Improving Performance of the ADCIRC Hydrodynamic Model

E.P. Gilje

R. L. Kolar, K. M. Dresback, and J.C. Dietrich

Environmental Modeling/GIS Laboratory (EM/GIS)
School of Civil Engineering and Environmental Science
University of Oklahoma
Norman, OK 73019

Technical Report No. 02-01

(Keywords: shallow water modeling, GWC equation, finite elements)

Abstract

This project's goal is to improve performance of the ADCIRC hydrodynamic Model. Two main areas of research were conducted. The first area involved a benchmarking study on a Linux cluster. These results were then compared to benchmarking results from a SUN cluster. To get a complete view of ADCIRC's performance both the job size and number of processors being used were varied both individually and simultaneously. By doing this conclusions were able to be drawn about the communication costs between processors, the effect of cache, as well as how the type of grid effects the ADCIRC run. The second area of research tried to further enhance the efficiency of ADCIRC by adding the NetCDF file type to ADCIRC. Results from this led to conclusions about how much time and space could be saved by ADCIRC by using different file types.

Introduction

The ADCIRC (ADvanced three-dimensional CIRCulation model) hydrodynamic model was initially created by Westerink and Luettich for the Army Corps of Engineers to aid them with dredging. However, ADCIRC can also be applied to a variety of other applications ranging from modeling hurricane storm surges to modeling current patterns in coastal areas. ADCIRC uses shallow water equations to model the hydrodynamic behavior in these varied applications (Kolar et al 1994). Both the wave continuity equation (WCE) and the generalized wave continuity equation (GWCE) improved the finite element solutions to these equations. Lynch and Gray (1979) introduced the WCE to suppress the spurious oscillations inherent to the primitive equations without dampening the solution either artificially or numerically. Kinnmark (1986) introduced the GWCE an d replaced the bottom friction tau with the numerical parameter G.

In non-linear applications ADCIRC encounters stability problems unless a severe Courant number restriction is imposed. For stability of the model the upper bound of the Courant number is set at .5, this would need to be further tightened if the simulation contained barrier islands and constricted inlets. In order to combat this restriction an alternative time-marching algorithm procedure was implemented so that some or all of the non-linear terms are treated implicitly (Kolar et al 1998). This algorithm is called the predictor corrector algorithm and is able to increase the maximum stable time step in both one and two dimensional applications (Dresback et al. 2001; Dresback et al. 2000). Thus this benchmarking study will evaluate the behavior of both this predictor corrector algorithm and the original algorithm.

Although the predictor corrector algorithm can greatly speed up the run time of ADCIRC, output file sizes can still be immense. Currently ADCIRC I/O can be of two file types, binary and ASCII. The binary file type is relatively small as well as fast, however, it is platform dependent. ASCII text is relatively slow and large and is not platform dependent. The NetCDF file type will hopefully provide the best of both worlds with faster output, smaller file sizes, and platform independence

Materials and Methods

For this study a Linux cluster located in the Computer Science Department at the University of Oklahoma was used. Benchmarking times from the Linux cluster were compared with times from a SUN cluster located in the GIS lab located in the Department of Civil Engineering and Environmental Science at the University of Oklahoma. Here is a table to compare the main components of the Linux and SUN clusters. On the right is the Linux cluster which uses Intel Pentium III chips, on the left is the Sun cluster which uses Sun. UltraSparc IIe chips.

Table 1. Sun Cluster vs. Linux Cluster

Component	Sun UltraSparc IIe	Intel Pentium III
Speed	500 MHz	1GHz
Cache	256 KB	256KB
Memory	128 MB	256 MB
Communication	100 Mb/s	100 Mb/s
Compiler	Sun Forte 6.0	NAG
MPI	Sun Cluster Tools	MPIch
Total Cost	$19,300	$27,500

In this case benchmarking runs consisted of either varying the number of processors for the job, the size of the job, or both. For the east coast grid and the quarter-circle harbor grid the number of processors for the grids were varied. Each ADCIRC grid consists of a certain number of vertices and it will calculate data at each of these vertices. For the Q-series benchmarking both the number of vertices and the number of processors was varied. In order to measure the cost of communication between processors the Q-series consists of runs where 5000 =(the size of the job)/(number of processors). By doing this each processor will have the same number of vertices assigned to it regardless of the number of total processors doing the job. Single processor runs where the number of vertices were varied were also done with the quarter-circle harbor grid to examine the work per unit time of the processors. Each set of runs except for this last one were done using both the original code and the predictor-corrector code, these run at their respective

maximum stable time steps, for the predictor corrector code the maximum stable time step was many times higher than for the original code. One last note on how the benchmarking was conducted, the actual times recorded for each of the runs are the wall clock time which was received from the "time" command and were compiled with the -O flag (Appendix B)

For the NetCDF aspect of this project the NetCDF libraries were obtained from the website of the University Corporation for Atmospheric Research. The software is free and all installation instructions are provided on the site. An online manual with coding and other instructions is also on the site. In order to initially test out the effectiveness of NetCDF it was first implemented in a 1-Dimensional version of the ADCIRC code, after this it was then implemented on a 2-Dimensional example. In the 2-Dimensional code NetCDF modifications were made in four main areas: a) the adcirc.F file, the timestep.F file, the global.F file, and where the input is read in. In the adcirc.F file the NetCDF data set was created, in the timestep.F file variables were stored in the NetCDF file and in the global.F new variables were declared for use in the NetCDF modifications. Finally the last modification that was made was where the input was read in, here an additional option was set where a user could enter in the number 3 in their input file and they would get netcdf output, previously, only 1 and 2 (binary and ASCII) were considered valid. A rather detailed explanation of some basic NetCDF coding for Fortran 90 is included in Appendix A.

