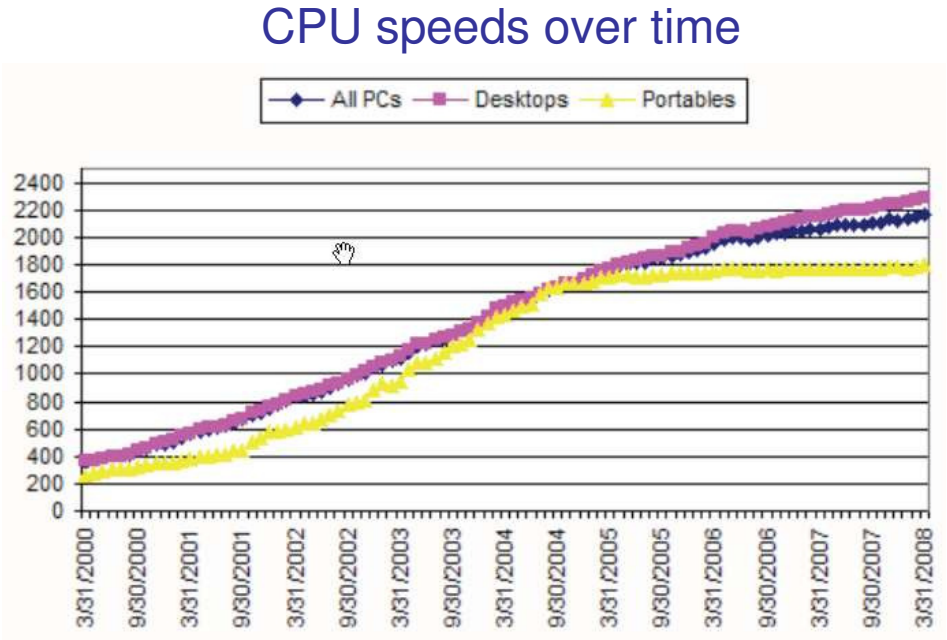
Parallel Processing, Math and MPI

(A demonstration solving the Discrete Poison Equation using Jacobi's Method and MPI)

Parallel Processing is the future.

- The increasing capabilities of serial computer's are leveling out. If we wish to continue to solve larger problems, we must use parallel processing.



- Parallel processing increases computational rate, but more importantly increases memory capacity and bandwidth, the true weak links.

Memory speeds are the true performance problem.

Maybe your CPU is 3GHz (3 billion cycles per second), but how fast is data being supplied?

The following supply 64 to 128 bytes per….

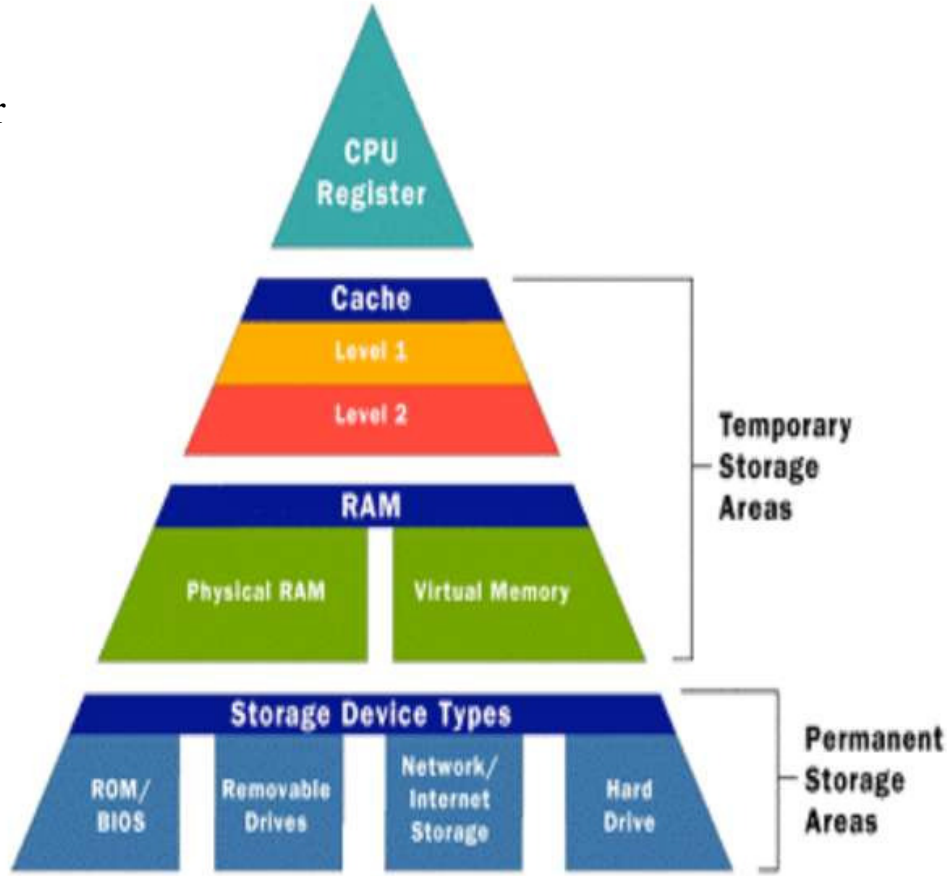
(Register) 1 cpu cycle (100's Bytes)

