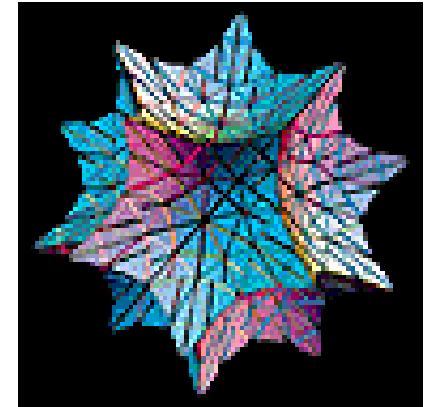
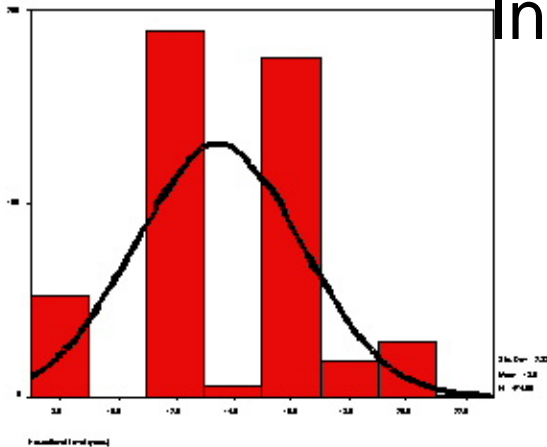


Introduction to Parallel Computing

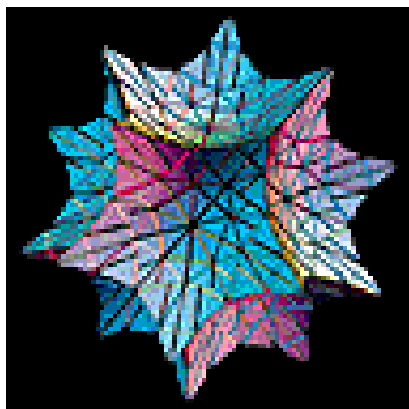
October 27, 2004



Presented by the
ITC Research Computing Support Group
Kathy Gerber, Ed Hall, Katherine Holcomb, Tim F. Jost Tolson

- Introduction to Parallel Computing – TODAY
- Mathematics on the Desktop by Kathy Gerber – Wednesday, November 3, at 3:30 PM
- Optimization by Ed Hall – Wednesday, November 10 at 3:30 PM
- Introduction to MPI by Katherine Holcomb – Wednesday, November 17 at 3:30 PM

ITC Research Computing Support Introduction to Parallel Computing



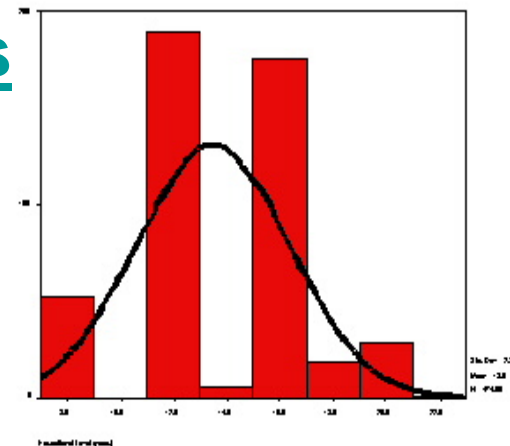
By: Katherine Holcomb

Research Computing Support Center

Phone: 243-8800 Fax: 243-6604

E-Mail: Res-Consult@Virginia.EDU

<http://www.itc.Virginia.edu/researchers>



Why Compute in Parallel?

Most large scientific problems
exceed the capabilities of a single
processor

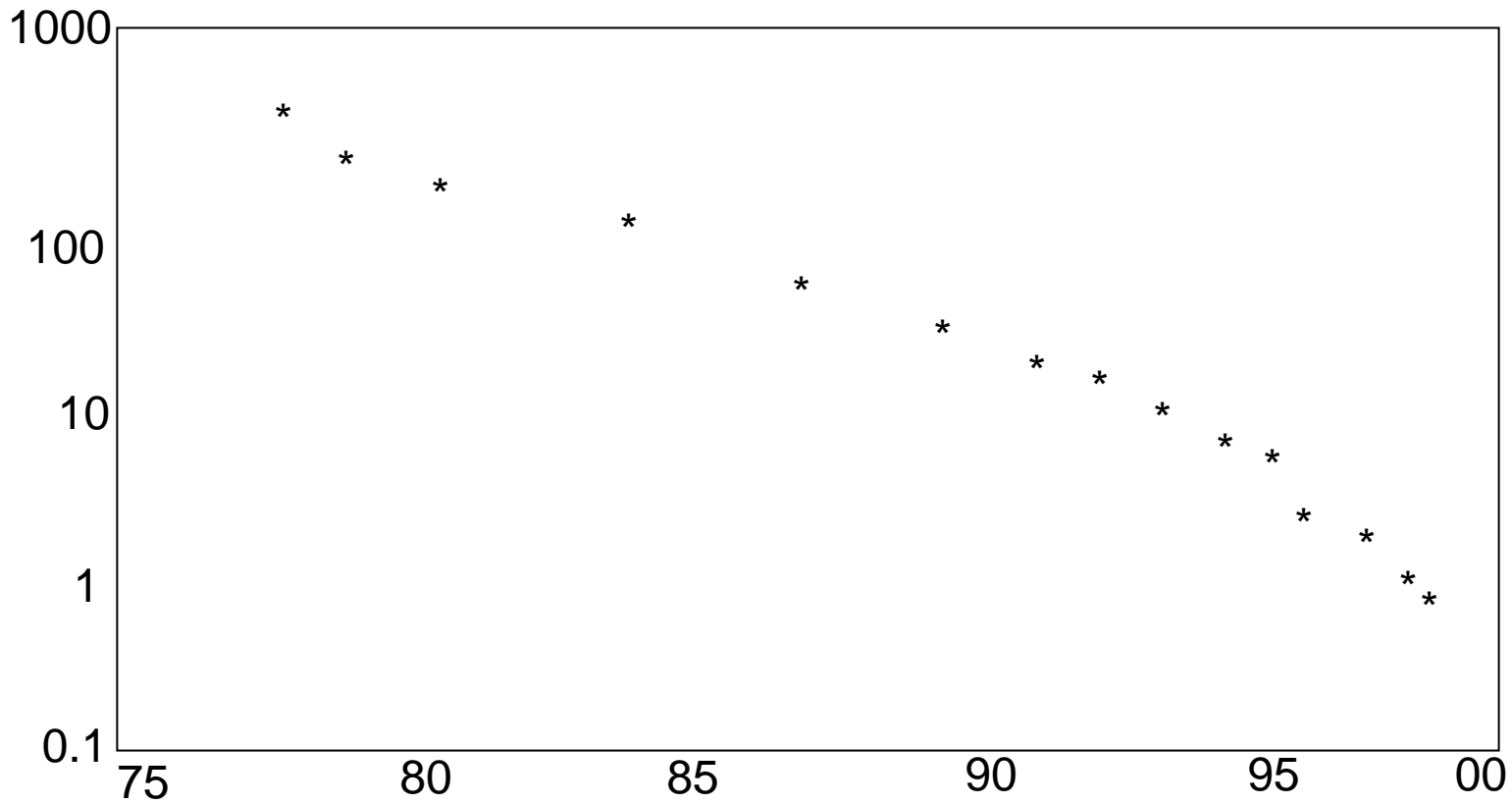
CPU clock cycle times continue to fall roughly according to Moore's Law – halving approximately every 18-24 months.