Results

East Coast Grid Results

Figure 1.1 - Unfortunately results that included all 16 processors could only be obtained for the predictor corrector SUN code. One processor on both the Linux and the SUN cluster went down and were unable to be repaired for this study. However, general trends can still be seen from the data that were collected. All of the Linux cluster runs were faster than the SUN cluster runs with the notable exception of the single processor run. Predictor-corrector runs also always ran faster than runs that used the original code. Finally, as you increase the number of processors used in the runs there is always a decrease in time.

Figure 1.2 - For better visualization of the relationship just discussed this graph shows the speed-up of the results from the east coast runs. What this graph

Figure 1.1 - Wall-Clock Time vs. Number of Processors
East Coast Grid, ~30K Nodes, Maximum Stable Time Step

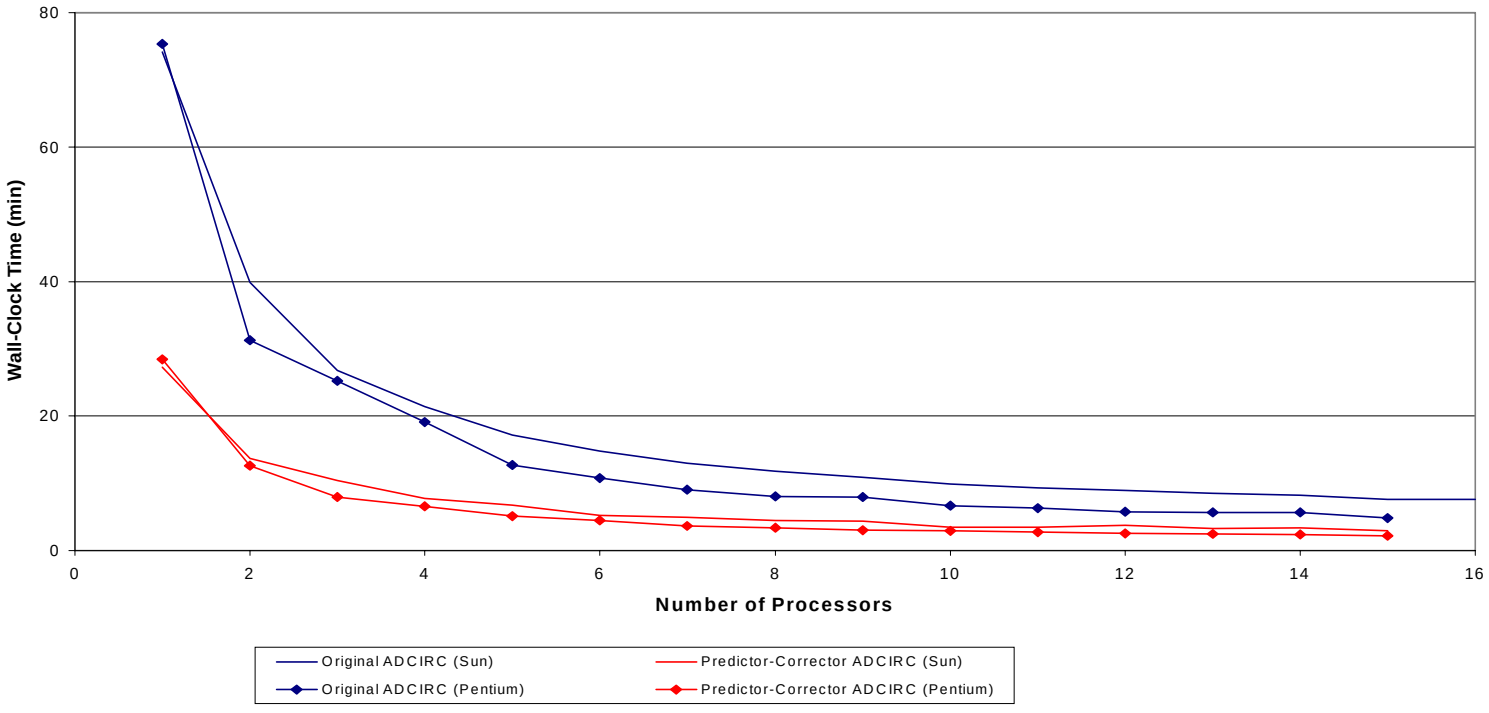
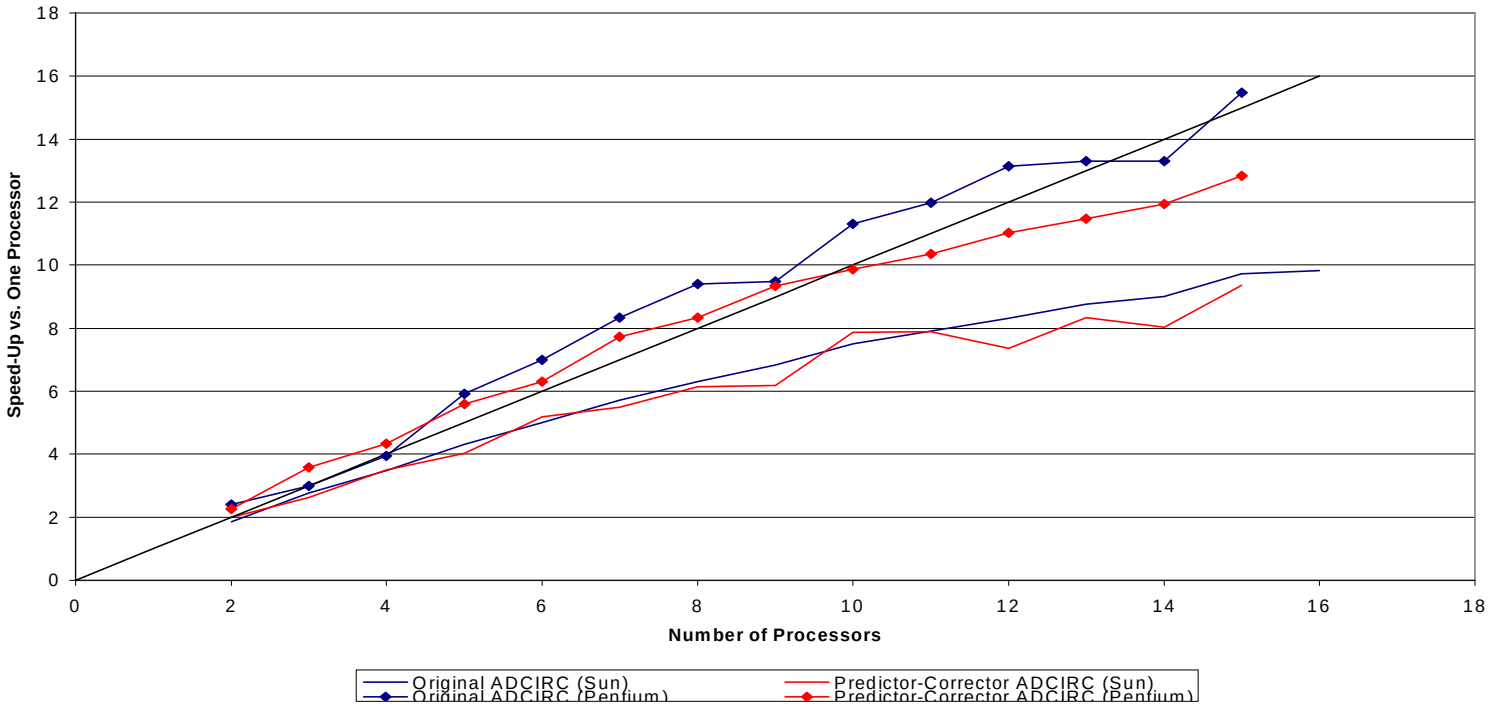