(Level 1 Cache) A few cycles (10's KB)

(Level 2 Cache) 5 to 30 cycles (512 KB+)

(RAM) 100's of cycles (multiple GB's)

(Disk) Million's of cycles (very large)



Example Problem:

Let's solve Poison's Equation using finite differences on the unit cube in 3D with $h = 1/1000$

$$-\nabla^2 u = 1 \quad \text{on } \Omega \quad \text{with} \quad u = 0 \text{ on } \partial\Omega$$

After substituting in finite difference approximations for all 3 second partial derivatives, we have 1 billion equations and 1 billion unknowns!

$$\frac{\partial^2 u}{\partial x^2}\left(u_{i,j,k}\right) \approx \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{h^2}$$

Let's Parallelize Basic Linear Iteration to Solve This.

We want to solve $Au = f$

Basic linear iteration: $u^{(k+1)} = \left(I - BA\right)u^{(k)} + Bf$

with initial guess: $u^{(0)} = 0$

with a Jacobi preconditioner:

$$A = L + D + U, \text{ let } B = D^{-1}$$

Disclaimer: To solve for this many variables, you wouldn't really want to use this method which converges in Order N^2. A better method would be basic linear iteration with a multigrid preconditioner (Order N) or Conjugate Gradient with a multigrid preconditioner (Order N?).

To calculate each new u^k will require billions of multiplications and to store u^k at each iteration to double precision will require 8 Gigabytes of storage.

A typical desktop would need to store each new u on the hard drive. Its 3GHz processor would be continually accessing the hard drive and be working overtime to compute each iteration.

We must parallelize the algorithm. Then we will use NCSA's supercomputer ABE, which has 9600 GB of main memory (RAM) and an 88,000 Gigaflops processor, to solve this.

Key parallelizing issues:

- Shared memory or not?
- Minimize serial sections (Amdahl's Law),
- Minimize non-productive work associated with parallel environment

  i.e initialization, communication, etc.
- Minimize load imbalance.

Math programming issue:

- Optimize for cache (faster memory) use.

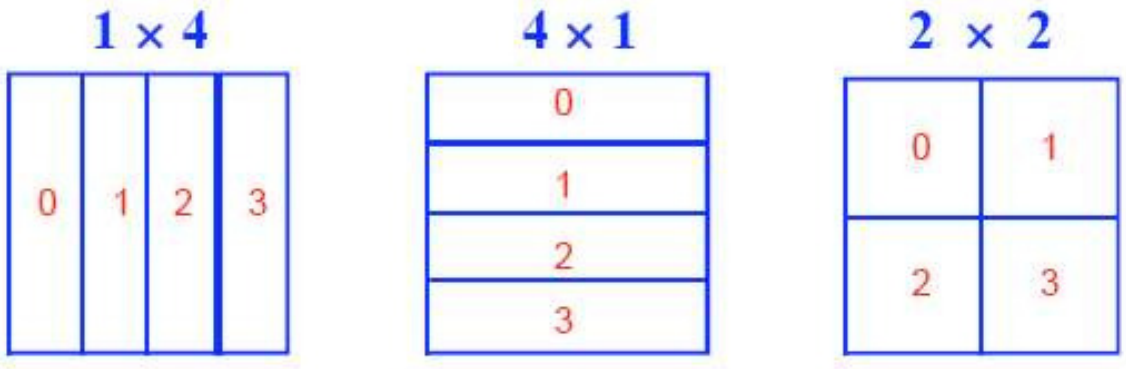
Analyze work dependencies and data dependencies:

