A Comparison of Using APPL and PVM for a Parallel Implementation of an Unstructured Grid Generation Program[*]

T. Arthur[†] M. Bockelie[‡]

December 28, 1992

Abstract

In this report we describe our efforts to parallelize the VGRIDSG unstructured surface grid generation program. The inherent parallel nature of the grid generation algorithm used in VGRIDSG has been exploited on a cluster of Silicon Graphics IRIS 4D workstations using the message passing libraries Application Portable Parallel Library (APPL) and Parallel Virtual Machine (PVM). Comparisons of speed up are presented for generating the surface grid of a unit cube and a Mach 3.0 High Speed Civil Transport. It was concluded that for this application, both APPL and PVM give approximately the same performance, however, APPL is easier to use.

1. Introduction

For many engineering applications, using distributed computing and a cluster of high performance work stations is a cost effective alternative to using a supercomputer. Historically, engineering applications were coded to run in serial mode on supercomputers because only a supercomputer could provide the large amounts of memory and fast computational speed required to perform such simulations. Due to the heavy work load of most supercomputers today, the elapsed time to perform a simple analysis can be far greater than the CPU time to perform the required computations. In distributing computing, programs are designed to execute the time consuming portions of the computations in parallel across one, or more, computers. In recent years, high performance workstations have become available that can provide the memory, inter-machine communication and computational speed required to

[*]This work was performed under NASA contract NAS1-19038
[†]Associate Member of the Technical Staff, Computer Sciences Corporation, Hampton, VA
[‡]Senior Member of the Technical Staff, Computer Sciences Corporation, Hampton, VA

perform such computations. Furthermore, high performance workstations can be obtained at a fraction of the cost of a supercomputer. Our principal objective in the present work is to investigate the use of distributed computing to perform surface grid generation for a realistic configuration.

In this report we describe our work on parallelizing the VGRIDSG unstructured surface grid generation program. Two software packages, APPL and PVM, are used to distribute the program over a cluster of Silicon Graphics IRIS (SGI) 4D workstations. Each package is described and performance results are given for generating the surface grid for a unit cube and a Mach 3.0 High Speed Civil Transport (HSCT). In addition, recommendations for using each message passing library are reported.

This report is organized as follows: Sections 2 and 3 contain an overview of APPL and PVM respectively. Section 4 is a description of the distributed architecture used. Section 5 contains a description of the VGRIDSG unstructured surface grid generation program. Section 6 briefly describes the modifications to the serial code in order to distribute the program. section 8 contains grid generation timing results and Section 9 compares the two software packages. Section 10 contains concluding remarks.

2. Description of APPL

Application Portable Parallel Library (APPL) is a message passing library developed by the Internal Fluid Mechanics Division at the NASA Lewis Research Center. The purpose of the software is to provide a basic set of portable subroutine calls for message passing over a variety of MIMD (Multiple Instruction Multiple Data) architectures. The software is free (subject to NASA's conditions for software distribution) and can be obtained by sending e-mail to Angela Quealy at fsang@kira.lerc.nasa.gov. The results presented in this report are for version 2.2, last updated March 31, 1992. The documentation provided with APPL consists of a nine page ASCII README file and several detailed man pages.

By using APPL, the user can access the local processors of a machine and/or homogeneous processors over a network of connected workstations. These independent parallel processes communicate only through message passing. For a multiple processor shared memory machine, such as a SGI 440, APPL will use semaphores to block access to memory locations. The use of semaphores creates the illusion of a distributed memory architecture. This is an important portability issue since many massively parallel machines have distributed memory. For example, the Intel iPSC/i860 Hypercube consists of processors that have their own private memory and thus must exchange data by message passing.

