Chris Deotte CSE260 Baden
CSE 260 Research Project Due 12/05/08

The Time to Pass Messages Limits Scalability

Project Proposal (10/21):

Goal: To determine the limiting effect that the time to pass messages poses and devise a formula for the optimal number of processors to employ when subdividing an iterative process. Also I wish to investigate a way to surpass this limiting barrier.

Hypothesis 1: Suppose multiple processors are working on an iterative process. When the time it takes to pass required information between iterations equals the time it takes to compute an iteration, then we are wasting current processing power and adding more processors will not speed up the process.

Hypothesis 2: Passing extra information that allows a process to complete more than one step of a boundary dependent iteration can speed up the process if it is operating at close to the efficiency barrier as defined in hypothesis 1.

Motivation: Solving a partial differential equation by an iterative finite difference method benefits greatly by parallelization. Poisson's Equation, $\sum_{i=1}^{D} \frac{\partial u}{\partial x_i} = b(x_1, x_2, ..., x_D)$, can be solved by discretizing a domain into a set of points, $\{x_i\}$ for $i = 1,2,3,...,N$, making an initial guess on this set $u(x_i) = y_i$ for $i = 1,2,3,...,N$, and then calculating a new value for each point based on old values. Here is psuedo code demonstrating this method:

Let N be the number of points in discrete domain
for (i=1 to N){

$$U_{NEW} = \left(\sum_{j=1}^{2 \cdot D} U_{OLD}(\text{neighbors}) - b \cdot h^2 \right) \Big/ 2 \cdot D$$

} //end for loop.

Above, *D* is the dimension of the domain, and *h* is the distance between adjacent points. If there are N points to update, one can see how M processors can be employed to solve this task. Just give N/M points to each processor to calculate their new values (special consideration is required if *M* doesn't divide *N*). It seems like we could continually speed up this serial process as *M,*the number of processors approaches *N,* the number of points. This may be the case if the different processors didn't need to pass information to each other. But, since the processors need to communicate, even if a processor can complete its work in no time, we must still wait for the processors to communicate between iterations. Also, as you add more processors, you increase the surface to volume ratio thus increasing the percent of communication to overall runtime. So the question arises, when does adding more processors not increase overall completion time? And are there any ways around this barrier?

Significance of This Question: Using extra processors that are not required has two disadvantages. One, we are wasting processor time which wastes energy and money. And second it may be the case, that extra processors slow down the process due to too much communication per computation.

This question is of utmost importance to the Multigrid procedure. In this procedure, the number of points N_0 is reduced to N_1, then N_2, ... down to N_k during runtime and the coarser mesh is solved at various times. If the program began with *M* processors operating efficiently on N_0 points, will *M* processors still be efficient on N_k points? To answer this question, it would be nice to have some sort of rule to allow us to know how many processors are optimal.

Experimental Method: In order to answer this problem, I will write C++ code utilizing MPI to solve Poisson's Equation in 2 Dimensions. I will then run it on a parallel computer and carefully record performance versus different processor geometries. I will then correlate this information with variables present in the problem such as mesh size, computational complexity, and amount of required boundary information needing to be passed. With this data, I will come up with a rule to determine optimal processor geometry for a given problem's parameters.

Next I will code Multigrid again using C++ and MPI and test out the rule by using it to reduce the parallelism during runtime.

Finally, I will experiment with passing extra boundary information between iterations and having each processor calculate more than one iteration. For example, instead of just passing a square's boundary of 1 point width, pass a 2 point width boundary. Then a given processor can use the bigger boundary in its first iteration to compute its 1 point width boundary to be used during its next iteration. This way, processors need only communicate every other iteration. I will see if this trick allows me to employ more processors on a given mesh size *N*, surpassing the previous restriction found, and obtaining faster performance.

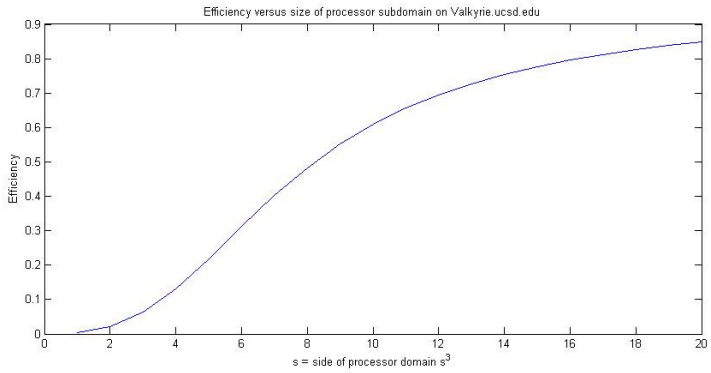
Project Progress (11/13):

Prediction: If an iterative process needs to compute *n* work units of computation per iteration, then it naively seems that we can achieve perfect efficiency by dividing these units over *p* processors. It is true that *p* processors can do *n* units of work in n/p time. And if our parallel algorithm didn't need to communicate, than efficiency, $E_p = T_1 / P \cdot T_p = 1.0$, would equal one. However, if one process needs information from another process than we need to factor in the time required for communication. And, this communication will limit our efficiency.