By analyzing $u^{(k+1)} = \left(I - D^{-1}A\right)u^{(k)} + D^{-1}f$

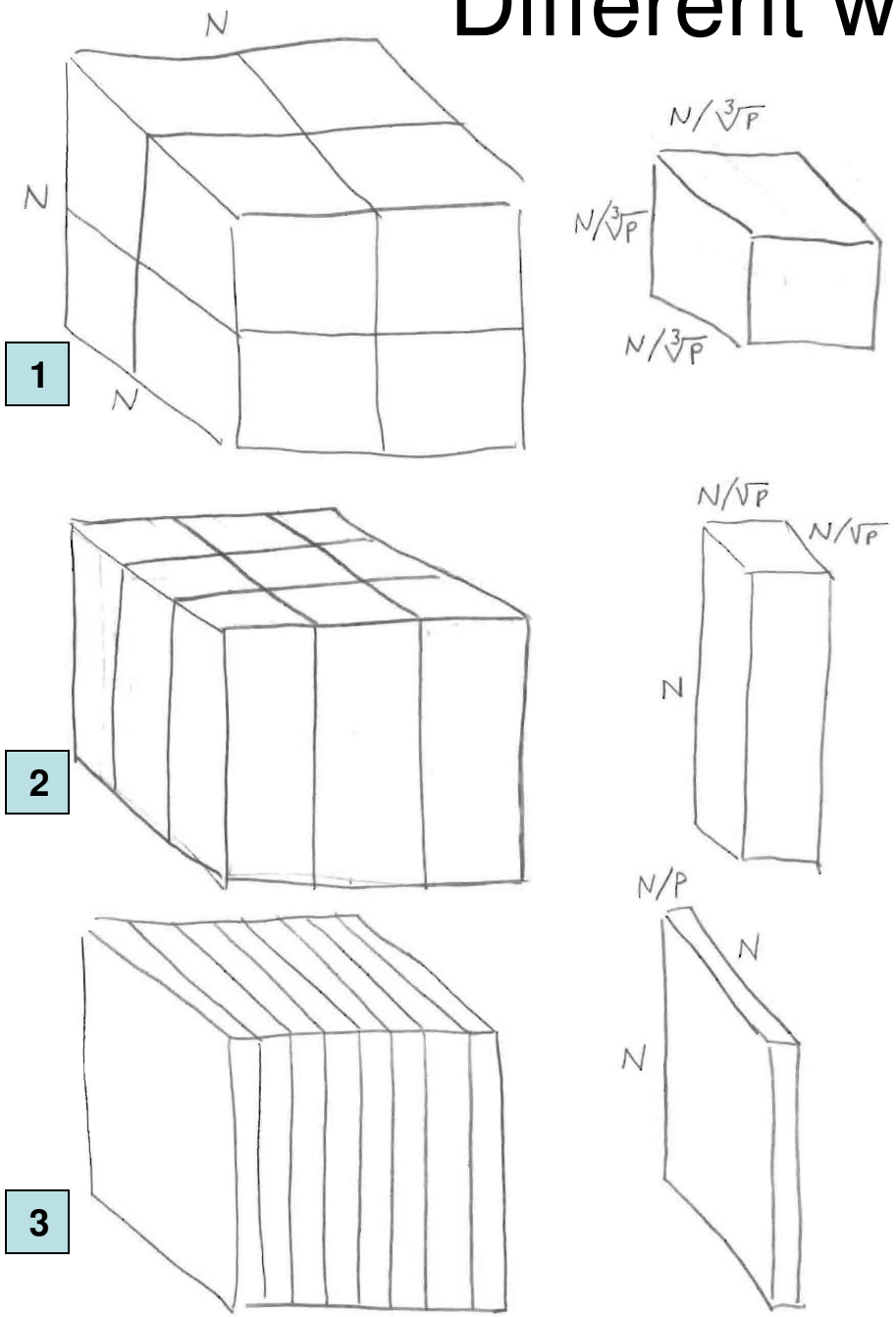
with our specific matrix A, we see:

$$u^{(k+1)}{}_{i,j,k} = \frac{1}{6}\left(u^{(k)}{}_{i,j,k+1} + u^{(k)}{}_{i,j,k-1} + u^{(k)}{}_{i,j+1,k} + u^{(k)}{}_{i,j-1,k} + u^{(k)}{}_{i+1,j,k} + u^{(k)}{}_{i-1,j,k} + fh^2\right)$$

So each processor can compute some $u^{(k+1)}_{i,j,k}$'s using localized data.