Messages can be sent either synchronous or asynchronous. When using synchronous messages, the process will wait until the message has completed before executing the next instruction. An asynchronous message will immediately execute the next instruction without waiting for the message to complete. The syntax for a synchronous message in Fortran is *call ssend(msgtype, msg, length, proc_id)* where *msgtype* is an integer to specify the message type, *msg* is the message being sent, *length* is the length of the message in bytes and *proc_id* is the process number to which the message is being sent. To receive a synchronous message, the routine *call srecv(msgtype, msg, length)* is used. The syntax for an asynchronous message is *call asend(msgtype, msg, length, proc_id)*. Similarly, the routine *call arecv(msgtype, msg, length)* should be used to receive asynchronous messages. Synchronous messages must be received synchronously and asynchronous messages must be received asynchronously. APPL also has routines for getting message information, global operations and timing.

To create a distributed computing environment for APPL, the user must create a processor description file. This file, *procdef* by default, contains the user's login id *(username)*, the target machine name *(hostname)*, the working directory *(work_dir)*, the number of processes *(num_procs)* and the executable name of the processes *(executables)*. The following is an example *procdef* file for SGI workstations:

```
# this is a comment line
# procdef for shared memory architecture
username hostname work_dir num_procs executables
```

On SGI workstations, invoking the *compute* command runs the *executables* in the *procdef* file. Before ending execution, the last call in the user's APPL program should be *pend()* which cleans up the semaphores and environment. If the program terminates abnormally before calling *pend()*, the user MUST release the semaphores and shared memory manually. The resources can be released by executing the following commands (do NOT execute these commands as root):

```
ipcs -m    | fgrep $LOGNAME | awk '{print "ipcrm -m", $2 }'  | sh
ipcs -s    | fgrep $LOGNAME | awk '{print "ipcrm -s", $2 }'  | sh
```

These commands query the allocated shared memory (*ipcs -m*) and semaphores (*ipcs -s*). The pipe, *fgrep $LOGNAME*, prints only the resources allocated by the user. The pipe, *awk '{print "ipcrm -m", $2 }'*, gets the process id for the resources allocated and uses the command *ipcrm* to deallocate the shared memory (*ipcrm -m*) or the semaphores (*ipcrm -s*). Note that the user will need to execute these commands on each machine on which the processes were running.

3. Description of PVM

Parallel Virtual Machine (PVM) is a heterogeneous message passing library developed at the Oak Ridge National Laboratory (ORNL). PVM allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. It is a library (two libraries if you use Fortran) and a daemon process. The idea is to couple together multiple resources in a parallel fashion to use the best properties of a particular machine for an application with moderately large granularity. The results presented in this report are for PVM version 2.4.1, last updated May 31, 1992. The documentation for PVM consists of a Postscript file that must be obtained separately from the software. PVM is free software and can be obtained by sending the e-mail message *send index from pvm* to the automated server netlib@ornl.gov.

The PVM processes communicate through the daemons created on each machine. Over a heterogeneous network, PVM uses the external data representation (XDR) to send messages over different architectures. PVM has the capability to recognize a homogeneous network and will not perform the XDR conversion in these situations (ref. 1).

For PVM, messages are sent asynchronous and received synchronous. A message is sent in three stages: initializing the send buffer, putting the message in the buffer and sending the message. The syntax for sending a floating point Fortran message is:

```
call finitsend()
call fputnfloat(msg, length, info)
call fsnd("executable\0", proc_id, msgtype, info)
```

where *msg* is a floating point message, *length* is the message length, *info* is an error code, *executable* is the name of the executable of the PVM process, *proc_id* is the processor the message is being sent and *msgtype* is an integer to specify the message type. To receive this message, the following routines are used:

```
call frcv(msgtype, info)
call fgetnfloat(msg, length, info)
```

The call to *frcv* will not continue until the message arrives in the buffer. PVM also allows messages to be sent in bytes. However, sending bytes will only work if the machines have the same byte ordering scheme.

PVM contains routines for obtaining information on messages and some synchronization routines. Unlike APPL, PVM does not have a timing function or global operation routines.

To make a distributed computing environment for PVM, the user must create a file similar to the APPL *procdef* file. This file contains the machines in the PVM, the user's login id for that machine *(username)* and the location of the PVM daemon (specified by $dx =$) (ref. 2). PVM searches the directory $HOME/pvm/HOST where HOST is the PVM name of the machine (see ref. 1 for names of machines) for the executable process. The following is an example of a file describing the PVM:

```
# this is a comment
# an asterisk in the first column sets the default
* dx= /tmp/pvm/pvmd
* lo= username
host1
host2
```

To execute the PVM in batch mode, type the command *pvmd host_file* &. To run *pvmd* interactively, type *pvmd -i host_file*.