BUT

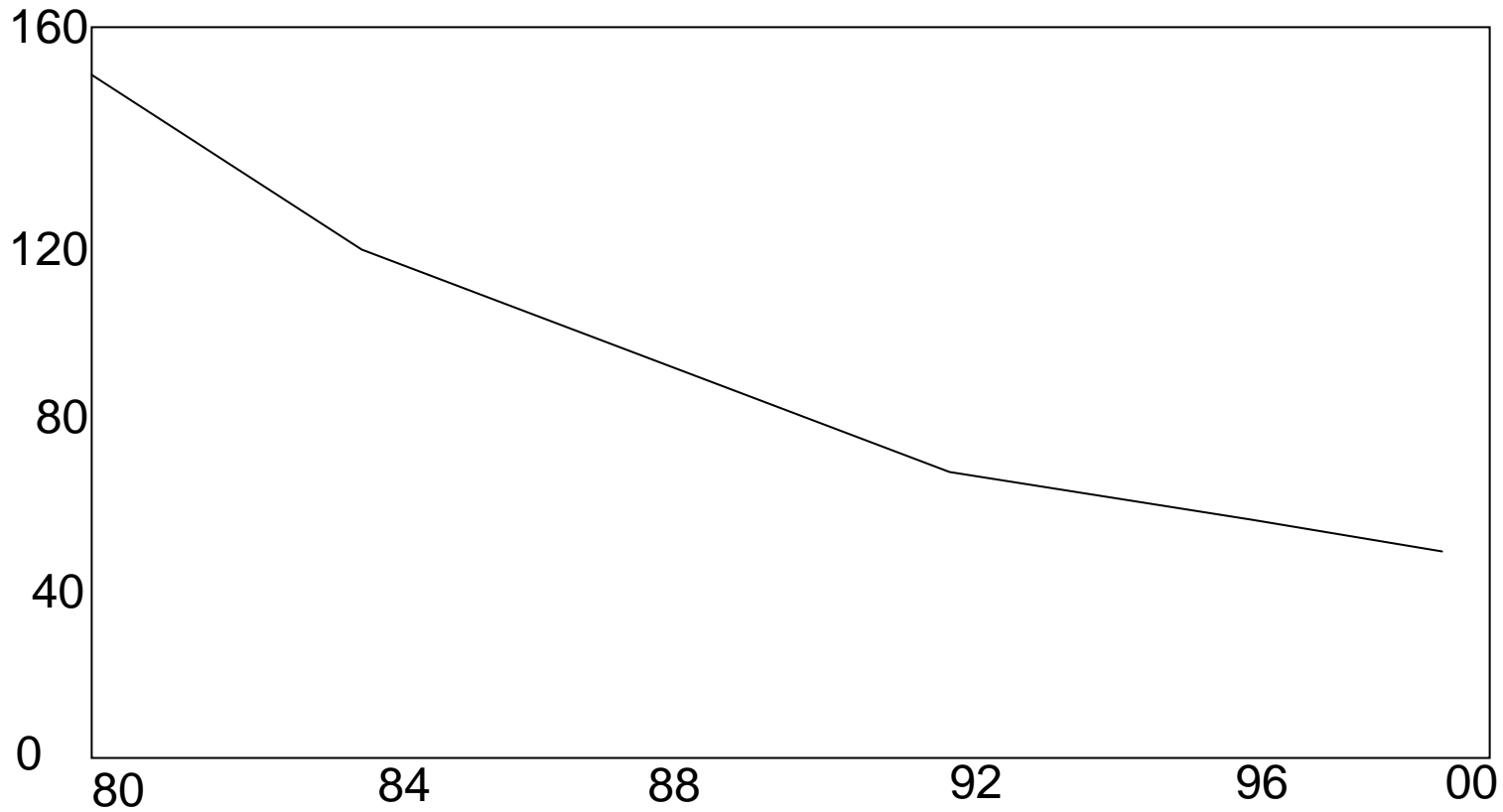
Decrease in memory latency (the time required to access the first bit of information) is not keeping up.