Theoretically, efficiency, $E_p = \dfrac{T_1}{P \cdot T_p} = \dfrac{P \cdot T_p^{\text{comp}}}{P \cdot \left(T_p^{\text{comp}} + T_p^{\text{comm}} \right)}$ where T_p^{comp} is the time needed to compute the updated grid points of a single processor's subdomain, $T_p^{\text{comp}} = N^3 T_\gamma / P$ where T_γ is the time needed to update a single point. And T_p^{comm} is the time needed for one processor to send and receive information to/from its neighbors. Therefore $E_p = \dfrac{T_p^{\text{comp}}}{T_p^{\text{comp}} + T_p^{\text{comm}}}$. Also if we divide the original 3-D domain into

equally sized cubes, then $T_p^{\text{comm}} = 6(\alpha + 8s^2\beta)$, and $T_p^{\text{comp}} = s^3 T_\gamma$ where $s = N/\sqrt[3]{P}$. The variable *s* is the edge length of a processor's 3-D subdomain of s^3 points.

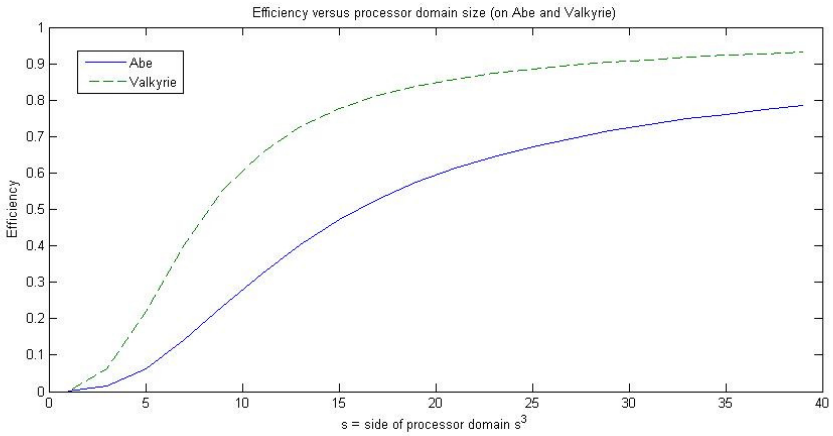
On Valkyrie, I have found experimentally $\alpha = 1.37\times10^{-5}$ seconds, $\beta = 1.182\times10^{-8}$ seconds, and $T_\gamma = 2.165\times10^{-7}$ seconds. Therefore theoretically, $E_p(s) = \dfrac{1}{1 + \dfrac{6\alpha + 48s^2\beta}{s^3 T_\gamma}}$. Below is a plot of this equation.



This theory implies that on Valkyrie, it would be wasteful for a single processor to process a subdomain smaller than number of points = 14^3, otherwise it's efficiency drops below 75%.

Let's analyze this inefficient domain of $s = 14$ having 14^3 points and see where the inefficiency resides in hopes of improving it. For $s = 14$, $T_p^{\text{comp}} = s^3 T_\gamma = 14^3 \cdot 2.165\times10^{-7} = 5.941\times10^{-4}$ and $T_p^{\text{comm}} = T_p^{\alpha-\text{comm}} + T_p^{\beta-\text{comm}}$ $= 6\alpha + 48s^2\beta = 8.22\times10^{-5} + 1.11\times10^{-4}$. Therefore on Valkyrie, the inefficiency lies about equal in the alpha and beta term. If we send twice the data every other iteration, we can effectively reduce the alpha term by two thus improving efficiency by 5.5% $= \left(\dfrac{5.941}{5.941 + 0.411 + 1.11}\right) \Big/ \left(\dfrac{5.941}{5.941 + 0.822 + 1.11}\right) - 1$ when $s = 14$. But if we send information while we are computing, we can perhaps remove the beta term completely and improve efficiency by 16.5% $= \left(\dfrac{5.941}{5.941 + 1.11}\right) \Big/ \left(\dfrac{5.941}{5.941 + 0.822 + 1.11}\right) - 1$ when $s = 14$.

I conducted a similar analysis of Abe and found that on Abe, $\alpha = 1.7\times10^{-6}$ seconds, $\beta = 1.27\times10^{-9}$ seconds, and $T_\gamma = 6.33\times10^{-9}$ seconds.