If the PVM daemons are successful in starting, the message *pvm is ready* appears. If the PVM returns a message *pvmd garbled response* or *expected pvmd but got ''*, this usually means that PVM can not read the password file. To remedy this situation, try using the *pw* option in the host file of checking your *.rhosts* file.

Once the daemons are ready, it is convenient to start the PVM processes using a host program, otherwise each process must be started individually on each machine. After the PVM processes have completed, the PVM daemons can be killed. When in interactive mode, the user can query the daemons on running processes. If one of the daemons in the PVM network abnormally terminates, then all of the PVM processes will be killed by PVM. Unlike APPL, the user does not need to clean up resources.

I/O in PVM is redirected to the user home directory. Therefore, the user should specify full path names of files. PVM does not have the capability to specify a working directory, as does APPL.

4. Hardware

The VGRIDSG program has been distributed over five SGI workstations located at the NASA Langley Numerical Geometry Laboratory (GEOLAB). The distributed computer consisted of the following workstations: an IRIS 4D/440, two IRIS 4D/420 and two IRIS 4D/320. These workstations are Ethernet connected on a inner hub of the Langley Computer Network (LaRCNET). The table below shows the expected performance of the workstations. In the table, DP Linpack is a double precision Linpack benchmark which utilized multiple CPU's on machines with more than one CPU (ref. 3).

CPU's	Clock	DP Linpack	Memory
4D/440	40 MHz	42 MFLOPS	96 MB
4D/420	40 MHz	23 MFLOPS	64 MB
4D/320	33 MHz	20 MFLOPS	48 MB

For this study, the FORTRAN compiler used was *f77* version 3.4.1 and the operating system used was IRIX release 4.0.1.

5. Overview of VGRIDSG

VGRIDSG is an unstructured surface grid generation program developed at GEOLAB (ref. 4). The code is based on the surface grid generation routines contained in VGRID3D (ref. 5) but has been modified to use a surface definition defined by networks of bi-cubic Hermite patches. Using the bi-cubic patches results in a much smoother surface than could be obtained with VGRID3D, but also required substantially more CPU time to compute the surface grid. To compute the surface grid for the HSCT used in this study (over 12,000 triangles) requires about 10 minutes of CPU time (serial mode) on the SGI 4D 440 workstation.

VGRIDSG generates the surface grid using an advancing front method which has a large degree of inherent parallelism. In the advancing front method, the grid points are first placed on the curves that constitute the edges of the surface patches. After discretizing all of the edge curves, the interior of the patches are filled with triangles. Note that the interior of a patch can be filled with triangles independently of the other patches. Thus, it is possible to compute the interior triangles of the surface patches in parallel. Because generating the surface triangles is the most time consuming portion of the computations in serial mode, we have focused our parallelization efforts only on this procedure. In the future, it may be possible to investigate parallelizing other portions of the code in order to achieve even greater improvements in the overall program speed up.

6. Modifications to VGRIDSG

To parallelize VGRIDSG required modifying the main body of the program and two subroutines, *frontuv* and *bsegad*. Subroutine *frontuv* is the main driver for computing the surface patches. Subroutine *bsegad* is called by *frontuv* after generating the surface triangles for a patch, and is used to store the coordinates and element connectivity of the new triangles into global arrays. The changes in the main program and *frontuv* are minor in comparison to the required changes to *bsegad* because *bsegad* contains all of the interprocessor communication that occurs in VGRIDSG. A file containing a detailed description of the changes to the source code can be obtained upon request from the authors.

The basic changes made to the main body of the program are: adding work arrays, initialization of the distributed computer, redirecting output through processor zero and passing processor information to the subroutines. The work arrays were needed for communication purposes. In addition, in our implementation all I/O is directed through processor zero.