Other factors: disk capacities have increased enormously, but they are mechanical devices and their latencies decrease fairly slowly.

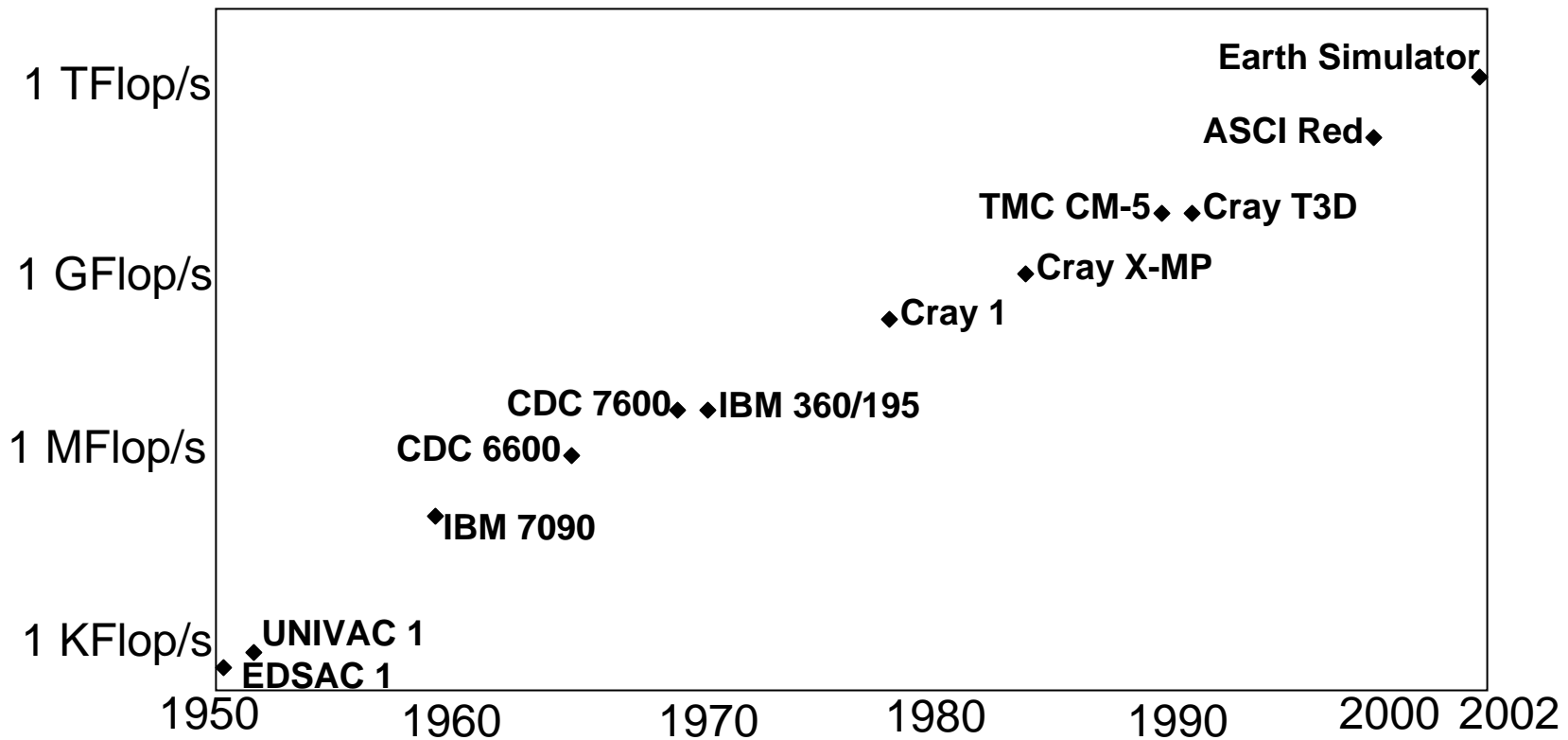
Network speeds (internal and external) have also increased fairly slowly, and this speed is ultimately limited by light travel times across the network.



Clock rate in nanoseconds versus time in years
Note the logarithmic scale on the vertical axis



**DRAM latency in nanoseconds versus time in years
Here the scale on the vertical axis is linear**



Peak performance of supercomputers versus time

Flop/s = floating-point operations/second

Peak performance for supercomputers has followed Moore's Law quite well, but only about half the improvement is due to increases in single-processor speed, while the rest is due to an increasing number of processors.

In many cases, parallelism has been the only means by which large computational runs can be performed.

Scalability

Parallelizing a code does not always result in a speedup; sometimes it actually slows the code down! This can be due to a poor choice of algorithm or to poor coding.

Define the speedup to be

$$\frac{T(1)}{T(N)}$$

where N = number of processors, $T(1)$ = time for serial run.

The best possible speedup is linear, i.e. it is proportional to the number of processors: $T(N) = T(1)/N$.

A code that continues to speed up reasonably close to linearly as the number of processors increases is said to be **scalable**. Many codes scale up to some number of processors but adding more processors then brings no improvement. Very few, if any, codes are indefinitely scalable.