Theoretical results predict that $s = 33.5$ has the undesired efficiency of 75%. For this $s = 33.5$, $T_p^{\text{comp}} = s^3 T_\gamma = 2.38\times10^{-4}$, and $T_p^{\text{comm}} = T_p^{\alpha-\text{comm}} + T_p^{\beta-\text{comm}} = 1.02\times10^{-5} + 6.84\times10^{-5}$. Therefore on Abe, the inefficiency lies over 6 times as much in the beta term as the alpha term. Sending twice the information every other iteration would improve our efficiency by 1.0% while removing the beta term will improve our efficiency by 27.5%.

The theory above suggests that for a fixed domain size N^3, we do not want to divide the work among more than a certain number of processors. For Valkyrie, the maximum number of processors to employ on a domain of N^3 points while maintaining good efficiency is predicted to be $P_{\max}^{\text{Valk}} = (N/14)^3$, or if you fix P then N shouldn't go below $N_{\min}^{Valk} = 14\sqrt[3]{P}$. For Abe that number is $P_{\max}^{\text{Abe}} = (N/33.5)^3$ for fixed N or $N_{\min}^{Abe} = 33.5\sqrt[3]{P}$ for fixed P. As I mentioned in my proposal, this has significant consequences for multigrid. Multigrid repeatedly solves an original problem with a given domain size on repeatedly coarser domains. Special consideration is required when N_{coarse} becomes less than or equal to $N_{\min}$. We now have two questions to investigate. How close does theory predict reality? And, is there a way to lower $N_{\min}$?

Project Research / Results (12/05):

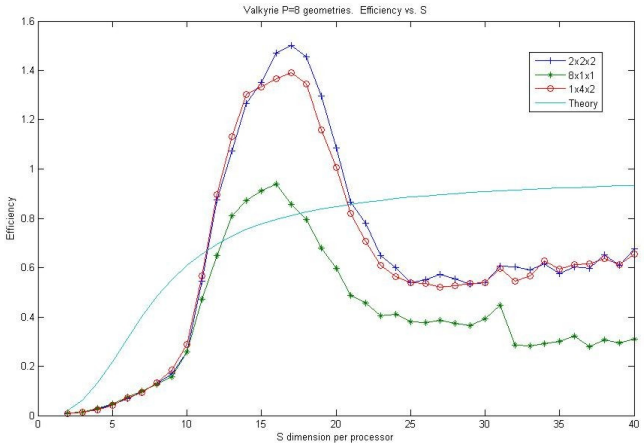
Goal 1: To determine the limiting effect that the time to pass messages poses and devise a formula for the optimal number of processors to employ when subdividing an iterative process.

To determine the lower bound on processor domain size and find a prediction equation, it seems the experimental method is straightforward. Write a program, run it for a variety of *N* values while holding other variables constant, and plot the efficiency. The plot should look like the theoretical graph, and then our prediction equation is just the one from above.

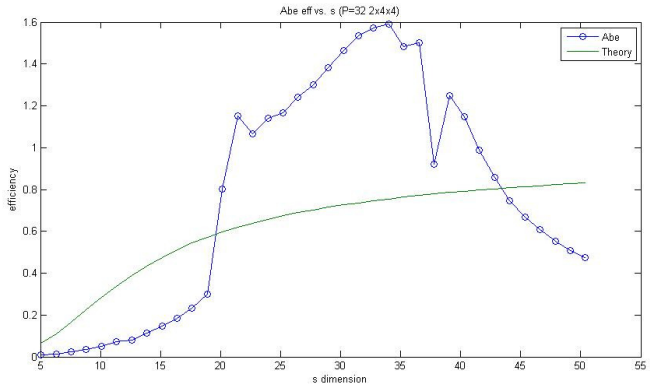
As you will see, reality didn't follow the above simple model. As a result, much was learned and even some unexpected results were discovered.

After finishing my program to solve Poison's Equation, I first ran it on a variety of different processor geometries with the same initial conditions and iteration counts. Each run produced the same L-infinity error. From this I concluded that the program ran accurately. I also had it solve some real life problems and visibly inspected the solutions to see if they matched analytic solutions and intuition. Again my code passed these tests. Some sample pictures are presented later in this paper.