The modifications to subroutine *frontuv* were rather minor. The changes include having new arguments passed in from the main body of the program, parallelizing the do-loop that controls the generation of the surface triangles in each surface patch, passing arguments to subroutine *bsegad* and getting timing information for each patch for load balancing.

The changes to subroutine *bsegad* were quite extensive. In our parallel implementation, the interior triangles for the surface patches are computed on separate processors. Hence, each processor calculates only a portion of the total data. In subroutine *bsegad*, the data from the separate processors is collected and stored into global arrays. Thus, all of the interprocessor communication takes place in *bsegad*. Because APPL and PVM pass messages differently, the APPL and PVM versions of this subroutine are quite different.

7. To Run The Parallel Code

The Makefile used to compile the program was modified to include the addition of the APPL and PVM libraries. When using PVM, it is convenient to create a host program to start the PVM parallel processes. To run the PVM version, compile both the host and node programs. Then start the PVM daemons using the command *pvmd* (see section 3). Once the daemons are ready, then run the host program. To run the APPL version of the code, simply type *compute*. The *compute* command will use the *procdef* file to create the parallel environment.

8. Grid Generation Timing Results

The performance of parallel programs are typically measured in terms of speed up and efficiency. Speed up is the ratio of the sequential time to the parallel time. In other words, the speed up of a program is (ref. 6):

$$S = \frac{\tau_1}{\tau_{\mathcal{P}}} \tag{8.1}$$

where τ_1 is the sequential CPU time and $\tau_{\mathcal{P}}$ is the elapsed time for the parallel version. Perfect speed up is when the speed up S is equal to the number of processors $\mathcal{P}$. The efficiency is a measure of how well the processors are being utilized. The efficiency of the parallel program is defined as (ref. 6):

$$\eta = \frac{S}{\mathcal{P}} \tag{8.2}$$

From Eq. 8.2, it can be seen that the speed up is directly proportional to the efficiency.

In this report, each of the performance figures contains three curves. The first is a curve for perfect speed up, used as a reference. The second curve is the speed up of the subroutine *frontuv*, the parallel portion of the code. The third curve is the speed up of the complete program. This last curve gives the user the expected improvement in run time for of the parallel program. In our implementation, we have focused on only parallelizing the tasks performed in subroutine *frontuv*. Thus, it is expected that the complete program will have a lower speed up than *frontuv* due to the additional serial computations that are performed elsewhere in the program.

The results presented in this report were taken during dedicated times, in the sense that no other users are logged onto the machine and no user processes were executing. In reality, this dedicated scenario does not exist for every day computing, so the reader should realize that performance can be significantly lower than what is reported in this report. In the following tests, we always use the faster 400 series workstations before making use of the slower 300 series workstations (see Section 4). In addition, the speed up calculations are based on the sequential CPU time for executing the code on a single processor on a 400 series workstation. Last, note that we only present results for using 1, 2, 3, 4, 6, and 12 processors because in our parallel implementation we require that the number of patches be an integer multiple of the number of processors used. Although this approach does not allow us to take advantage of all the available processors, it does simplify the code modifications required to alter which processors are used.

To verify the parallel versions of VGRIDSG, our first test case was to generate the surface grid for a simple unit cube (see Figure 1). For this problem, each face of the cube was split into two patches, resulting in a total of twelve surface patches. In addition, the grid point spacing was set to a uniform value so that each patch generates approximately the same number of triangles. Because each patch requires about the same number of calculations, the cube problem has a near perfect load balance.

Figures 2 and 3 show the speed up versus processors for the cube problem using APPL and PVM, respectively. Both packages show approximately the same performance. Comparing the curves for the complete program and subroutine *frontuv*, it can be seen that, as expected, *frontuv* has a greater speed up than the complete program. The performance curves for the complete program and *frontuv* approach maximum speed ups of 8.1 and 6.2, respectively. The curves are linear up to four processors, because all four processors reside on one machine. The curve is slightly less than linear for the six processor case because the two additional processors reside on a second machine and this adds some communication overhead. For the 12 processor test, the efficiency decreases due to using a combination of processors with different clock cycles from five machines.