Different ways to divide the work (domain), affect communication.



$\alpha = 1.7 \times 10^{-6}$ sec/setup packet on ABE

$\beta = 1.27 \times 10^{-9}$ sec/byte transfer on ABE

$P =$ number of processors

$T_p^{comm} =$ time for every processor to send and receive their boundaries simultaneously

$= \alpha + \beta x$ where $x =$ bytes of data

Let $P = 128$ and $N = 1000$

1 $T_p^{comm} = 6P\left(\alpha + \beta\left(N^2 / P^{2/3}\right)\right) = 0.040$ sec

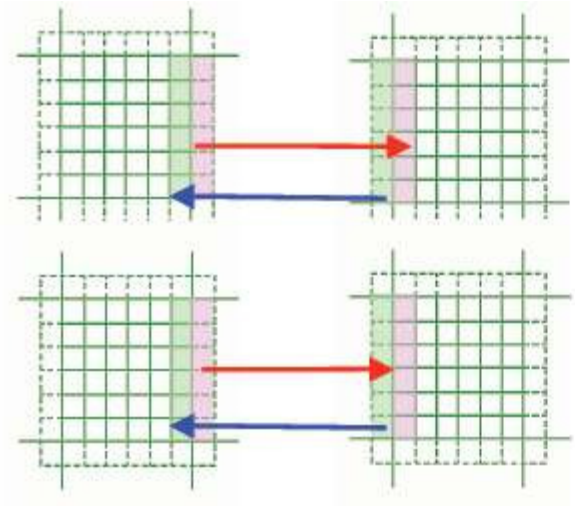
2 $T_p^{comm} = 4P\left(\alpha + \beta\left(N^2 / \sqrt{P}\right)\right) = 0.058$ sec

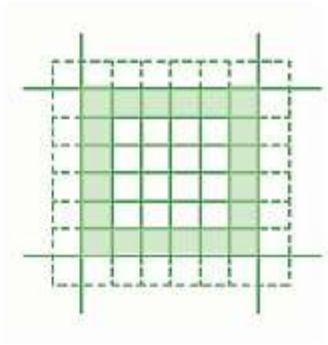
3 $T_p^{comm} = 2P\left(\alpha + \beta N^2\right) = 0.33$ sec

Mpirun this Core Code

```
1.  procedure Solve (Array3D<double>A, h, f)
      // A is a (n+2)-by-(n+2)-by-(n+2) array
2.  begin
3.  for i = 1 to n do
4.    for j = 1 to n do
5.      for k = 1 to n do
6.        A'[i,j,k] = (1/6) * ( A[i,j,k+1] + A[i,j,k-1] + A[i,j+1,k]
              + A[i,j-1,k] + A[i+1,j,k] + A[i-1,j,k] - f * h^2);
7.      end for
8.    end for
9.  end for
10. A[:,:,:]=A'[:,:,:];
11. end procedure
```

While exchanging boundary information and we're done.





MPI (Message Passing Interface) the basic routines

- MPI_Init — Initializes MPI
- MPI_Finalize — Terminates MPI
- MPI_Comm_size — Determines number of processors
- MPI_Comm_rank — Determines the label of the calling process
- MPI_Send — Sends a message
- MPI_Recv — Receives a message

Sample MPI program

```
#include <stdio.h>
#include "mpi.h"

int rank,size,i;
char name[30];
MPI_Status status;
const int tag=99;

int main(int argc,char **argv)
{
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);

   if (rank==0)  {
     printf("Processor 0 would like to know your name: ");
     scanf("%s",name);
     for (i=1;i<size;i++)
       MPI_Send(name, 30, MPI_CHAR, i, tag, MPI_COMM_WORLD);
   }
   else{
     MPI_Recv(name, 30, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
     printf("Hello %s from Processor %d of %d\r\n",name,rank,size);
   }

   MPI_Finalize();
   return(0);
}
```