I ran my first experiment on Valkyrie. I fixed the number of processors to 8 and repeatedly solved the Poisson's Equation for different values of *N*. I also solved the same problem with my best serial solver. I ran each trial 3 times and selected the minimum run time. I repeated this procedure for 3 different geometries, $1\times4\times2$, $8\times1\times1$, and $2\times4\times1$. Here is a plot of the results:

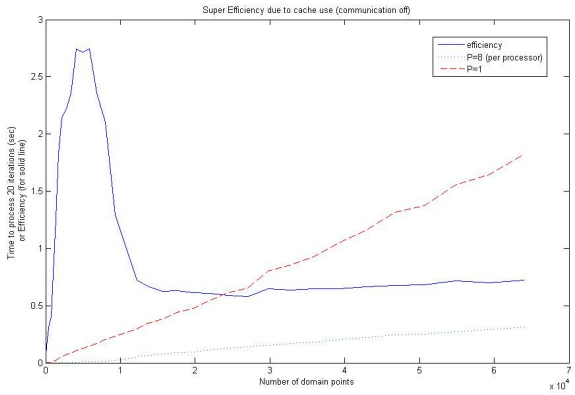


Whoa! This isn't what should happen! Where is the nice gently rising curve? I see two things that are unexpected. First there is a bump in the actual data around $s=15$, and second the efficiency is real low after the bump, $E_p \approx 0.60$ for $s \geq 25$. Thinking and hoping these results may be a fluke with Valkyrie, I ran the same experiment on Abe only to get the same results:



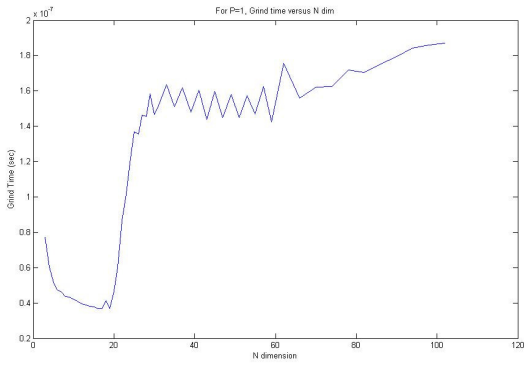
After seeing this graph a second time on a different machine, I figured my program exhibits this behavior. Now I need to figure out what is causing this bump before I can come up with my predictive model.

I decided to plot the efficiency with communication turned off. I figured that this plot should be a straight line at efficiency = 1. I ran my code on Valkyrie's best 8 processor geometry, $1\times4\times2$, for each of the same *N* values that I used above.

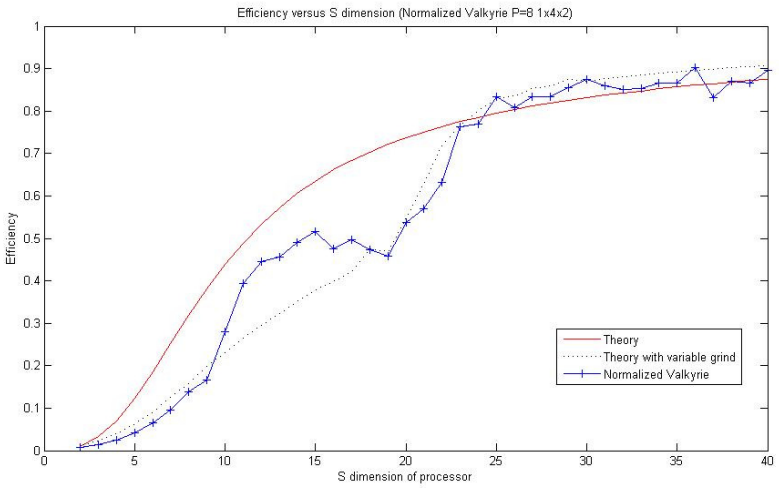


Well this is interesting. We're getting to the cause of the bump. The multiple processors are accomplishing super efficiency when an individual processor has domain edge dimension *s* with $10 \le s \le 20$ corresponding to a processor's number of domain points of $1000 = 10^3 \le dp \le 8000 = 20^3$. When a processor's domain is under $64\text{kb} = 8 \cdot 8000$, it runs much faster than the serial algorithm which is working on a domain size 8 times as large, maybe $512kb$. I suspect this is because each of the eight processors are working out of cache memory while the serial process works from RAM.

This is a big discovery. This gives me an idea to write a better serial algorithm for the Poisson solver. Instead of storing the original $N\times N\times N$ data structure in a single variable residing as a continuous block of memory, we should sub divide this domain into smaller cubes each with total size less than cache memory. Then store each cube in its own variable. This will preserve better locality in memory for the computation loop which requires info from 6 neighbors in different 3-D directions. Since I don't have time to code this better algorithm now, I will solve for the varying grind time of my current algorithm and normalize it out.



Here is a plot of the normalized efficiency:



Now it looks closer to theory. The normalization fixed two things; first, it lowered the bump and second, it raised the low efficiency region after the bump. Why is the unnormalized efficiency with communication turned off only around 60% after the bump, $s \geq 25$? Why isn't it 100%? This low efficiency will also interfere with our predictive model.

The theoritical graph is a function of the time to update one point (grind time), beta, and alpha. To plot the theory graph above, I fixed T_γ, α, β to a constant value corresponding to the actual value at one instance of *N*. If I use the variable calculated T_γ, the new theory line becomes the dotted one above.