Our second test case was to generate the surface grid for a Mach 3.0 High Speed Civil Transport (HSCT). Here, eighteen patches were used to define the vehicle surface, sting and far field computational boundaries (see Figures 4 and 5). Illustrated in Figures 6 and 7 are the speed up versus the number of processors using APPL and PVM, respectively, for the case of not load balancing the surface patches; that is, computing the surface patches in the order in which they were defined. Clearly, there is little gain in calculating the grid in this manner. To improve the performance, the grid was recomputed by re-ordering the sequence in which the surface patches are generated in order to obtain better load balancing.

For this test we used a simple, but effective, scheme to improve the load balancing. First, we computed a very coarse version of the surface grid that required only a small fraction of the time required to compute the desired fine mesh. Serial timings were made for each patch and a linear relationship for the the increased amount of time needed to compute the fine grid was assumed. The patches were then re-ordered in descending order based on the estimated CPU times (i.e., compute those patches requiring the most CPU time in parallel).

Figures 8 and 9 show the effect of load balancing the problem for using APPL and PVM, respectively. The speed up of subroutine *frontuv* is asymptotic to about 3.4 and the overall speed up is asymptotic to approximately 3.0.

The test results presented here indicate that performing unstructured grid generation using VGRIDSG may not a scalable problem. To a certain extent, this is due to the hardware configuration on which these tests were performed. In parallel processing, the measure of performance is affected by the slowest processor of the group. Because the 300 series workstations are slower than the 400 series workstations, the overall performance is adversely affected. Another aspect effecting the scalability of the problem is that in unstructured surface grid generation the work load amongst the surface patches can be very unevenly distributed. Based on the desired grid point spacing and type of surface definition used, computing the grid for some patches can require substantially more time than will other patches. For example, in the HSCT problem approximately 85% of the total execution time is spent computing only 4 of the 18 patches. Furthermore, there is no simple means to divide the work load of these four patches amongst multiple processors without substantially increasing the human effort to define the problem. Clearly, many more tests using different vehicles will be required before making a conclusion on this issue.

9. Comparisons of Using APPL versus PVM

In this report, we found that APPL has some note-worthy advantages over PVM. First, we found it much easier to implement message passing in APPL. It takes two lines of code to send and receive a message in APPL, but PVM takes five lines to do the same. Second, APPL has global functions while PVM has none. In addition, APPL allows I/O redirection but PVM does not. Last, we found it easier to execute (control) the parallel program using APPL, because to use PVM requires starting of daemons. These properties of PVM makes it difficult to execute batch jobs in UNIX, however, if the Distributed Queuing System (DQS) is installed on the distributed computer, the batch scripts are significantly easier to write (see ref. 7).

There are some drawbacks to using APPL. First, abnormal termination of the executing processes will not release allocated resources. This could cause large portions of the memory to be allocated to a single user rendering the workstation useless for other users until the resources have been released. In addition, APPL is limited to a homogeneous environment.

Debugging the parallel code is difficult in both packages. Though ORNL is developing a X Window debugger, neither package currently has a debugger.

It should be noted that both APPL and PVM are continuously being upgraded. Currently, the developers of APPL are considering a modification to allow the utilization of a heterogeneous network. The next release of PVM, known as PVM 3.0, is scheduled to be made available by December 1992. It appears that both packages are converging to the same solution and will offer the same functionality at some point in the future.

10. Conclusions

In this report we have presented the results of our work on using distributed computing to improve the performance of the VGRIDSG unstructured surface grid generation program. The program was distributed across a cluster of high performance workstations using the APPL and PVM message passing libraries. For the tests reported here, APPL had approximately the same performance as PVM. Based on overall ease of use, it was determined that APPL was the better of the two software packages for this application.