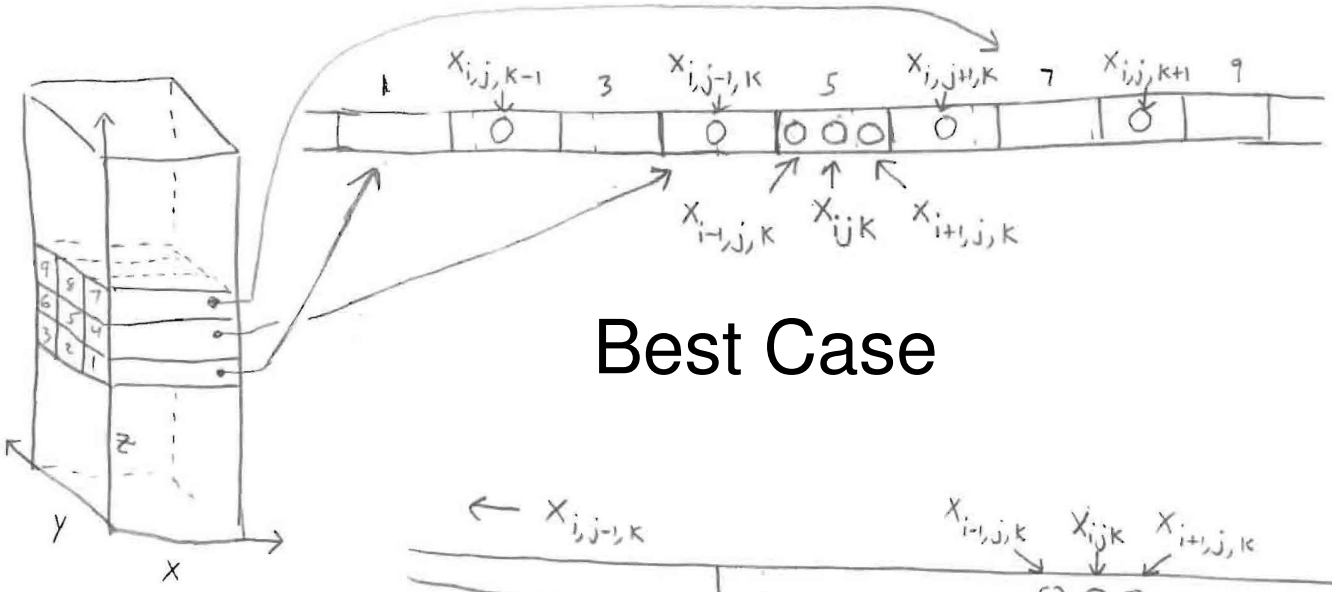
Compile by typing “mpicc HelloWorld.c –o HelloWorld”

Run on 10 processors by typing “mpirun –np 10 ./HelloWorld”

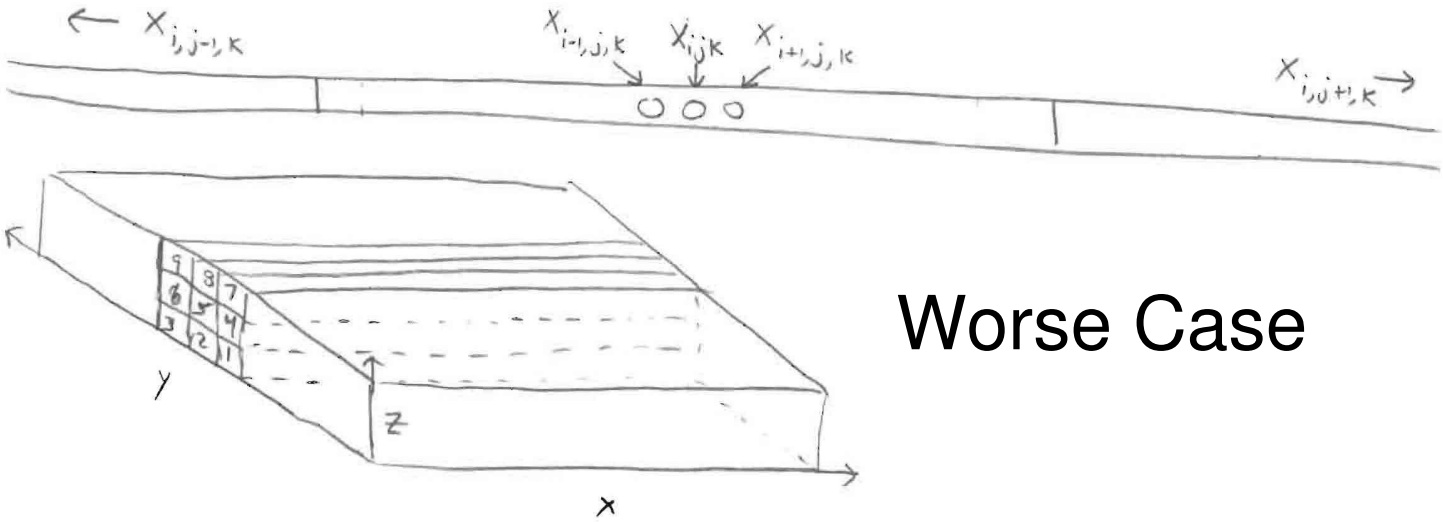
Optimize for Cache Use

3D Subdomain to 1D Memory $u_{i,j,k} \rightarrow \text{memory}\,[i + N_x j + N_x N_y k]$

