Remainder of slides from Week 4

1

Amdahl's Law

Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors. Two equivalent expressions for Amdahl's Law are given below:

$t_N = (f_p/N + f_s)t_1$ Effect of multiple processors on run time

$S = 1/(f_s + f_p/N)$ Effect of multiple processors on speedup

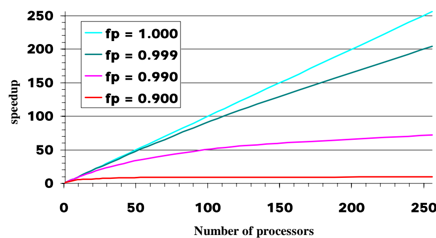
Where:

f_s = serial fraction of code
f_p = parallel fraction of code = 1 - f_s
N = number of processors

2

Illustration of Amdahl's Law

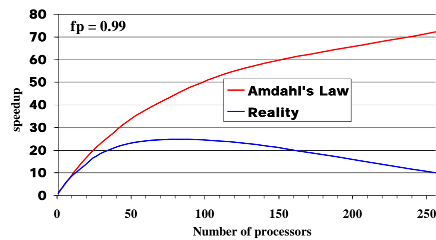
It takes only a small fraction of serial content in a code to degrade the parallel performance. It is essential to determine the scaling behavior of your code before doing production runs using large numbers of processors



3

Amdahl's Law Vs. Reality

Amdahl's Law provides a theoretical upper limit on parallel speedup *assuming that there are no costs for communications.* In reality, communications (and I/O) will result in a further degradation of performance.



4

More on Amdahl's Law

- **Amdahl's Law can be generalized to any two processes of with different speeds**
- **Ex.: Apply to $f_{processor}$ and f_{memory}:**
  - The growing processor-memory performance gap will undermine our efforts at achieving maximum possible speedup!

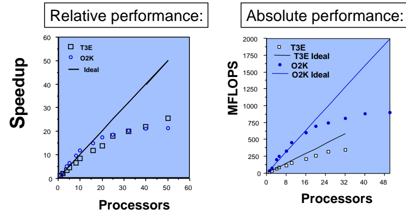
5

Gustafson's Law

- **Thus, Amdahl's Law predicts that there is a maximum scalability for an application, determined by its parallel fraction, and this limit is generally not large.**
- **There is a way around this: *increase the problem size***
  - *bigger problems mean bigger grids or more particles: bigger arrays*
  - number of serial operations generally remains constant; number of parallel operations increases: *parallel fraction increases*

6

Parallel Performance Metrics: Speedup



Speedup is only one characteristic of a program - it is not synonymous with performance. In this comparison of two machines the code achieves comparable speedups but one of the machines is faster.

7

Fixed-Problem Size Scaling

- a.k.a. Fixed-load, Fixed-Problem Size, Strong Scaling, Problem-Constrained, constant-problem size (CPS), variable subgrid
- Amdahl Limit: $S_A(n) = T(1) / T(n) = \dfrac{1}{f / n + (1 - f)}$
- This bounds the speedup based only on the fraction of the code that cannot use parallelism ($1- f$); it ignores all other factors
- $S_A \rightarrow 1 / (1- f)$ as $n \rightarrow \infty$

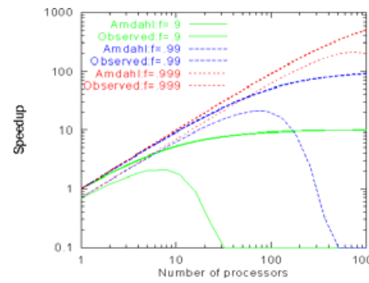
8

Fixed-Problem Size Scaling (Cont'd)

- Efficiency (n) = T(1) / [T(n) * n]
- Memory requirements decrease with n
- Surface-to-volume ratio increases with n
- Superlinear speedup possible from cache effects
- Motivation: what is the largest # of procs I can use effectively and what is the fastest time that I can solve a given problem?
- Problems:
  - Sequential runs often not possible (large problems)
  - Speedup (and efficiency) is misleading if processors are slow

9

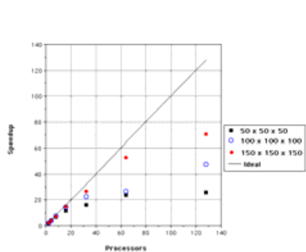
Fixed-Problem Size Scaling: Examples



S. Goedecker and Adolfy Hoisie, Achieving High Performance in Numerical Computations on RISC Workstations and Parallel Systems, *International Conference on Computational Physics: PC'97 Santa Cruz, August 25-28 1997.*

10

Fixed-Problem Size Scaling Examples



11

Scaled Speedup Experiments

- a.k.a. Fixed Subgrid-Size, Weak Scaling, Gustafson scaling.
- Motivation: Want to use a larger machine to solve a larger global problem *in the same amount of time.*
- Memory and surface-to-volume effects remain constant.

12

Scaled Speedup Experiments

- Be wary of benchmarks that scale problems to unreasonably-large sizes

    - scale the problem to fill the machine when a smaller size will do;

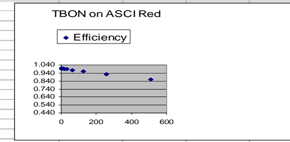
    - simplify the science in order to add computation
        -> "World's largest MD simulation - 10 gazillion particles!"

    - run grid sizes for only a few cycles because the full run won't finish during this lifetime or because the resolution makes no sense compared with resolution of input data

- Suggested alternate approach (Gustafson): Constant time benchmarks
    - run code for a fixed time and measure work done

Example of a Scaled Speedup Experiment

Processors	NChains	Time	Natoms	Time per Atom per PE	Time per Atom	Efficiency
1	32	38.4	2368	1.62E-02	1.62E-02	1.000
2	64	38.4	4736	8.11E-03	1.62E-02	1.000
4	128	38.5	9472	4.06E-03	1.63E-02	0.997
8	256	38.6	18944	2.04E-03	1.63E-02	0.995
16	512	38.7	37888	1.02E-03	1.63E-02	0.992
32	940	35.7	69560	5.13E-04	1.64E-02	0.987
64	1700	32.7	125800	2.60E-04	1.66E-02	0.975
128	2800	27.4	207200	1.32E-04	1.69E-02	0.958
256	4100	20.75	303400	6.84E-05	1.75E-02	0.926
512	5300	14.49	392200	3.69E-05	1.89E-02	0.857



12 Ways to Fool the Masses

1. Quote 32-bit performance results and compare it with others 64-bit results
2. Present inner kernel performance figures as the performance of entire application
3. Quietly employ assembly code and compare your results with others C or Fortran implementations
4. Scale up the problem size with the number of processors but fail to disclose this fact
5. Quote performance results linearly projected to a full system
6. Compare with an old code on an obsolete system
7. Compare your results against scalar, un-optimized code on
8. Base Mflop operation counts on the parallel implementation, not on the best sequential algorithm
9. Quote performance in terms of processor utilization, parallel speedup or Mflops/dollar (Peak not sustained)
10. Mutilate the algorithm used in the parallel implementation to match the architecture.
11. Measure parallel run time on a dedicated system but measure conventional run times in a busy environment
12. If all these fails, show pretty pictures and animated videos, and don't talk about performance

D. H. Bailey, "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers," in Proceedings of Supercomputing '91, Nov. 1991, pp. 4-7.