References

1. Beguelin, A.; Dongarra, J.; Geist, A.; Mancheck, R.; Sunderman, V.:*A Users' Guide to PVM.* Oak Ridge National Laboratory, Oak Ridge, TN, July 1991.
2. Grant, B.K.; Skjellum, A.: *The PVM Systems: An In- Depth Analysis and Documenting Study - Concise Edition.* Lawrence Livermore National Lab, Livermore, CA, 20 August 1992.
3. Gorey, K.: Periodic Table of the IRISes. 21 July 1992.
4. Bockelie, M.J.: *Summary of Modifications to VGRID3D.* unpublished report, Sept 8, 1992.
5. Parikh,P.; Pirzadeh, S.; Lohner, R.: *A Package for 3D Unstructured Grid Generation, Finite Element Flow Solution, and Flow Field Visualization.* NASA CR 182090, September 1992.
6. Wilson, G.V.: *Practical Parallel Programming (Glossary).* September 1992.
7. Revor,L.: *DQS Users Guide.* Argonne National Laboratory, September 15, 1992

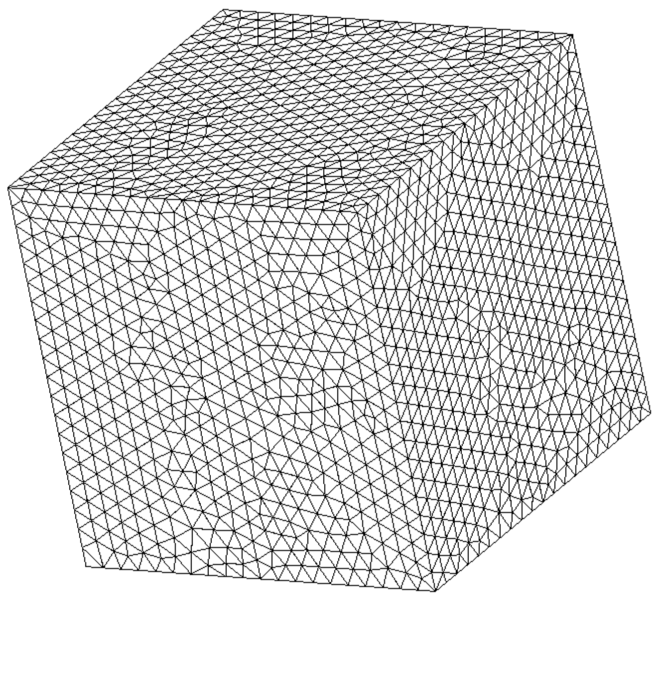

Figure 1. Coarse unstructured grid for a cube.

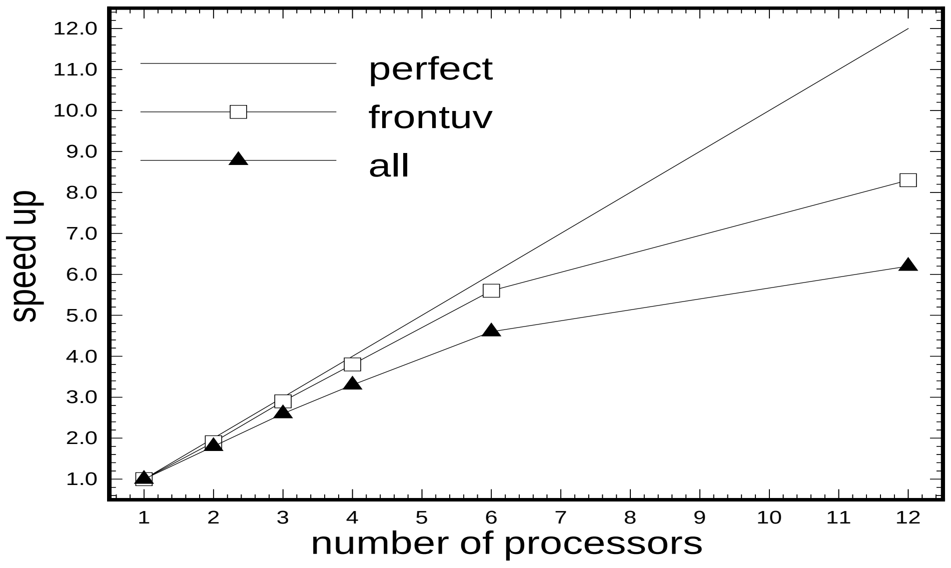

Figure 2. Speed up for cube using APPL.