Figure 1.2 - East Coast Grid
Real-Time Speed-Up Compared to One Processor



attempts to do is to show what effect adding one processor to the run affects the run-time. In addition to the results from the runs there is a theoretical speed-up line. This line represents what the theoretical speed up should be. For example, if a run takes ten minutes on one processor than theoretically it should take five minutes on two processors. This line represents this theoretical relationship.

Linux cluster results for the original code had a super-linear speed up for all runs except the run on 14 processors, thus doing better than it theoretically should have on most runs. Linux cluster results for the predictor-corrector code were super-linear until about 10 processors, at which point they became sub-linear. SUN cluster results did not have as good a speed-up, they were sub-linear through all processors for both the predictor-corrector and the original pieces of code.

Quarter-Circle Harbor Grid Results

Figure 2.1 - Much of what was said of the east coast grid results apply to the quarter circle grid results but there were some notable exceptions. Linux cluster times always beat SUN cluster times even on one processor which the SUN cluster had done better on in the east coast grid results. However, as was the case with the east coast grid results the predictor corrector times were always better than the original times. Here again, though, some technical difficulties were encountered, predictor corrector results on both the SUN cluster and Linux cluster on eleven processors were unable to be obtained, ADCIRC would continuously crash when attempts at these runs were made. In addition to this there were still processors down on both the Linux and SUN clusters so no 16 processors runs could be done on the Linux cluster and no 15 processor run could be done on the original code on the SUN cluster.

Figure 2.1 - This graph shows the speed-up of the quarter-circle harbor grid results. The theoretical line on the graph is the same as the one described in the east coast grid results section. Linux cluster times again provided a better speed-up than SUN cluster times, the Linux cluster original code was superlinear through all processors and the predictor-corrector code was superlinear until about ten processors at which point it became just linear. SUN cluster results started off as near linear, but as the number of processors increased they quickly became sub-linear. One other thing to note is that the speed-up lines on this graph are a lot smoother than those on the east coast grid.

Q-Series Results

Figure 2.1 - Wall-Clock Time vs. Number of Processors
Quarter-Circle Harbor Grid, 100K Nodes, Maximum Stable Time Step