(Be Careful !)



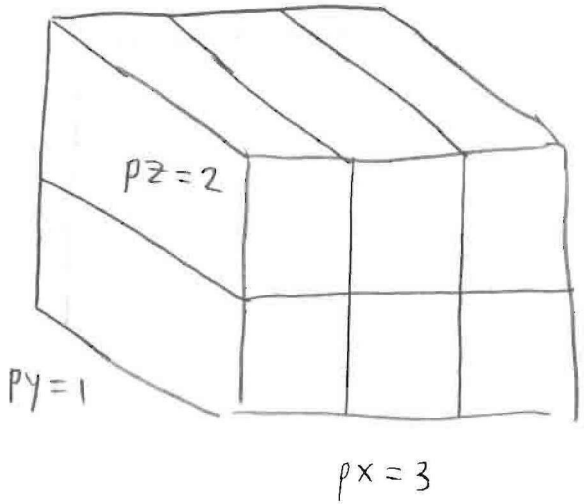
Best Case



Worse Case

Actual running times on ABE for N = 1000 and P = 128. Times are in seconds for 20 iterations with and without communication for comparison.

$px \cdot py \cdot pz = P$ processors
with domain divided like this:



Px	Py	Pz	w/ comm	no comm	comm	percent
1	1	128	23.83	14.95	8.88	37.27
1	2	64	20.33	14.63	5.69	28.01
1	4	32	21.12	14.49	6.63	31.40
1	8	16	17.48	14.20	3.28	18.78
1	16	8	16.55	14.07	2.48	14.99
1	32	4	19.25	13.43	5.81	30.21
1	64	2	16.89	11.88	5.01	29.65
1	128	1	22.35	12.03	10.31	46.16
2	1	64	23.46	14.66	8.80	37.51
2	2	32	21.76	14.49	7.27	33.42
2	4	16	18.26	14.18	4.08	22.32
2	8	8	18.21	13.92	4.29	23.54
2	16	4	16.02	13.36	2.66	16.61
2	32	2	15.23	11.66	3.58	23.49
2	64	1	19.31	11.53	7.78	40.31
4	1	32	20.01	14.54	5.46	27.32
4	2	16	19.11	14.21	4.90	25.65
4	4	8	17.27	13.93	3.34	19.33
4	8	4	19.04	13.20	5.84	30.67
4	16	2	16.26	11.26	4.99	30.72
4	32	1	16.97	11.19	5.78	34.05
8	1	16	18.83	14.31	4.52	24.00
8	2	8	20.26	14.01	6.25	30.85
8	4	4	17.19	13.21	3.98	23.17
8	8	2	15.36	11.43	3.93	25.60
8	16	1	16.70	10.92	5.78	34.60
16	1	8	19.69	14.29	5.40	27.43
16	2	4	18.57	13.29	5.28	28.43
16	4	2	17.34	11.41	5.92	34.17
16	8	1	16.66	10.91	5.75	34.50
32	1	4	21.89	13.99	7.91	36.11
32	2	2	17.28	12.13	5.15	29.80
32	4	1	19.13	11.30	7.83	40.91
64	1	2	22.89	12.64	10.24	44.76
64	2	1	20.97	11.86	9.10	43.42
128	1	1	33.00	13.18	19.83	60.07

We Can Do Better

Reducing communication percentage increases efficiency and speedup.