Amdahl's Law

It is for practical purposes impossible to parallelize all parts of a code. Let the fraction of the code that is perfectly parallel be p , so the time for the parallel part is $T_p = pT(1)/N$; the time for the serial part is then $T_s = (1-p)T(1)$. **Amdahl's Law** says that

$$\text{Speedup} = \frac{1}{(1-p) + p/N}$$

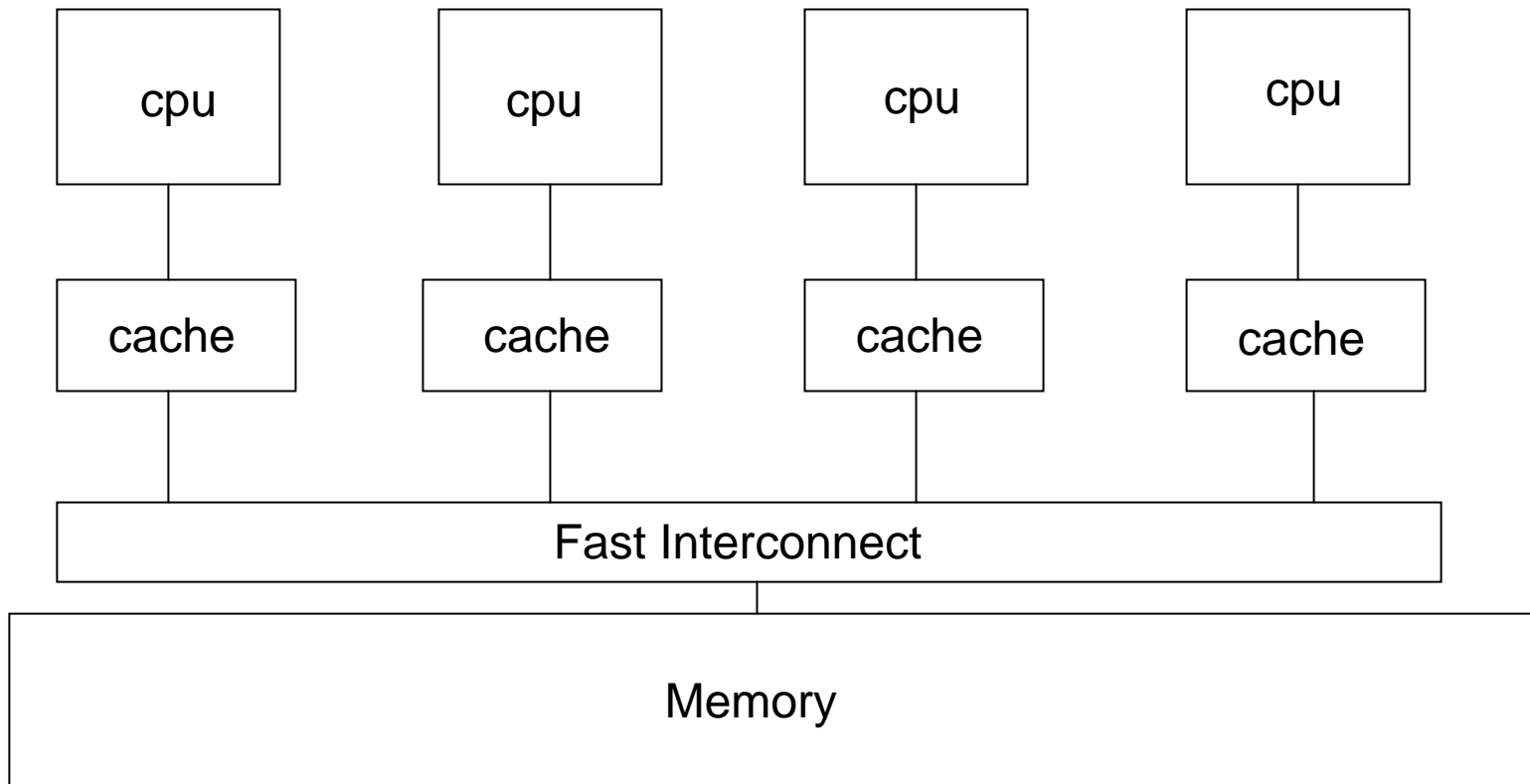
which is bounded by $1/(1-p) = T(1)/T_s$ as $N \rightarrow \text{infinity}$.

One way to reduce $T_s/(T_s+T_p)$ is to increase the problem size so that the parallel parts dominate.

Parallel Architectures

MIMD (multiple instructions multiple data)

SMP – Symmetric Multiprocessing
Uniform Memory Access (UMA)