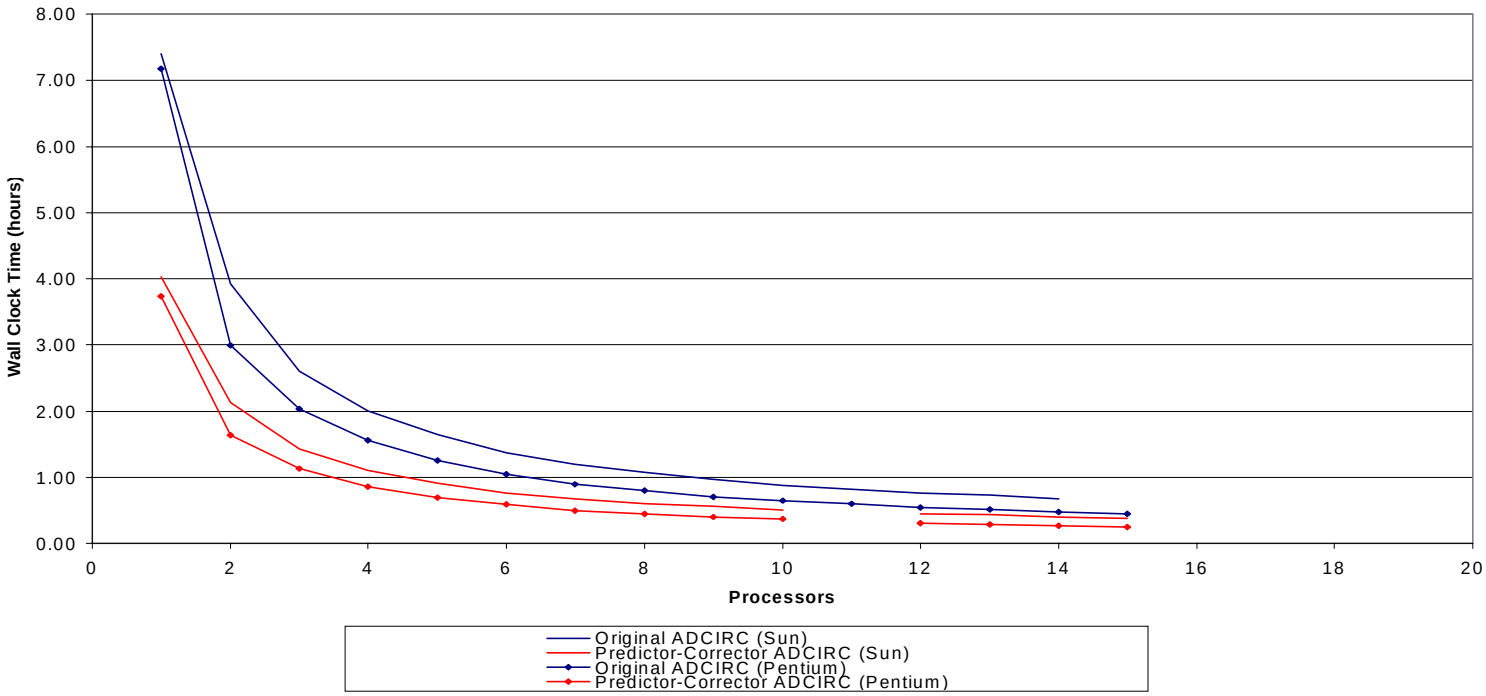


Figure 2.2 - Real-Time Speed-Up Compared to One Processor
Quarter-Circle Harbor Grid, 100K Nodes, Maximum Stable Time Step

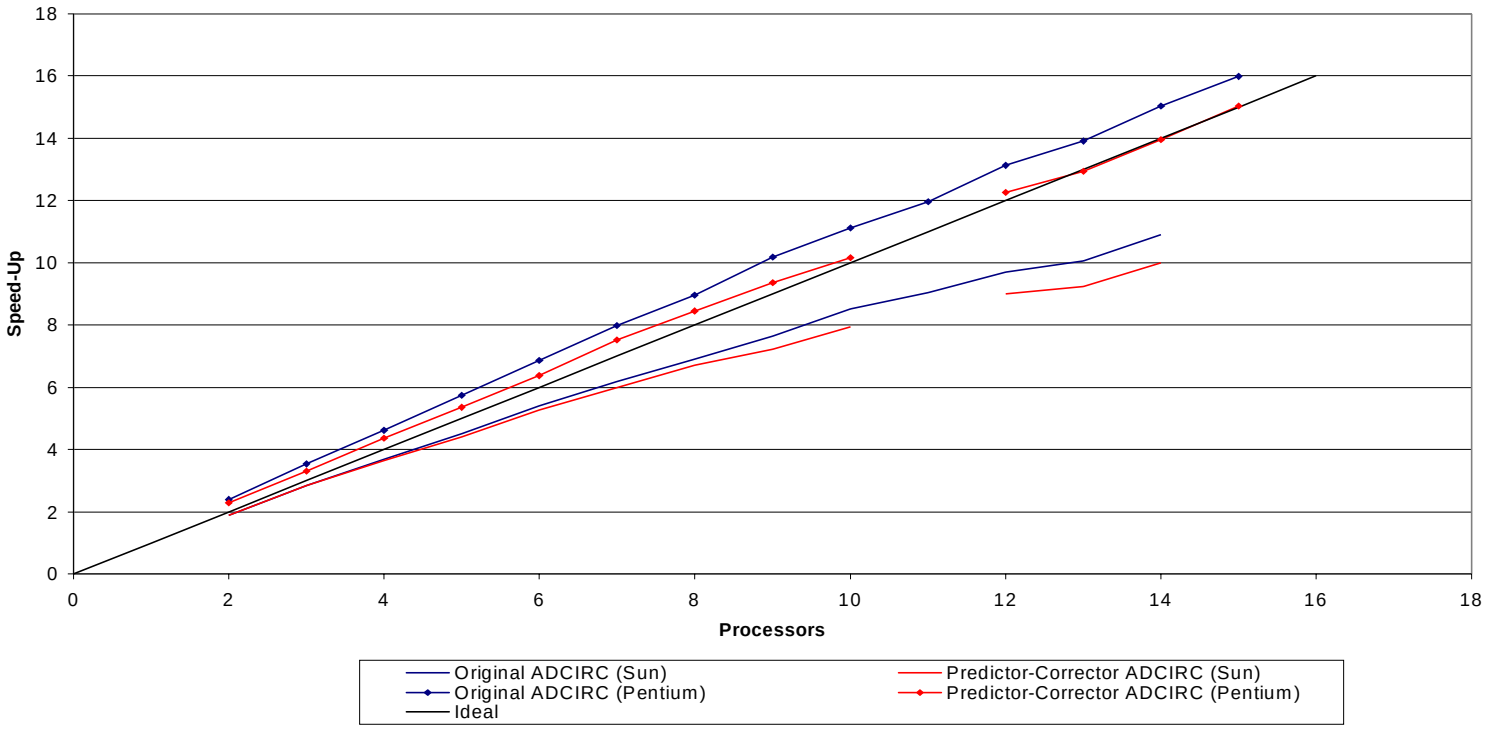


Figure 3.1 - Wall-Clock Time vs. Number of Processors
Quarter-Circle Harbor Grid, Constant 5000 Nodes per Processor, Time Step of 25 Seconds

