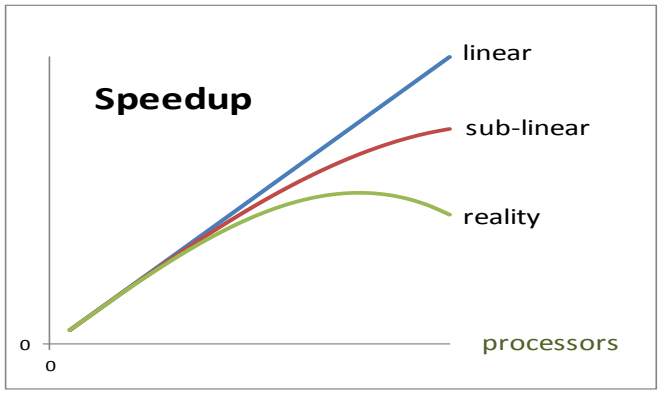
Measuring Scalability (strong scalability)

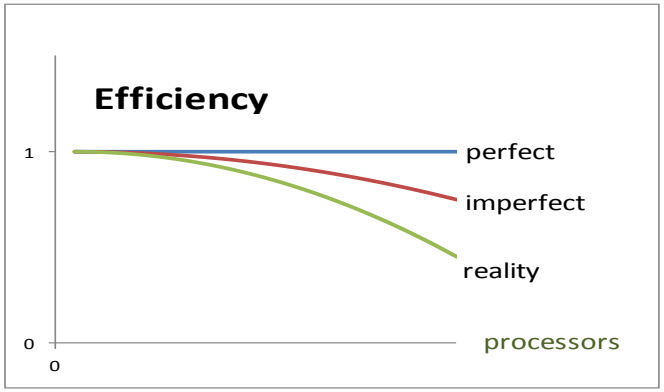
Reality sets in:

- Expect to lag behind perfection
- Expect a sweet spot beyond which adding processors will make things worse

There are several reasons:

- Various kinds of overhead
- Communication costs
- Load imbalance
- More





How fast can we bake a cake?

Assume (with no claim of realism)…

- Two hours to prepare the cake for baking
- Another half hour to bake the cake

So total time is…

```
Prep: 120 minutes
Bake:  30 minutes
Total: 150 minutes
```

How fast can we bake a cake?

But what if we have two cooks?

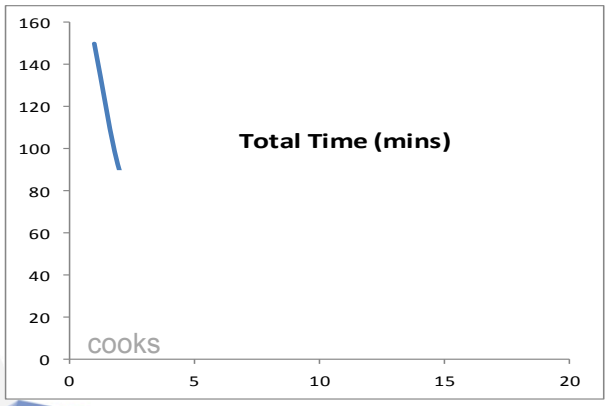
- Assume they work perfectly together
- No wasted time, no overhead