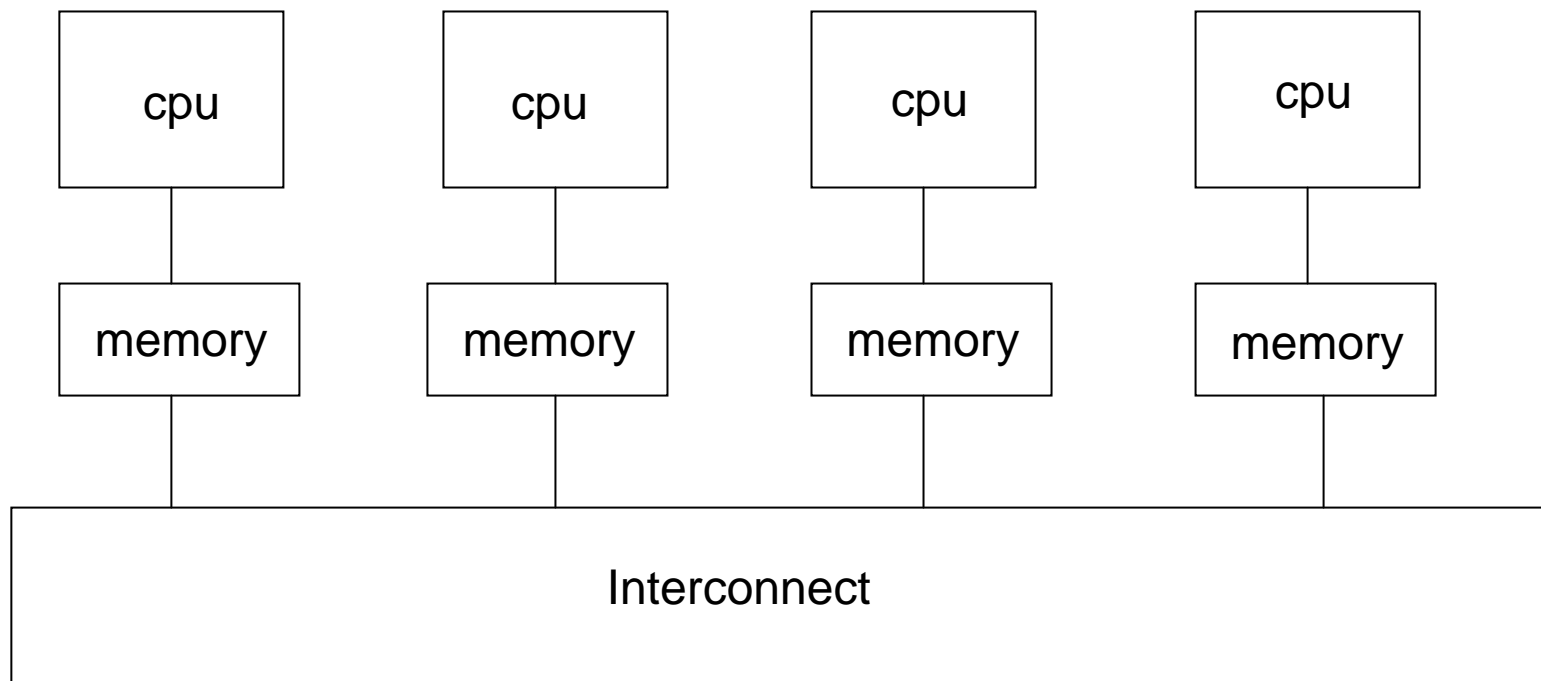


Parallel Architectures

MIMD (multiple instructions multiple data)

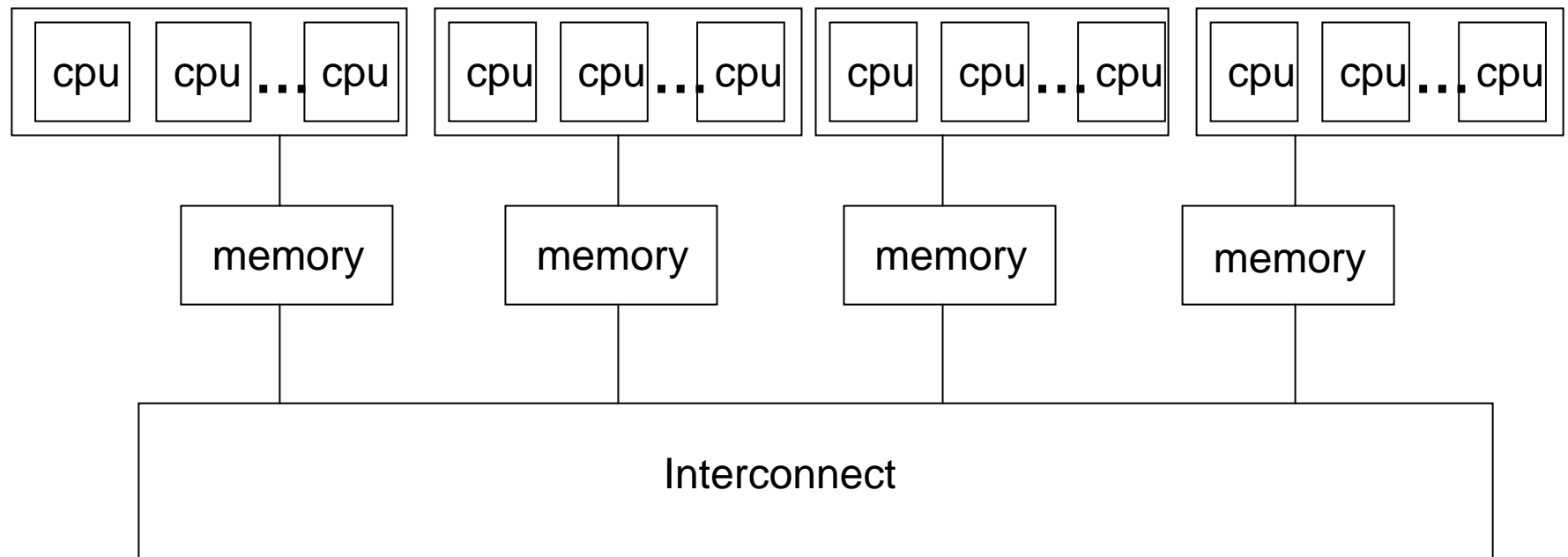
MPP – Massively Parallel Processor

Updated version is NonUniform Memory Access (NUMA)



Parallel Architectures

“Constellation”