Speed Up = S_p = best serial time / parallel time
Efficiency = E_p = speed up / number of processors

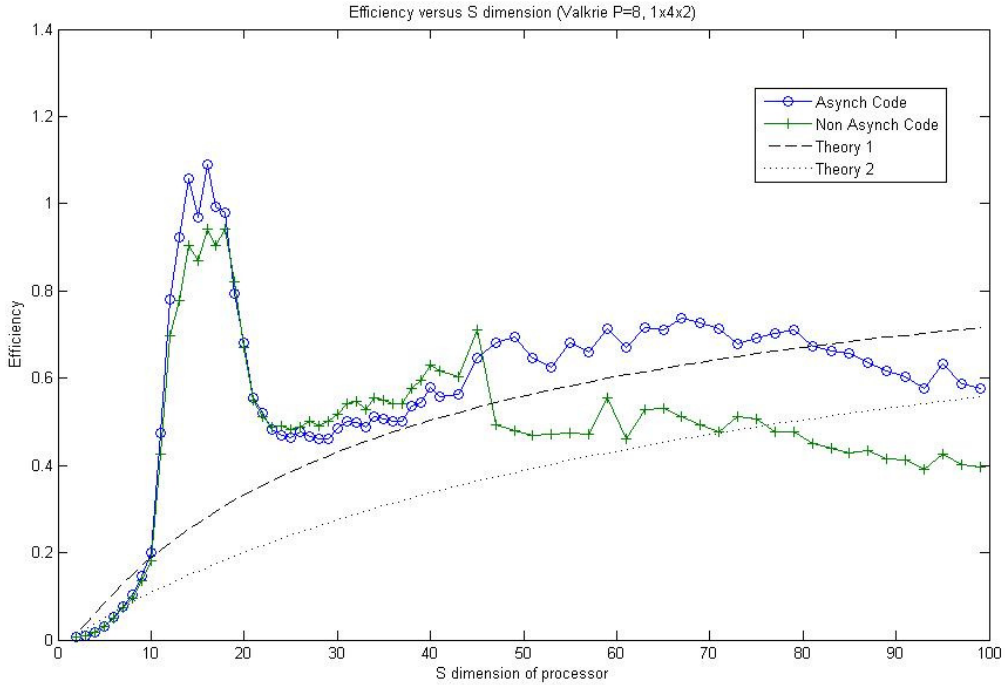
$$E_p = \frac{S_p}{P} = \frac{T_1}{P \cdot T_p} \approx \frac{P \cdot T_p^{comp}}{P \cdot \left(T_p^{comp} + T_p^{comm}\right)} = \frac{T_p^{comp}}{T_p^{comp} + T_p^{comm}} = \boxed{1 - \frac{T_p^{comm}}{T_p^{total}}} = \frac{1}{1 + T_p^{comm} / T_p^{comp}}$$

T_p^{comm} = time for one processor to retrieve its boundary

T_p^{comp} = time for one processor to update its domain once.

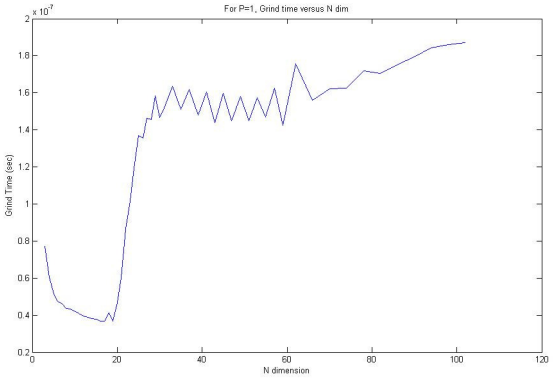
We can transmit boundary information while we are computing effectively lowering T_p^{comm}

Efficiency Increases



Increase in efficiency depends on the size of each processor's subdomain size because the ratio of surface area to volume does.

$$S = \sqrt[3]{\text{subdomain unknowns}}$$
$$= N \Big/ \sqrt[3]{P}$$



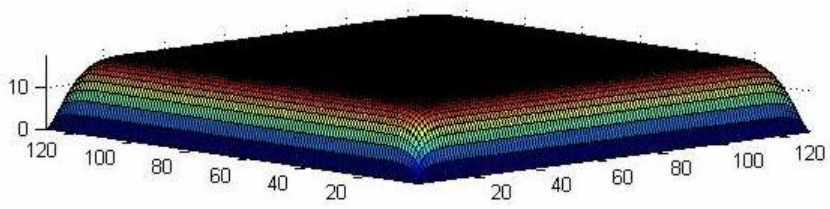
Unexpected Discovery:

Serial processes can be accelerated by mimicking parallel processes.