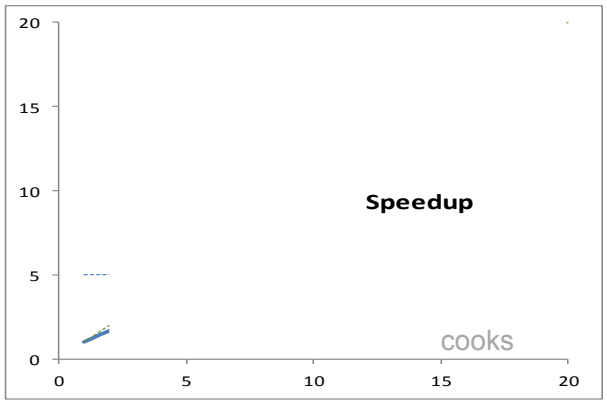
Then total time is…

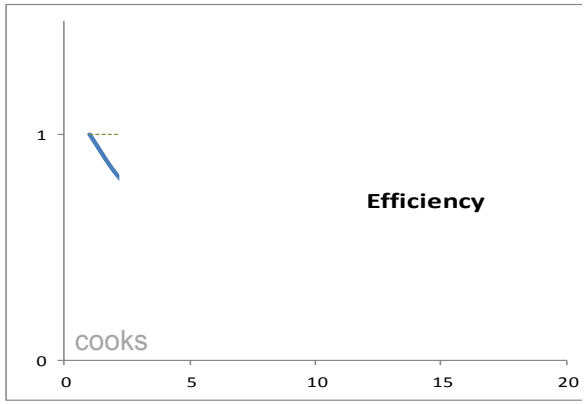
```
Prep: 120/2 = 60 minutes
Bake:         30 minutes
Total:        90 minutes
```

How fast can we bake a cake?

cooks	prep*	bake	total	speedup	efficiency
1	120/1 = 120.0	30.0	150.0	1.0	1.0
2	120/2 = 60.0	30.0	90.0	1.7	0.8

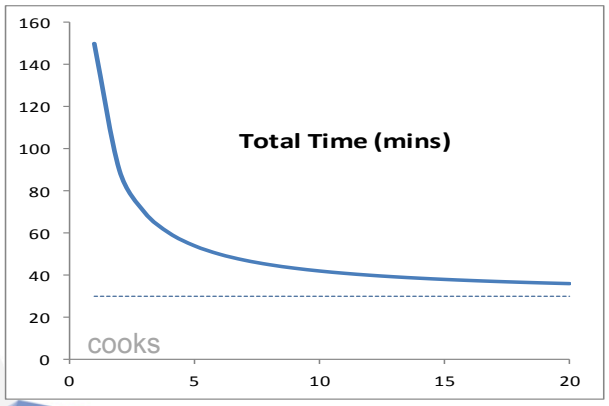


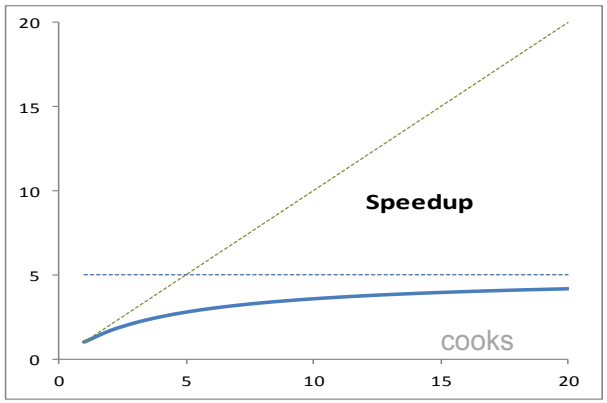


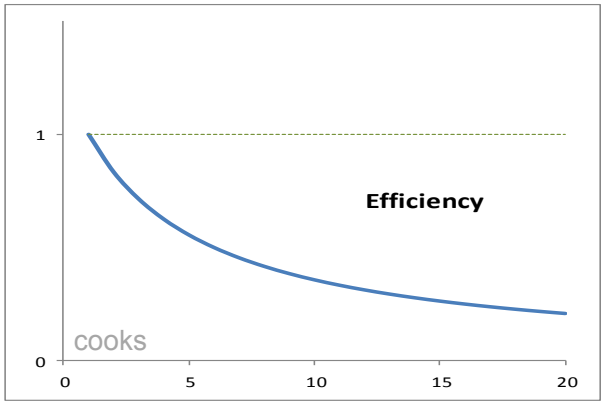


How fast can we bake a cake?

cooks	prep*	bake	total	speedup	efficiency
1	120/1 = 120.0	30.0	150.0	1.0	1.0
2	120/2 = 60.0	30.0	90.0	1.7	0.8
3	120/3 = 40.0	30.0	70.0	2.1	0.7
4	120/4 = 30.0	30.0	60.0	2.5	0.6
5	120/5 = 24.0	30.0	54.0	2.8	0.6
19	120/19 = 6.3	30.0	36.3	4.1	0.2
20	120/20 = 6.0	30.0	36.0	4.2	0.2