We've nearly explained everything now. The only deviation of reality from our developed theory is a small bump where the old large bump used to be. Using measured values of efficiency, we can solve for what beat needs to be in the theory to cause this small bump. It turns out beta would need to be negative. This means the small bump is caused by beta and alpha being lowered for $10 \leq s \leq 20$. We've discovered that these cache optimal *s* values obtain an alpha and beta better than the ring program!

<u>Goal 1's Conclusion:</u>

In conclusion, our original predictive model is alright if the original algorithm has been optimized for cache use. Otherwise, you need to factor in an increase of efficiency proportional to the ratio of RAM access times to cache access times for $s_L \leq s \leq s_H$ where $s_H = \sqrt[3]{\text{cache size}}$ and s_L seems to be the value where $E_p = 0.50$ in our theory equation above (I don't know why it just seems like that). Furthermore you need to divide the predictive model by the leveling off efficiency of your algorithm when *N* is large.

If $E_p(s) = \dfrac{1}{1 + \dfrac{6\alpha + 48s^2\beta}{s^3 T_\gamma}}$ and E_L = leveling off efficiency and $B(s) = \left\{ \begin{matrix} \text{RAM access/cache access} & s_L \le s \le s_H \\ 1 \text{ otherwise} & \end{matrix} \right\}$ = bump function. Then $\widetilde{E}_p(s) = E_p(s) \cdot B(s) / E_L$ and you shouldn't let your number of processors go below $P_{\max} = (N/s_k)^3$ where $s_k = \widetilde{E}_p^{-1}(0.75)$.

Goal 2: Also I wish to investigate a way to surpass this limiting barrier.

After seeing the data collected and reading a recommendation and paper from Professor Baden, I have decided to exceed the limit rule if a different way than proposed in the beginning of this report. Communication time equals alpha plus beta times *x*. Alpha is the message start up time, beta is the time needed to pass one byte of data, and *x* and the length of the data. Instead of trying to reduce the alpha term by passing extra boundary information and doing multiple computation iterations between communications, I will attempt to reduce the beta term by passing information while at the same time doing computation.

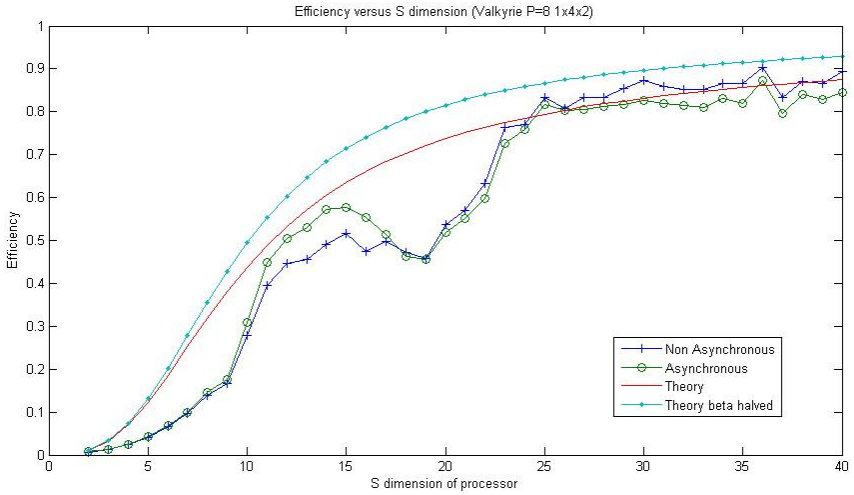
Currently each processor performs the following sequence of computation and communication:

For i = 1 to number of iterations desired
    Compute new points
    Transmit boundary to neighbor processors
    Receive boundary from neighbor processors
Repeat

An alternative approach is to transmit data while computing:

For i = 1 to number of iterations desired
    Begin transmission of boundary points.
    Compute new points excluding boundary points (transmission in progress)
    Receive boundary from neighbor processors
    Compute new boundary points using data from neighbors
Repeat