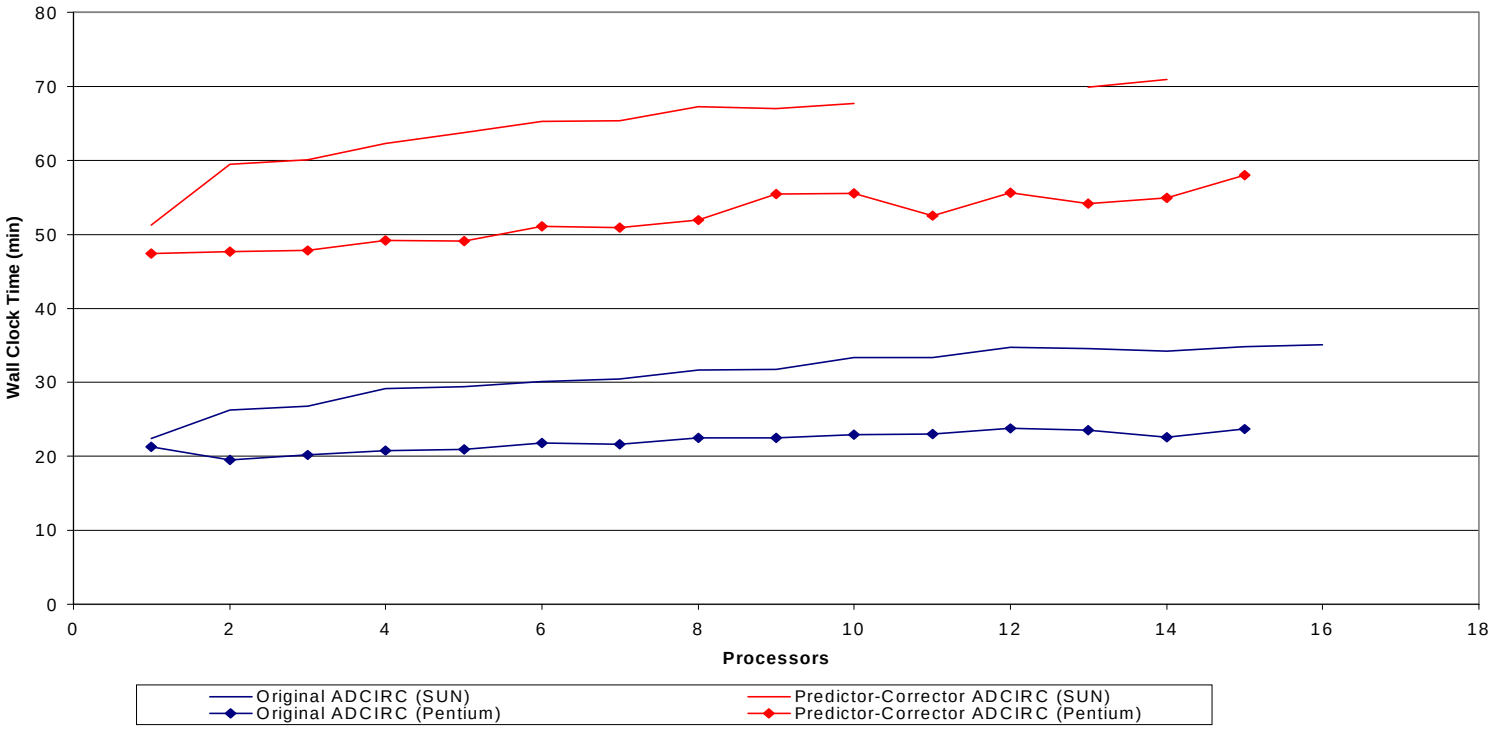


Figure 3.1 - As the number of processors increase there is a general trend for the time to increase as well. However there are some deviations from this, the Linux predictor-corrector code runs at two, seven, thirteen, and fourteen, Linux original runs at eleven, thirteen, as well a predictor-corrector SUN cluster run at fourteen all have a decrease in the amount of time, even though a processor is being added. Despite these small deviations the upward trend is still discernible, furthermore, the upward trend on the SUN cluster is greater the one on the Linux cluster, thus the SUN cluster lines have a greater slope. Difficulties were once again encountered on some of these runs, the eleven, twelve, and thirteen processors runs on the predictor-corrector code on the SUN cluster continuously crashed. Unfortunately here again processors were also down on the Linux and SUN clusters so the only sixteen processor run that was completed was on the original code on the SUN cluster.

Single Processor Harbor Grid

Figure 4.1 - This graph shows us the effects of all the workload being put on one processor, and then increasing that workload. There is a noticeable downward trend for both the Linux and SUN clusters, as the workload increases the work per unit time decreases. The important aspects of this graph are that when the SUN cluster reached a grid of 100,000 vertices there was a distinct drop off in the work per unit time. In addition to this there is a peculiar loop up in the Linux cluster results, then at 1,000 vertices the results align with the SUN cluster results.