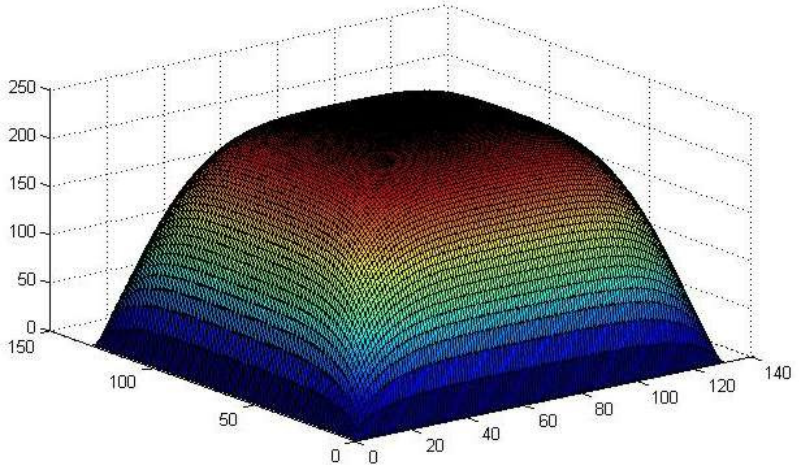
The data on the right shows that with smaller domains, a processor can compute 4 multiplications as fast as 1!

The Solution Converging

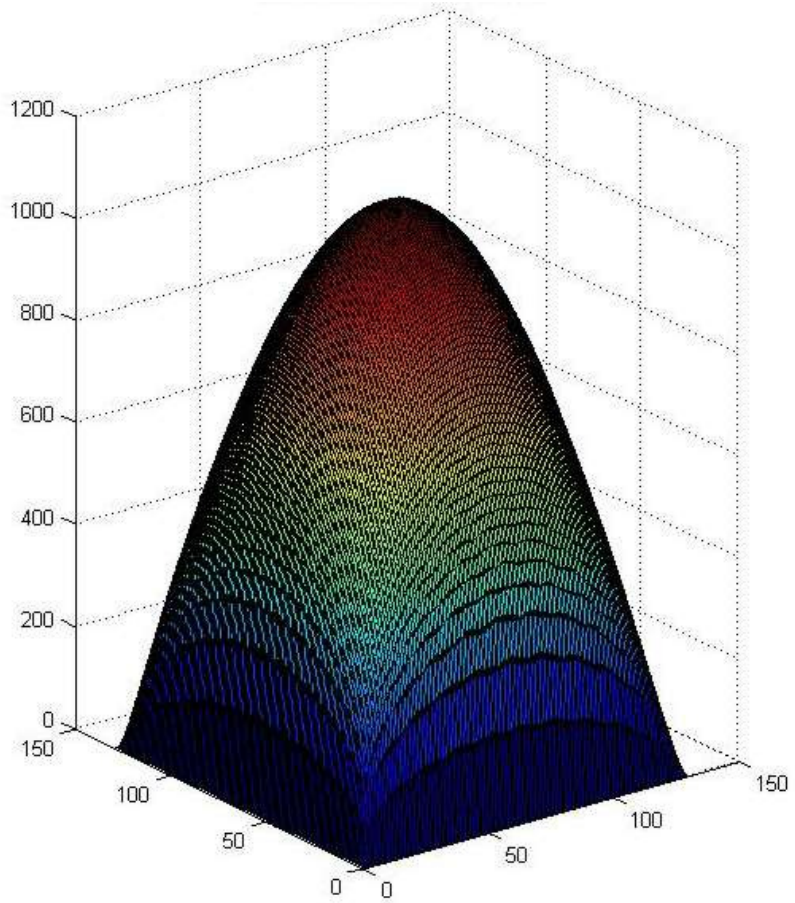
(Actually N = 128, P = 10, on UCSD's Valkyrie)



100 iterations (err = 0.8127)



1,000 iterations (err =0.4729)



10,000 iterations (err = 0.02796)

Future Work

- Change serial algorithm code to mimic a parallel algorithm resulting in better cache use and performance
- Reduce communication by lowering the alpha term using double boundary passing every other iteration.
- Allow users to view the converging solution on the cluster from a local machine using Mike's SG (Socket Graphics) OpenGL tool.
- Finish debugging my second algorithm that uses a multigrid preconditioner.

References:

Scott Baden's (UCSD) CSE260 Parallel Computation

Introduction to Parallel Computing 2nd Ed Grama, Gupta, Karypis, Kumar

This power point presentation can be downloaded off my website: math.ucsd.edu/~cdeotte/