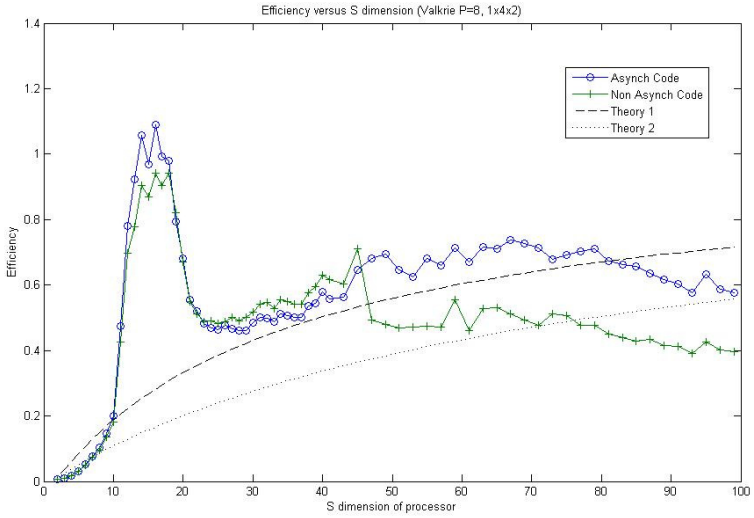
This will lower the beta constant which hopefully will allow us to use more processors for a given iterative task. After coding the above psuedo code, I tested it for numerical accuracy against our old non-asynchronous algorithm. Given the same problem, it solved it exactly the same as the other algorithm. You can turn off this new feature at the command line with the '–noasy' option. Next I plotted and normalized the efficiency of this new algorithm for $0 \le s \le 40$ where *s* is a single processors domain edge corresponding to a sub domain size of s^3. I ran it on the same $P = 8,\ 1 \times 4 \times 2$. Again, this geometry was chosen since it is the best performer for 8 processors.



As you can see, it does increases the efficiency by about 15% in the bump region as was predicted by the theory in the beginning of this paper. The change in the shape of the graph is what one would expect. When you decrease beta, the graph rises faster (making the graph to appear to shift left sort of) as you see here. To demonstrate this, I also plotted another theory line with a decreased beta so you can see how changing beta changes the graph. It doesn't appear to allow us to employ more processors though. I guess this is to be expected. We can't begin rising beta before $s \approx 10$ for $0 \le s \le 10$. Before this *s* value, we are communicating more than computing so we can't overlap all our communication on top of our computation. Since we begin rising beta so far right on the *s* axis, we don't move the graph too much to the left. Maybe we could shift the graph left a little by lowering alpha. That is something to try. The only other way to modify the shape of the graph is to change the grind time, but it would require a lower grind time to shift the graph to the left and that is undesirable.

Goal 2's Conclusion:

In conclusion we can not really increase $P_{\max}$ (lower s_k) by altering the beta term in the above manner. We are stuck with our original upper bound on the number of processors to use. All is not a loss though. I discovered that using a communication and computation overlap has great results when the efficiency is higher and the computation time is greater than the communication time. That is when we can effect beta the most since we can completely overlap transmission with computation. Here is the continuation of the above plot to $s = 100$. As you can see efficiency is greatly increased in this region.



the above graph, I've also plotted two theoretical plots the first one has a beta value calculated from actual communication when $N=100$. This beta is much lower than the one calculated above by the ring program which describes the behavior of the code running with domains that fit into cache memory. For $N=100$, I experimentally found $\beta=1.69\times10^{-7}$. This was derived from running $N=100$, $i=1000$, $P=8$, $1\times8\times1$, non – asynchronously and getting a run time of 40 seconds, then turning off communication and getting 17.5 sec, then running $i=900$ and getting a run time of 36.3 seconds, and turning off communication and getting 15.8. Therefore $(40-17.5)-(36.6-15.8)=100\cdot8\cdot100^2\beta$ implies $\beta=1.69\times10^{-7}$.

The asynchronous version can solve the same $N=100$, $i=1000$ problem in 24 seconds! That's nearly half the time. But it makes sense because the non-asynch process in running at 50% efficiency which means it spends half its time communicating. The asynchronous version essentially removes all this communication time. Amazing!

Application and Further Investigation:

Now let's incorporate our results into a multigrid algorithm. Multigrid is a method that is used to accelerate an interactive process convergeing upon a solution. For most iterative algorithms updating a mesh of domain points, the *N* domain points are the discretization of some space. In our Poisson solver, our points are the discretization of three dimensional space.

You can think of solving a differential equation with an iterative solver as two parts. One part is to bring the intial guess into the ballpark of the solution and the second part is the fine tune this solution. Basically multigrid allows you to converge into the ball park in no time. Imagine you are solving for an object's position between times 0 and 1 seconds. If you break this interval into one million or so pieces, you will need to solve for the solution on a domain of $N=1,048,576$. If you initial guess is zero

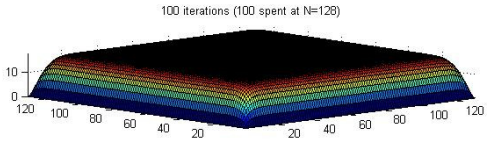
everywhere but the true solution is 10 everywhere, it may take a while to converge. Multigrid solves the problem on coarser domains to find the ball park. In the above example, multigrid will attempt to find the solution for the domain broken into $N_2 = 524,288$ and then to find that it looks at $N_3 = 262,144$. It continues this reduction recursively until it only needs to solve on a very small domain like $N_{19} = 4$ or something. Now it find the position for at time equals 0, 0.25, 0.50, 0.75, and 1.0 seconds instead of finding it at one million time moments. It solves this in no time and finds the solution to be around 10. It then begins to fill in the details between these 8 domain points until it has the final answer on the original $N = 1,048,576$. This is analogous to a painter sketching an outline of a drawing and then filling in the details versus working in one corner and beginning in high detail. The first method is much easier and faster because you have the big picture to work with.