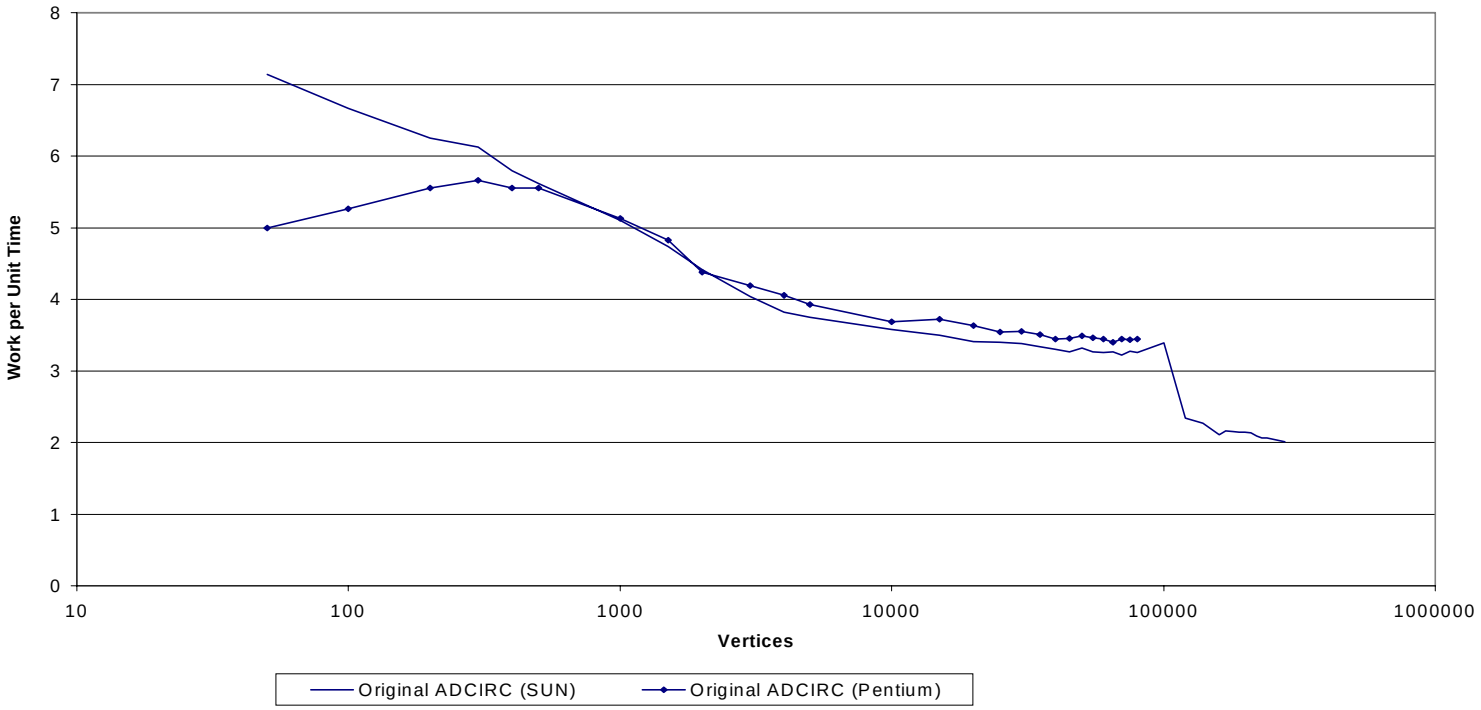
NetCDF results

Table 2.

type of code	run time with NetCDF (min)	run time without NetCDF (min)	output file size with NetCDF (bytes)	output file size without NetCDF (bytes)
1-D	0:25	1:08	11,376	35,944
2-D	31:19	31:18	4,126,400	12,426,162

When NetCDF was used in the one dimensional code it had a drastic affect, cutting down the run time by 1/2 and the file size by 2/3. However, when NetCDF was applied towards the much more

Figure 4.1 Work per Unit Time vs. Number of Vertices
Quarter-Circle Harbor Grid, Single Processor Runs



complex two dimensional code it did not have a significant effect on the time but it did reduce file size by 2/3.

Discussion

East Coast Grid

Several important questions arise from the results of the east coast grid. One of the most obvious is: why does the SUN cluster run this grid faster than the Linux cluster on a single processor, when the Linux cluster has faster processors. Well processor speed is not the only factor in how long it takes for a run to complete, cache size, communication speeds, and many other factors play a role in how long it takes a cluster to compute information. In the one processor case the SUN cluster had some of these other things that were better than what the Linux cluster had, so the SUN cluster had faster times. Furthermore, the Linux cluster was then able to have better times on the multiple processor runs because these things that the SUN was doing better on the single processor runs were more than compensated for by the fact that the Linux cluster has better communication between its processors than the SUN cluster has.

The one processor runs were the hardest part of these results to explain. The predictor-corrector being faster than the original code is explained by the fact that even though the predictor-corrector causes nearly twice as many computations to occur, it is being run a much higher time step, and this more than compensates for these extra computations. As previously explained the Linux cluster has faster times than the SUN cluster on multiprocessor runs because it does have faster processors and better communication between processors, these and other factors combine to compensate for the things that the SUN cluster does better.

As far as the scalability graph is concerned, why does the Linux cluster get super-linear speed-up? There are probably several factors that contribute to this. First of all for some reason the one processor runs on the Linux cluster just take a longer amount of time in comparison to the multi-processor runs, and since everything in this graph is compared to the one processor runs, it makes the multi-processor runs look good. One possible reason for this could be that the Linux cluster is getting good cache re-use. With only one processor there is only one cache, but with two processors there are two caches, so the more data that is being stored in cache the faster the run time is, and if the cache re-use is over compensating for the additional communication time between processors then a

super-linear speed-up scenario can occur. In all likelyhood this is one of several things contributing to this super-linear speed-up.

This explanation of what is occurring can be applied to what is happening to the SUN cluster's speed-up, however, rather than the cache overcompensating for the communication costs, it is the other way around. Communication costs are overwhelming any benefit that is obtained from this cache use, thus creating a scenario where sub-linear speed-up occurs. Two main factors could be the cause for this difference in communication. The first is that the Linux cluster has better communication software, and this enables the processors to talk to one another faster. The second is that the SUN cluster is a public cluster, meaning that the processors are connected to each other through the building's network, so theoretically data might have to travel all around the building before it got back to the other processor it was looking for. The Linux cluster on the other hand is a private cluster, each processor has a more direct communication with each other. Both of these factors contribute significantly to why one cluster has super-linear speed-up and the other has sub-linear speed-up.

Quarter-Circle Harbor Grid

Results from the east coast grid and the quarter-circle harbor grid are very similar, however, there are a few main differences. In the quarter-circle harbor grid results the Linux cluster one processor run is faster than the SUN cluster one processor run. How could this be? Well the important thing to realize is that these two grids are very different, the quarter circle harbor grid is a much more homogenous grid with nice smooth boundaries. The east coast grid on the other hand has much more complex coastal boundaries that match the real shape of the eastern coast line of the United States boundaries, as well as including some islands in the Caribbean. While it may seem that this difference should not have an effect, it does, for one thing it provides more noise in the times from the east coast grid, and it might cause some of the components that the SUN cluster had that made it perform better on a heterogeneous grid may not matter as much on a homogenous grid. Not to discount the previous argument that was provided about why the SUN cluster is faster than the Linux cluster on the east coast grid, that argument still stands up, because we can see that the difference between the one processor and two processor times is much greater on the Linux cluster than the SUN cluster, so there is obviously a battle going on between cache and communication. The issue here however, is that the homogeneousness of the grid also comes into play.