Amdahl's Law: Strong Scaling

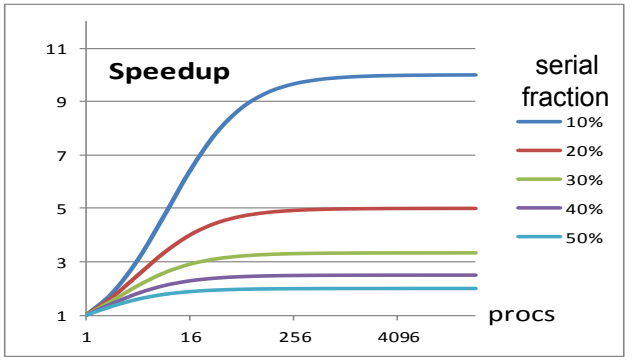
Scalability bounds: through Speedup

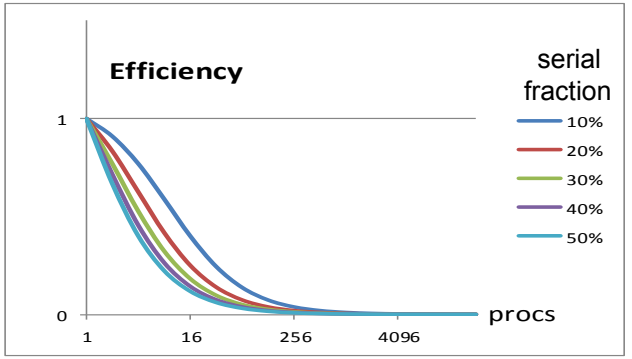
Same work, increase processor count.

$$t_n = (f_s + f_p / p) t_1 \quad \rightarrow \quad t_1/t_n = S$$

$$S = \frac{1}{f_s + f_p / p} \qquad S = \frac{1}{f_s + (1-f_s)/p}$$

t_1 = time with 1 processor
t_n = time with N processors
f_s = serial fraction of code
f_p = parallel fraction of code
p = number of processors





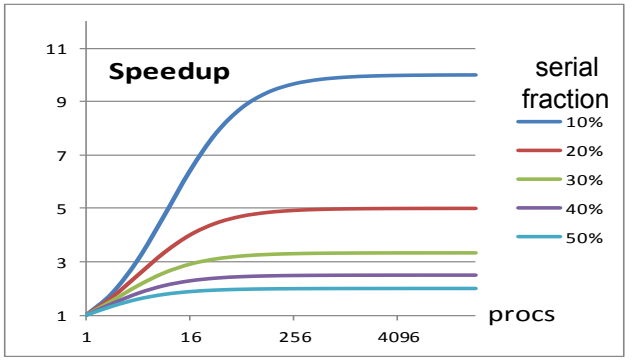
See PCSE_Scaling_Theories document.

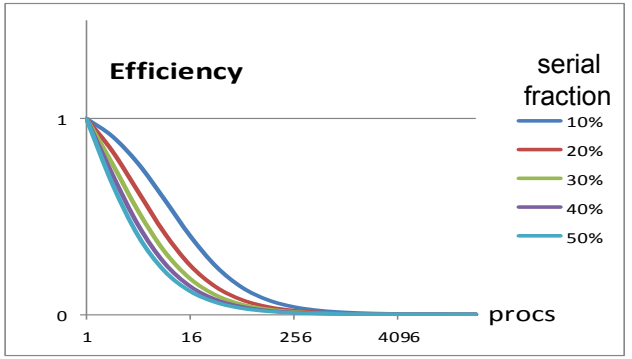
Amdahl's Law: Strong Scaling

Scalability bounds:

Under reasonable assumptions…

- Serial fraction is a severe constraint
- Can't run faster than serial time
- As you add processors, you reach a point of diminishing returns
- Speedup bounded by the reciprocal of the serial fraction





Gustafson's Law: Weak Scaling

Scalability bounds: through Speedup

Increase Parallel work with processor count.