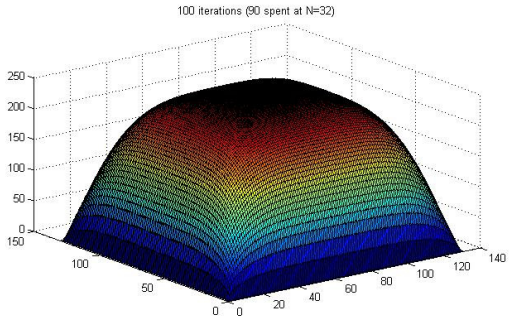
Of course multigrid in practice can become much more complicated than this. An algorithm can continually cycle in and out of high and low resolution, varying the time spent solving at different levels. Again, this is what a painter may do.

Let's solve $\sum_{i=1}^{3} \frac{\partial u}{\partial x_i} + b(x) = 0$ with $b(x) = 1$ and $\partial\Omega(x) = 0$. The last equation says that our solution *u* has value zero on the boundary. Let's solve this on the unit cube subdividing this domain into $128^3 = 2,097,152$ points. In order to solve this, I coded multigrid into our Poisson solver and then tested it for accuracy. On the command line of Jacobi3D_mpi, add the switch –mg <x> <y> to make the program begins with a multigrid v-cycle. Once Jacobi3D begins, it runs two initial iterations of Jacobi's method, then the v-cycle will descend *x* levels pausing for 2 iterations at each. At the coarsest level, it will run for *y* Jacobi iterations before climbing back to the original *N*. On its way up, it will stop at each level to run one Jacobi iteration.

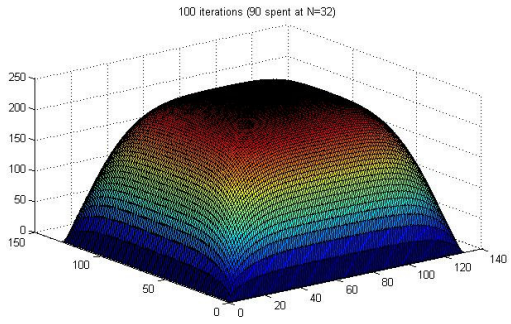
In addition to the multigrid switch, I also added a new feature to allow us to visualize results. By adding a command line option of '-o', then after a solution is found a cross section of the solution slicing the z-plane in the middle is saved to a text file. We can then plot this slice with matlab. This will help us see the difference between using and not using multigrid. The final solution to the above partial differential equation is an upside down parabolic bowl of height 1000. We will start with an intial guess of $u(x) = 0$ which is way off.

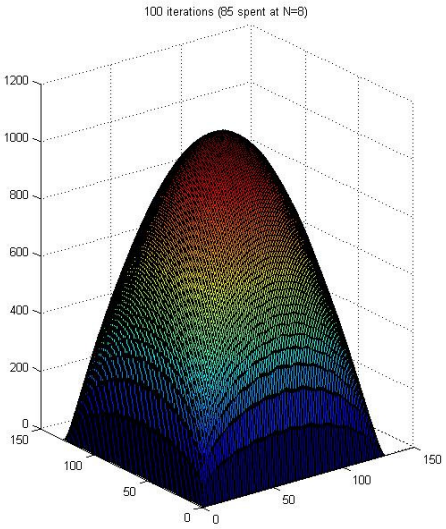
Multigrid will repeatedly solve on a coarser domain, $64^3, 32^3, 16^3$, etc. If we start with 8 processors on Valkyrie and would like to stay operating above 75%, our research work above tells us that we shouldn't let *N* go below $28 = 14 \cdot \sqrt[3]{8}$. That means we will cycle down two levels spend some time there and then cycle up. Here is a plot without multigrid on the left and with multigrid on the right.





You can see, multigrid is accelerating our approach to the solution. As described above this is because one iteration on the coarser mesh brings the big picture of our solution much closer to the actual solution than one iteration on the finer mesh. If that is the case, then wouldn't it be better if we descended into an even coarser mesh, ignoring our inefficiency restriction? Below on the left is our 2 level descent and on the right is a 4 level descent down to $N = 8$ with each processors getting $s = 4$, 16 points.