Outside of the one processor results everything else is relatively easy to explain, if we follow the logic previously given in the east coast grid discussion, then it is expected that on multiprocessor runs the Linux cluster would always be faster than the SUN cluster and the predictor-corrector version of the code would be faster than the original version of the code.

When comparing this speed up graph to that of the east coast grid it is easy to see that we have a more homogenous and uniform grid. The speed-up lines are nice and smooth, however, they do follow the general trend of the east coast grid speed-up grid, and much of that explanation can be used here. Again the single processor time was much greater than multi-processor times for the Linux cluster, thus helping to reinforce the fact that the cache is overcompensating for the communication between processors, thus having a superlinear speed-up. The reverse of this can be used for the SUN speed-up lines, the communication is winning the battle over the cache in this case and cause in a sub-linear speed-up.

Q-Series

In the quarter series results it becomes very apparent how much better the communication is between processors in the Linux cluster is versus that of the SUN cluster. Both do have a general trend of increasing the number of processors when you increase the amount of time, however, for both the predictor corrector code and the original code the slope is less for the Linux cluster than the SUN cluster. If we examine the jumps between single processor and two processor times we can further see how significant this communication cost it. The difference on the SUN cluster is about eight minutes on the predictor corrector code, and five minutes on the original code. If we look at the Linux cluster, its time actually goes down by about two minutes on the original code and it goes slightly up by less than a minute on the predictor corrector code. These communication costs are what makes the biggest difference between these clusters. Keep in mind too

that for these results that cache should not a have a significant effect since on each run each processor should be getting the same number of vertices as it did in a run with fewer processors, since 5,000 vertices are being added on each run.

Unlike the previous results both the predictor-corrector code and the original code were being run at the same time step. Since the predictor-corrector code has twice as many computations as the original code the fact that the predictor-corrector code times are twice as long as the original code times is easily understood.

Single Processor

The primary goal of these results was to get an idea of how big a job had to be before it overwhelmed a processor. For the SUN cluster this appears to happen at right about 100,000 vertices per processor. Unfortunately, Linux cluster runs did not extend this far, so there is some uncertainty as to what the work load per processor needs to be for the Linux processors to be overwhelmed. More work needs to be done on the Linux cluster to figure this out. The most peculiar thing about this graph is that the SUN cluster has a pretty steady downward trend from 100 vertices on, while the Linux cluster has a hump and then has a steady downward trend starting at about 1,000 vertices. The only explanation for this is some sort of noise. Runs that are done at less than 1,000 vertices take such a small amount of time, small time differentials can cause things such as this to occur, thus, this is not a big trend that would cause any worry, because the most significant thing that this graph says is that at 100,000 vertices the SUN cluster processor is overwhelmed.

NetCDF

The only significant conclusion that can be drawn from these results is that no matter how complex the version of ADCIRC might be, NetCDF can significantly cut down on the file output size. Even though NetCDF did cut down the amount of time that one dimensional ADCIRC took to run significantly, this run was not all that long, it was only a difference of about thirty seconds, and the difference for the two dimensional run was almost nothing. Run-time is much more heavily dependent on precisely how NetCDF is encoded into ADCIRC, rather than if it is just used. If NetCDF is coded inefficiently it could take much longer than ADCIRC takes without using NetCDF. There was probably not a significant change in time on the two dimensional code because it was so complex, so it was more difficult to code efficiently, and it is possible that the time it takes to write to a file is not as significant as the other processes it is undertaking. Whereas with the one dimensional code the dominant process was writing to a file, so NetCDF was able to do this much faster, the one dimensional code was also far simpler so it was easier to implement NetCDF in an efficient way.

Conclusions

Several issues dealing with the performance and the improvement of ADCIRC have been discussed dealing with both NetCDF and benchmarking. Benchmarking has revealed that there are in fact many things that contribute to run time. With the results from the quarter-circle harbor grid and east coast grid it was apparent that there is a battle between cache and communication costs, and that these had significant effects on what happened. Yet overall it is clear that the Linux was able to get the better of this battle, however, it did for the most part have better and more expensive components. Through benchmarking the effects of homogeneous vs. non-homogenous grids could also be seen. Q-series results added further proof to the ideas that the Linux cluster is getting much better communication than the SUN cluster. The single processor runs also showed that 100,000 vertices/processor is an extremely important an valuable cut off point for the SUN cluster. Finally, NetCDF turned out to be a viable solution for cutting down on file sizes but not for cutting down on run time.

Appendix A: NetCDF

The following are some guidelines and instructions dealing with NetCDF in order to enable other people who are working on ADCIRC to become familiar with it.

Methodology:

1) Follow the installation instructions that are online at the unidata website.

2) Link NetCDF to the directory where it is going to be used.

a)For the 1-D code you only have to do this once. type: “ln -s/usr/local/src/netcdf-3.5.0/src/f90/netcdf.mod”

b)For the 2-D code you also have to only do this once, but the way you do it is different. Link to the directory that you run the “gmake adcirc” command. If you run the “gmake adcirc” command the link will go in the odir4 directory, to recompile adcirc after this initial compilation type “rm -R odir4,” and remove every file but the NetCDF file then proceed to run the “gmake adcirc” again.

3)For the 1-D code you will need to compile the adcirc code every time you change it by typing “f90 -c adcirc.cdf.f90” for example. Then in order to finish the compilation you will need to type something like “f90 adcirc.cdf adcirc.cdf.o -L/usr/local/src/netcdf-3.5.0/src/lib -lnetcdf” Then you should be able to run the executable just by typing “adcirc”

4)In order to incorporate the previous step into the 2-D code you will need to modify the compiler flags. NetCDF is incompatible with the “-dalign” flag so you must remove that. Then you must modify where the flags link to things so that it will link to the netcdf libraries. For example you might add to FFLAGS5 a line that says “-L/usr/local/src/netcdf-3.5.0/src/lib.” Then right under the line that says “LIBS” and links to metis, you must add a line of code that says “LIBS2 := -L/usr/local/src/netcdf-3.5.0/src/lib -lnetcdf.” After completing this step you should be ready to compile NetCDF code with ADCIRC with no problems.