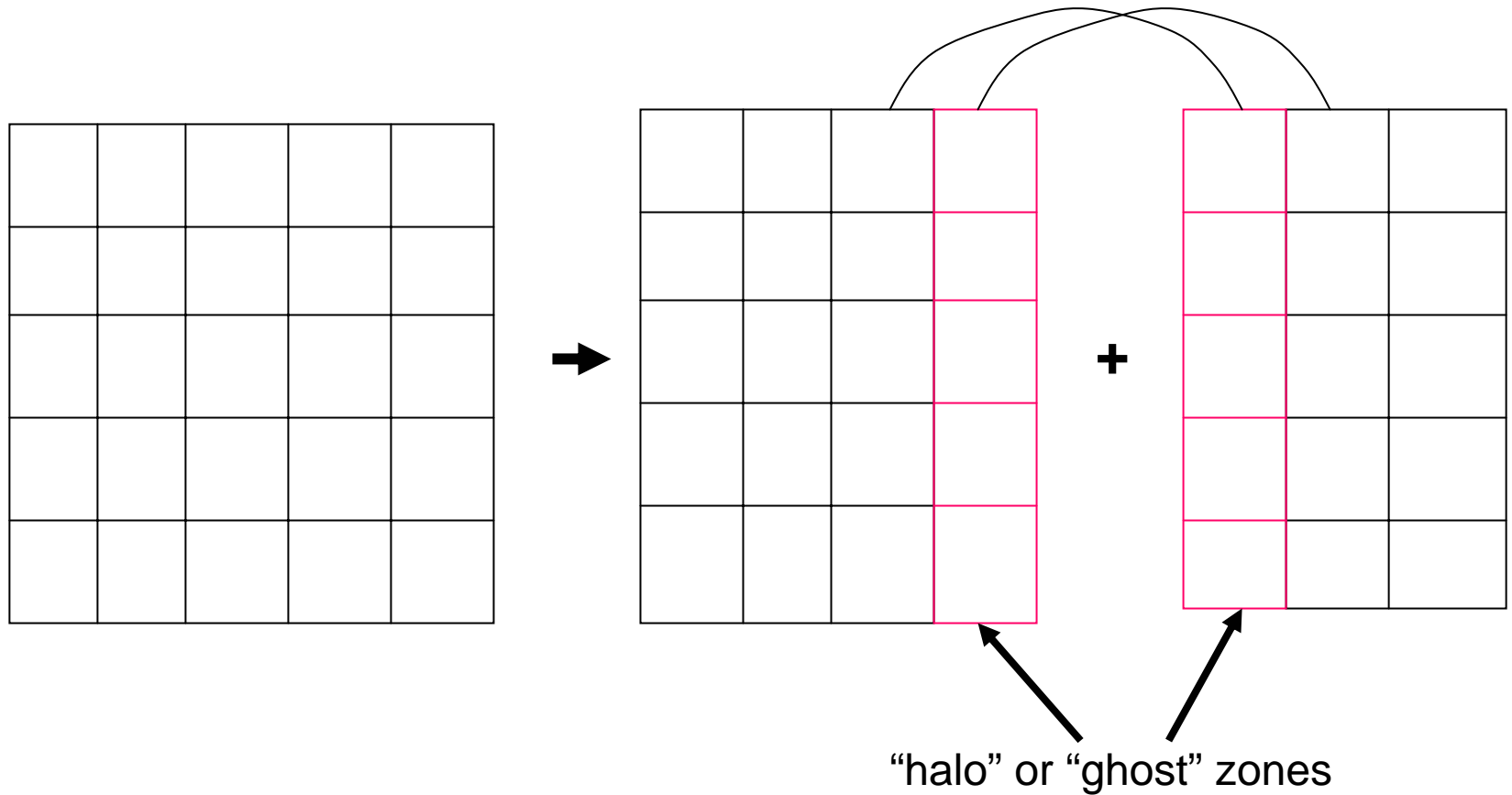
Programming Models

- Threads/OpenMP
 - For SMP
 - Easy to use but hard to get a speedup
- Message Passing
 - For MPP or Distributed Memory Clusters
 - Harder to use but generally gives best results
 - Can be used with SMP systems with right libraries
- Hybrid
 - For “constellations”
 - Rarely results in any benefit over message passing

Message Passing

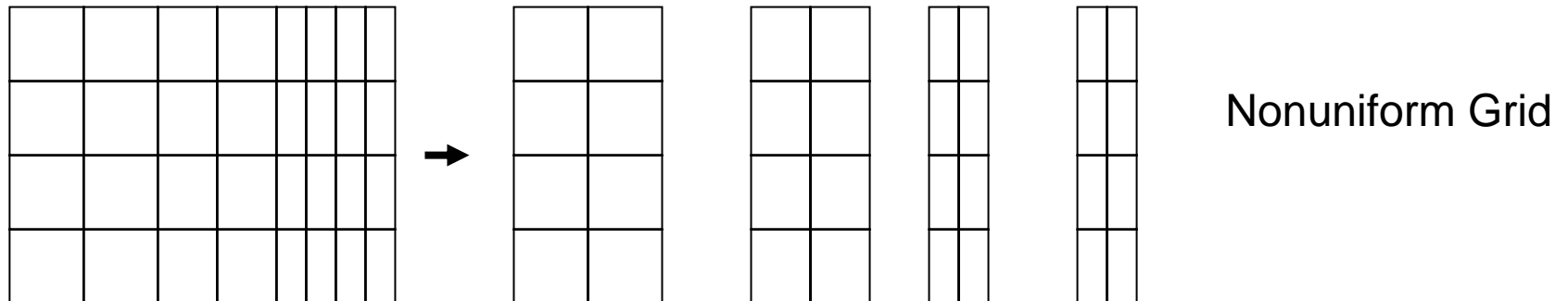
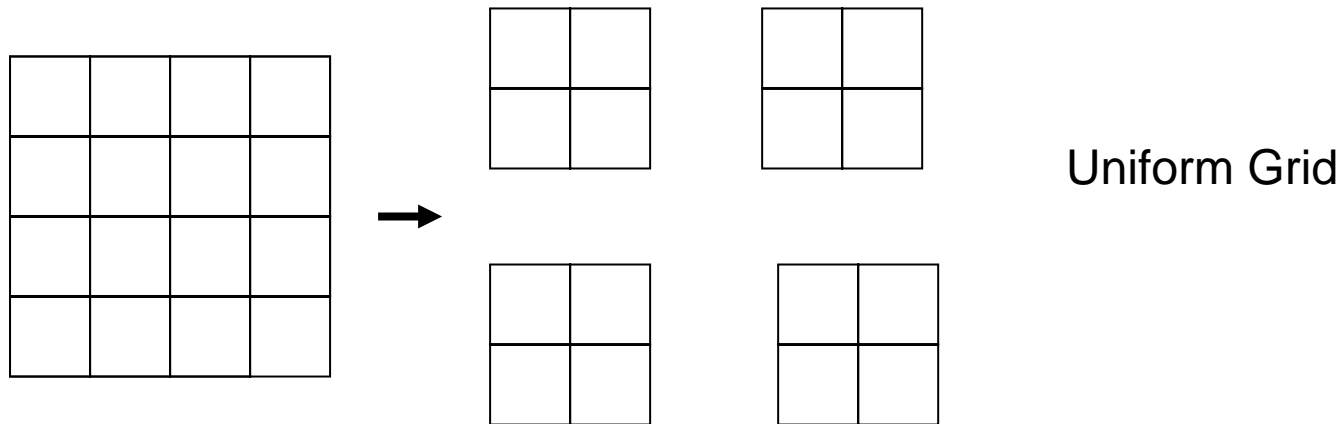
- Task parallelism
 - divide tasks among the processors.
 - sometimes can be “embarrassingly parallel” with very little interprocess communication required.
- Data parallelism
 - divide data among the processors.
 - processors must exchange boundary information at regular intervals.