If the parallel work is increased directly with the processor count (i.e. $t_p(p)/p$ is constant) the scalar fraction (f_s) is given by:

$$f_s = \frac{t_1}{t_1 + t_p(p)/p}$$

t_1 = serial time
t_p = parallel time
p = proc. count

The fraction of time spent executing parallel computation is $(1 - f_s)$

$$(1 - f_s) = \frac{t_p(p)/p}{t_1 + t_p(p)/p}$$

→ $t_1 = (t_1 + t_p(p)/p) f_s$
$t_p = (t_1 + t_p(p)/p)(1 - f_s)p$

$s = t_1 / t_p$

Speedup (s) is a ratio of time, elimination and rearrangement yield:

$$s \leq p + (1 - p) f_s$$

Amdahl / Gustafson Difference

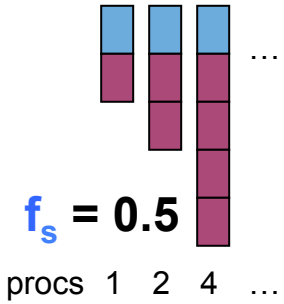
Amdahl --Fixed Work

$$s = 1 / [f_s + (1-f_s)/p]$$

$$s_2 = 1/ [0.5 + (1-0.5)/2] = 1.3$$

$$s_4 = 1/ [0.5 + (1-0.5)/4] = 1.6$$

Gustafson --Fixed Work/proc



$$s = p + (1 - p) f_s$$

$$s_2 = 2 + (1 - 2) 0.5 = 1.5$$

$$s_4 = 4 + (1 - 4) 0.5 = 2.5$$

Performance Model with Processor Dependent Work $\kappa(p)$

$$S(p) = (t_s + t_p) / (t_s + t_p/p + \kappa(p))$$

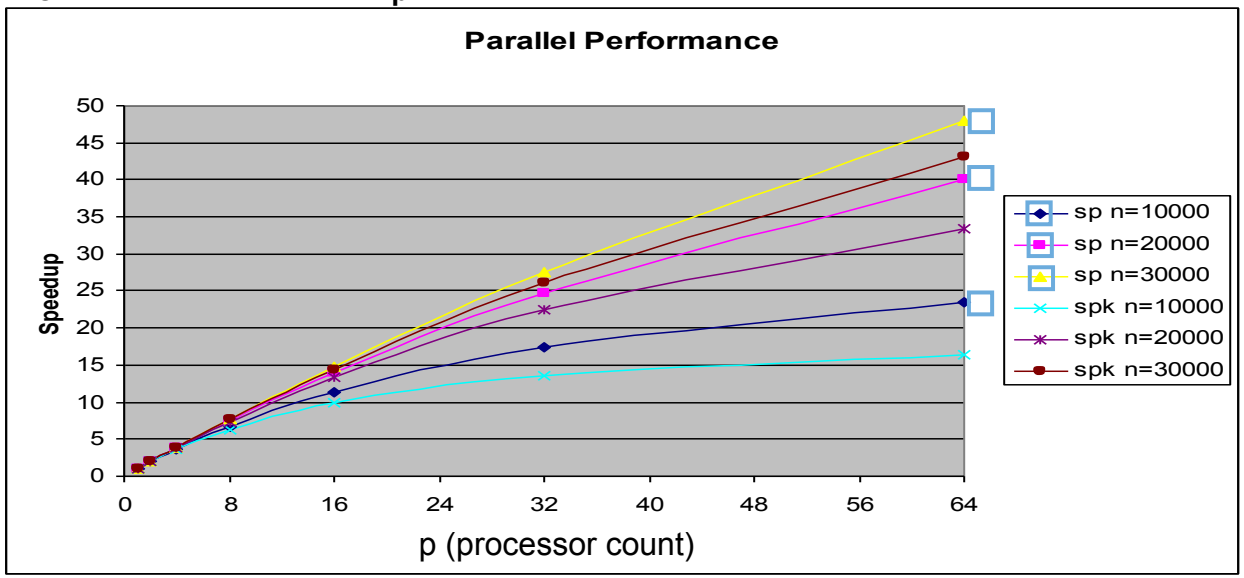
S(p) = Speedup
t_s, = serial time
t_p = parallel time
$\kappa(p)$ = "overhead" work

Next: consider all variables as dependent on n, the # of compute operations. That is consider the "complexity" of the speedup. (Requires detailed compute information.)