NetCDF Coding

Here are some instructions on NetCDF coding, however, for the most in depth instructions it is best to refer to the NetCDF handbook.

In order to use the NetCDF module you must put the “USE netcdf” statement before the “implicit none” in each file you are coding in. In addition to this the other main thing to remember when dealing with NetCDF is that whenever you open, or create a NetCDF file you must at some point close it when you are done using it.

```
status = NF90_CREATE(“netcdf.nc”, NF90_CLOBBER, ncid)
```

This line of code creates a netcdf data set. The data sets name is netcdf.nc, the NF90_CLOBBER means that if there is already a netcdf.nc that exists it will be overwritten, and finally, the ncid is an integer value that is the ID number of the data set. NF90_CREATE will return an integer which will be assigned to “status.” The purpose of this is to pick up any errors that might occur. If there has been an error an invalid value will be assigned to “status”. In order to see what the error is you can add a line like this

```
if (status /=nf90_noerr) call handle_err(status)
```

Of course you must then create a handle_err subroutine and there is a good example in the netcdf handbook. The next step in creating a netcdf data set is to set the dimensions of your variables, for example.

```
status = NF90_DEF_DIM(ncid, “GElevNetY” , NDSETSE, GElevNetY)
```

Here the “ncid” means that this dimension belongs to the data set with the ID “ncid,” the “GElevNetY” is the name of the dimension, the “NDSETSE” is a variable from ADCIRC and it denotes the size of the dimension, and finally the GElevNetY is the dimension ID. After assigning all the dimensions that you need, you must then create the variables that you will need, for this you will use a line such as this.

```
status = NF90_DEF_VAR(ncid, “VelocityStations” , NF90_DOUBLE, (/VelNetW/), velSID)
```

Again the “ncid” means that this variable is part of the netcdf data set with the ID ncid, the name of this variable is “Velocity Stations”, NF90_DOUBLE is the data type, VelNetW is the ID of the dimension that the variable has, and velSID is the ID of the variable. After this you can exit the define mode, make sure that when you do this all the variables you will need have been defined. The next step is storing values in the variables, this will probably look something like this.

```
status = NF90_PUT_VAR(ncid, runinfoID, VALUE, (/2/))
```

Again the “ncid” means this variable is stored in the netcdf file with the ID ncid and the variable with the ID runinfoID, the value that is actually being stored is VALUE. Whenever you are done with the netcdf file you must close it by doing.

```
status = NF90_CLOSE(ncid)
```

This line will close the netcdf file with the ID of ncid. To best optimize your code you should minimize the amount of times you open and close the data set. If you have already created the data set and need to access it after you've closed it you can use the NF90_OPEN to interact with it again.

Appendix B: Compiler Flags

In addition to the benchmarking studies, I also took a look at the compiler flags that ADCIRC was ran with. The Compiler flag that I did all of the benchmarking with was the -O flag, which is also the same as the -O2 flag. On a basic run with ADCIRC I compared the -O1, -O2, -O3, -O4, -Ounsafe, -Oassumed.

Table 3.

Flags	Real Time	User Time	System Time
-O1	1:11	1:05	0:01
-O2	1:06	1:00	0:01
-O3	1:08	1:07	0:01
-O4	1:09	1:02	0:01
-O2 -Ounsafe	0:58	0:51	0:01
-O2 -Ounsafe -Oassumed	1:00	0:52	0:01
no flags	2:26	1:08	0:01

Acknowledgements

I would like to take this opportunity to thank the National Science Foundation for providing the funding for this research project. The opinions, views, and findings do not necessarily reflect those of the National Science Foundation.

References

Dresback KM, Kolar RL. An Implicit time-marching algorithm for 2-D GWC shallow water models. In *CMWR XII: Computational Methods in Water Resources*, vol. 2, Bentley LR, Sykes JF, Brebbia CA, Gray WG, Pinder GF(eds). A.A. Balkema: Rotterdam, 2000: 913-920.

Dresback KM, Kolar RL. An implicit time-marching algorithm for shallow water models based on the generalized wave continuity equation. *International Journal for Numerical Methods in Fluids* 2001; 36: 925-945.

Kinnmark IPE. The shallow water wave equations: formulations, analysis and application. In *Lecture Notes in Engineering*, vol 15, Brebbia CA, Orsszag SA (eds). Springer-Verlag, Berlin, 1986; 187.

Kolar RL, Gray WG, Westerink JJ, Luettich RA. Challow water modeling in spherical coordinates: equation formulation, numerical implementation and application. *Journal of Hydraulic Research* 1994; 32(1): 3-24.

Kolar RL, Looper JP, Westerink JJ, Gray WG. An improved time marching algortihm for GWC shallow water models. In *CMWR XII: Computational Methods in Surface Flow and Transport Problems*, vol. 2, Burganos VNI, Karatzas GP, Payatakes AC, Brebbia CA, Gray WG, Pinder GF (eds). Computational Mechanics Publications: Southampton, 1998: 379-385.

Luettich RA, Westerink JJ, Scheffner NW. ADCIRC: an advanced three-dimensional circulation model for shelves, coasts, and estuaries. Report 1: theory and methodology of ADCIRC-2DDI and ADCIRC-3DL. Technical Report DRP-92-6, Department of the Army, USACE, Washington, DC, 1992.

Lynch DR, Gray WG. A wave equation model for finite element tidal computations. *Computer and Fluids* 1979; 7(3): 207-228.

Westerink JJ, Luettich RA, Blain CA, Scheffner NW. ADCIRC: an advanced three-dimensional circulation model for shelves, coasts and estuaries. Report 2: Users MAnual for ADCIRC-2DDI, Department of the Army, USACE, Washington, DC, 1994.