Block Data Decomposition



Load Balancing

Data should be distributed among processors so that each has an approximately equal quantity of work. Otherwise some will sit idle waiting for communications from other processes.

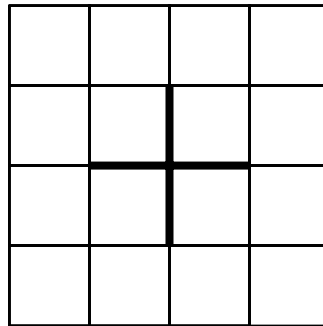


Example: Jacobi Iteration

Laplace Equation: $\nabla^2 T = 0$

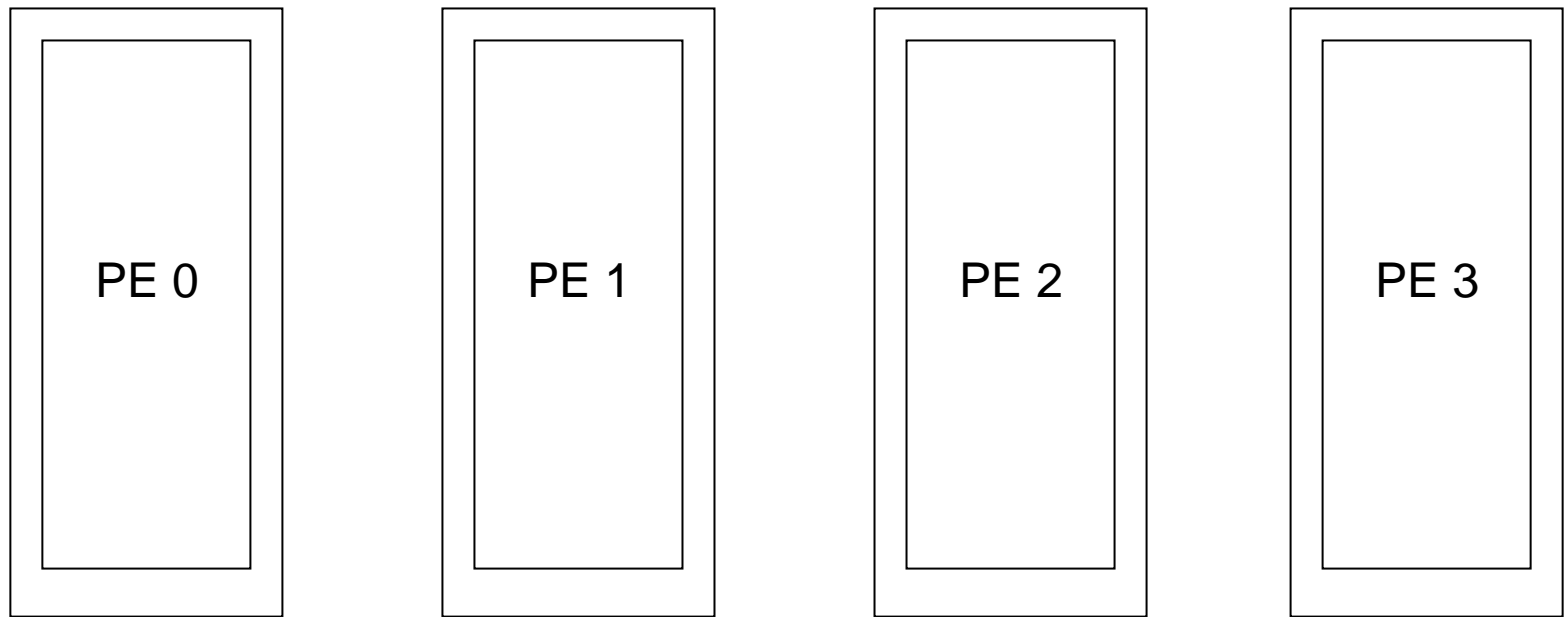
$$T_n = 0.25 * (T_{n-1}(i-1,j) + T_{n-1}(i+1,j) + T_{n-1}(i,j-1) + T_{n-1}(i,j+1))$$