Performance with Variable Work (complexity)

$$s(n,p) = [t_s(n) + t_p(n)] / [t_s(n) + t_p(n)/p + \kappa(n,p)]$$

t_s = n + 18000 t_p = n**2 * 0.01 κ = n/10 + 10,000 log(p)



n = # of operations
t_s, t_p serial/parallel time
units = μsec
sp (uses t_s & t_p only)
spk (uses t_s, t_p & κ)

Amdahl Effect:
Increasing problem size increases efficiency (speedup).

Karp-Flatt Metric

Determines serial fraction from speedup, $s(\tau_s, \tau_p)$:

$$T_{(p)} = \tau_s + \tau_p / p$$
$$T_{(1)} = \tau_s + \tau_p$$
$$f_s = \tau_s / T_{(1)} = \text{serial fraction}$$

τ_s = time for serial component
τ_p = time for 1 parallel component
$T_{(1)}$ = total time, 1 proc.
$T_{(p)}$ = total time, p procs.

$$T_{(p)} = T_{(1)} f_s + T_{(1)} (1 - f_s) / p \qquad \text{now: } T_{(p)} = T_{(1)} / s$$
$$T_{(1)} / s = T_{(1)} f_s + T_{(1)} (1 - f_s) / p$$
$$\ldots$$
$$f_s = \frac{1/s - 1/p}{1 - 1/p}$$

Note, you can calculate the serial fraction from the speedup and the number of processors!

Karp-Flatt Metric

Limited by scalar sections.

p	2	4	8	16	32	64
s	1.98	3.89	7.51	13.0	24.8	40.2
f_s	0.027	0.027	0.027	0.027	0.027	0.027

****From sp n=20000 Curve**

Increased "serial fraction" implies parallel overhead.

p	2	4	8	16	32	64
s	1.98	3.86	7.35	13.4	22.4	33.5
f_s	0.035	0.036	0.038	0.041	0.043	0.046

**** From spk n=20000 Curve**

** See *Performance with Variable Work Slide*.

Architectures and Programming Models

Levels of Parallelism

Practical and Theoretical Performance Limits

Other Issues and Challenges

Summary

Correctness and Independence

Your program will/can happily execute your incorrect code

The game is to identify independence and manage dependencies

Fanghong 2005
Wikipedia Commons
http://commons.wikimedia.org/wiki/File:GreatWall2.jpg

Correctness and Independence*

C

```
for ( int i=0; i<n; i++ )
{ a[i] = b[i] + c[i];  }
```

Independent and safe

F90

```
do k=2, n-1
  a(k) = a(k-1) + a(k+1)
end do
```

Unsafe: order matters! (read/write dependence between iterations)

F90

```
do j=2, n-1
  b(j) = a(j-1) + a(j+1)
end do
```

Independent and safe (no writes to a)

C

```
for ( int i=0; i<n; i++ ) {
  temp = a[i];
  a[i] = b[i];
  b[i] = temp;}
```

Unsafe as written: temp shared

*These examples reflect situations typically encountered in OpenMP, but similar MPI examples are not hard to generate.

Correctness and Repeatability

Repeatability: the great debugging challenge

- Code may work sequentially and fail in parallel
- Behavior may vary from one run to another
- Problems may occur only at large scale

No magic bullet, but general advice

- Avoid temptation to blame the environment
- Learn to use parallel debugging tools
- Test serial/parallel behavior regularly
- Test on small test problems

Debugging and Profiling

Debugging parallel programs

- Debugging parallel program is “X” times harder
- More possible issues (Deadlock, Race condition, etc.)
- The order of operations matters (Data dependency)
- Parallel debugger

Profiling parallel programs:

- Waiting time/Idle time
- Workload balance
- Synchronization

Architectures and Programming Models

Levels of Parallelism

Practical and Theoretical Performance Limits

Other Issues and Challenges

Summary

Summary

Parallel computing is “diverse”

- Shared vs Distributed Memory (OpenMP vs MPI), Hybrid

Parallel computing is not omnipotent

- Limitation of scaling, expect a sweet spot

Parallel computing is important and fun but not trivial

- It is worth the effort