Yes, our previous research and theory says that we are operating inefficiency on the coarse mesh when obtaining the above solution on the right. But, our time on the coarsest mesh is such a small proportion of our overall run time because most of our time is spent on the fine mesh. Therefore inefficiency on the coarsest level doesn't really make a difference. Furthermore, you can see from the plots that we don't want to avoid using the coarsest levels. It is also worth noting that the upper right picture took 0.45 seconds as opposed to the upper left of 0.71 and the non multigrid plot of 4.3 seconds.

If your application requires you to spend the majority of your time on coarser meshes, and, if those meshes will have $N < s_{\min}\sqrt[3]{P}$ where $s_{\min}$ is the minimum edge dimension of a processors subdomain as described in the above paper, then you will benefit from reduced parallelism. Once you arrive at the coarser mesh, you can reduce parallelism by turning off some of your processors and passing their domain points to another processor, thus increasing the remaining processor's *s* dimension and increasing efficiency.

Increasing efficiency has two benefits. One, you save money by getting more performance from the money you spend on processor time, and two you actually decrease running time under certain circumstances. If $E_p(x)$ represents the efficiency as a function of *x,* the number of processor domain points, then whenever $\frac{\partial E_p(x_0)}{\partial x} \cdot x_0 > E_p(x_0)$, then removing processors actually speeds up the process!

From our collected data above, you can see that on Valkyrie, $\frac{\partial E_p(x)}{\partial x} \cdot x - E_p(x) = 0.26$ for $10 \le x \le 14$ therefore if we are ever operating with $10 \le s \le 14$ then we can speed up our process by removing processors. For example if we are employing 8 processors with 1000 domain points each, then this $s = 10$ is causing us to run at 0.2 efficiency (from the data above). If we drop down to 4 processors with 2000 domain points each or $s = 12.5$ and a new efficiency of 0.66. We first decrease our speed by 2 and then increase it by 3. We would gain a net speed increase of 50%.

To prove this is true I ran the above example on Valkyrie, it takes 8 processors 1.29 seconds to compute 10,000 iterations of $N = 20$ and it takes 4 processors 0.981 seconds to complete the same iterations. The four processors finished 31.5% faster.

Final Conclusions:

In the first part of the paper we came up with a formula for the maximum number of processors to deploy when running a parallel process to solve an iterative computation. Specifically we found $P_{\max}^{\text{Valk}} = (N/14)^3$ to be the maximum for Valkyrie and $P_{\max}^{\text{Abe}} = (N/33.5)^3$ for Abe. We also came up with a general formula, $P_{\max} = (N/s_k)^3$ where $s_k = \widetilde{E}_p^{-1}(0.75)$ where $\widetilde{E}_p(s) = E_p(s) \cdot B(s)/E_L$.

The derivation of this general formula demonstrated the complicated nature of efficiency for processors running very small subdomains. When we analyzed Valkyrie we discovered a bunch of peculiar behaviors which we then identified. These behaviors were the result of our specific code and Valkyrie's memory architecture, and networking properties. The complexity and dependency suggest that it is probably more reliable when approaching a new machine and new code to run some tests and experimentally determine efficiency versus processor edge dimension instead of using theory before deciding how and when to reduce parallelism.

Another big discovery resulting from investigating the behavior of small processor subdomains is the importance of cache use. Everyone knows about cache use but the data above really demonstrates its importance. I learned how to best optimize a serial algorithm that has heavy reliance on memory data representing a multidimensional domain. The code should essentially behave like a parallel

implementation. An optimal serial algorithm should repeatedly solve the problem on data structures that have less than cache size and piece the overall domain together.

In an effort to increase $P_{\max}$, we learned that asynchronous communication with computation doesn't help much to shift a graph left where the beginning of that graph represents a process that is mostly communication and not much computation, i.e. $E_p \leq 0.5$. We did discover however that asynchronous communication radically increases efficiency when a process is doing computation and communication nearly equal, i.e. $E_p = 0.5$. This was the case in our original program when $10 \leq s \leq 14$ and $s \geq 40$, and we witnessed the amazing results.

Finally, we learned that you need to be careful when interpreting and applying the theory developed from this research. It was exciting to see everything come together in our multigrid implementation. We had our faster asynchronous code working together with the accelerating power of multigrid resulting in truly powerful computation! We identified the lower levels of multigrid as processor inefficient but experimentally learned that they shouldn't be avoided. They are the heart and secret to the power of multigrid. Analyzing run times at this coarser levels, we also saw how little time multigrid actually spends there as a fraction of overall runtime. Regarding the multigrid process, theory may suggest to avoid the coarsest levels or redistribute the domain when a processor's responsibility drops below $(s_{\min})^3$ points. However, experiments demonstrate that we should use the lowest levels anyway and it isn't necessary to spend time and money on developing code to reduce parallelism (unless *px, py,* or *pz* is less than the coarsest N)