This leads to a five-point stencil



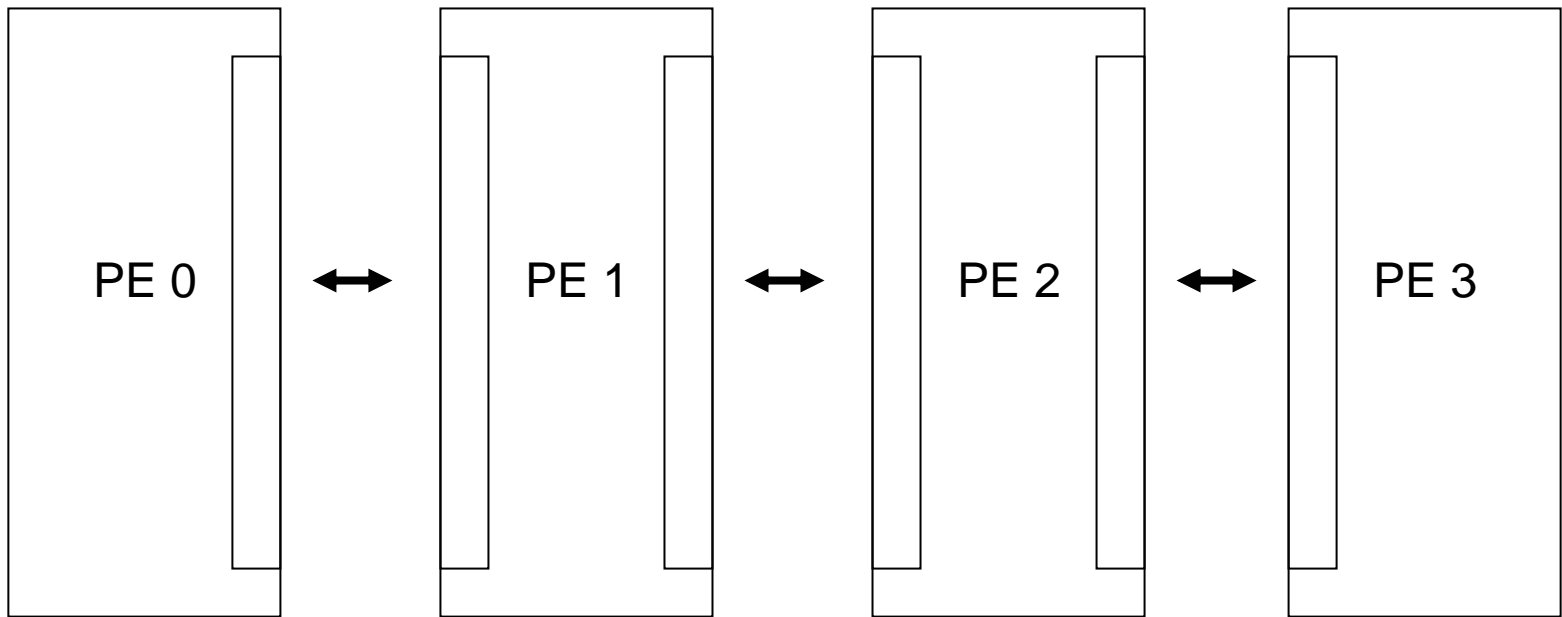
Sample Parallelization Strategy

Break the grid into groups of columns for Fortran, rows for C.



The outer rectangles accommodate boundary and “ghost” zones.

Exchange Edge Data at Each Iteration



General Advice for Parallelization

- Start with a **correct, legible** serial code. Rewrite if necessary.
- Determine whether the serial algorithm can be parallelized. Consider alternatives that may be more scalable.
- Test and debug on a small number of processes (2 to 4).

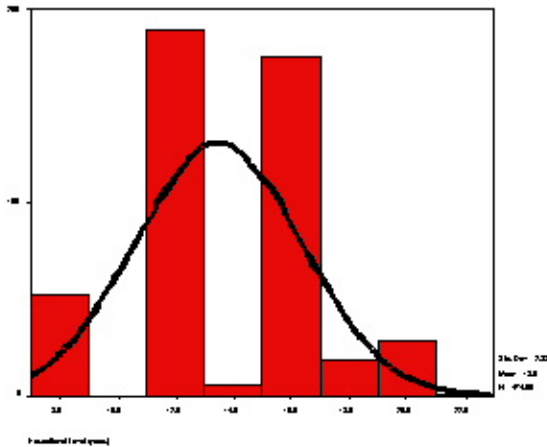
Parallel Libraries

- Well tested and widely used—don't reinvent the wheel.
- If not already present, ITC can install them.
- Examples include ScaLAPACK <http://www.netlib.org/scalapack>, PETSc <http://www-unix.mcs.anl.gov/petsc/petsc-2/>

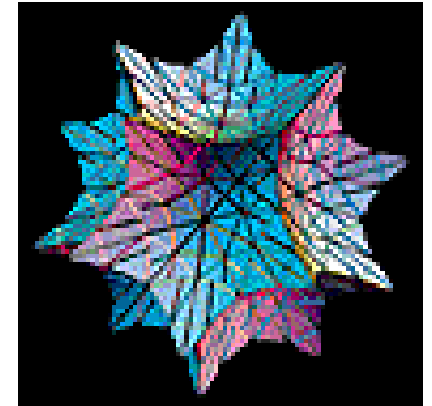
Summary

- Simultaneous use of multiple compute resources.
- Parallelism can be coarse- or fine-grained.
- Saves wall-clock time, solves bigger problems
- Make sure serial program optimized before parallelizing it.

www.llnl.gov/computing/tutorials/workshops/workshop/parallel_comp/



Upcoming Talk



Introduction to MPI
Wednesday, November 17 at 3:30 PM
Wilson 244

Tutorial is online at www.itc.virginia.edu/research/linux-cluster/hands-on

Talks are online at www.itc.virginia.edu/research/talks