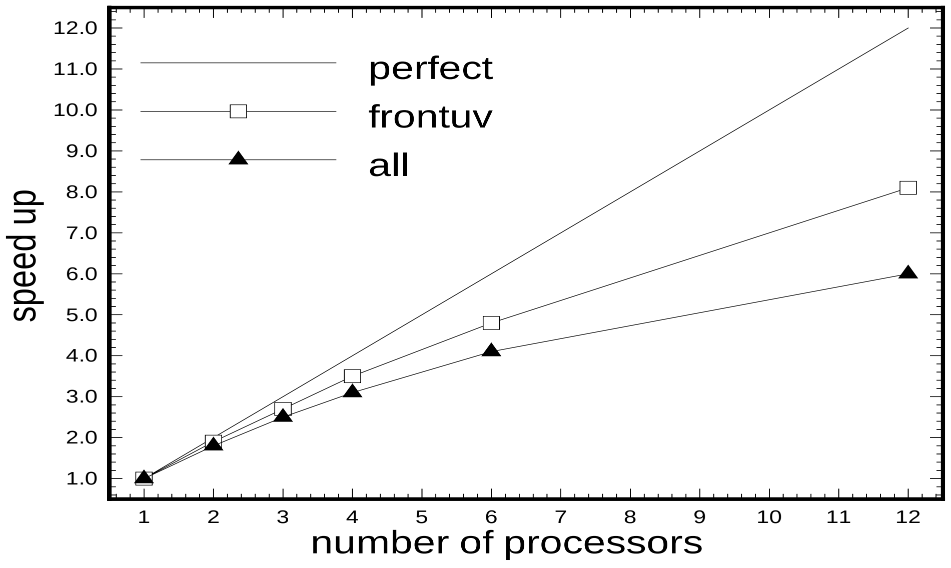

Figure 3. Speed up for cube using PVM.

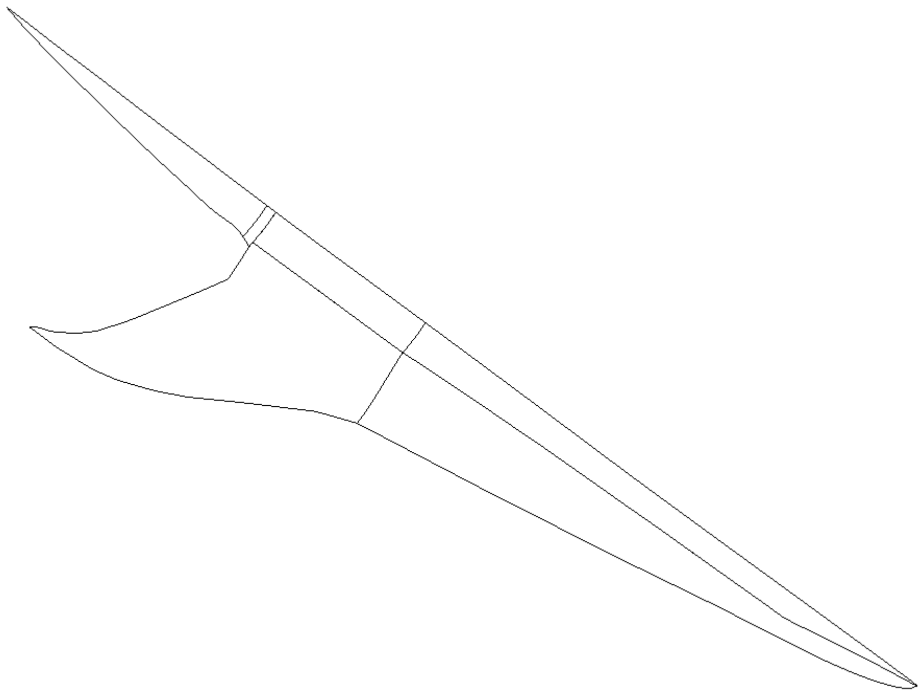

Figure 4. Surface patch boundaries for upper surface of HSCT.

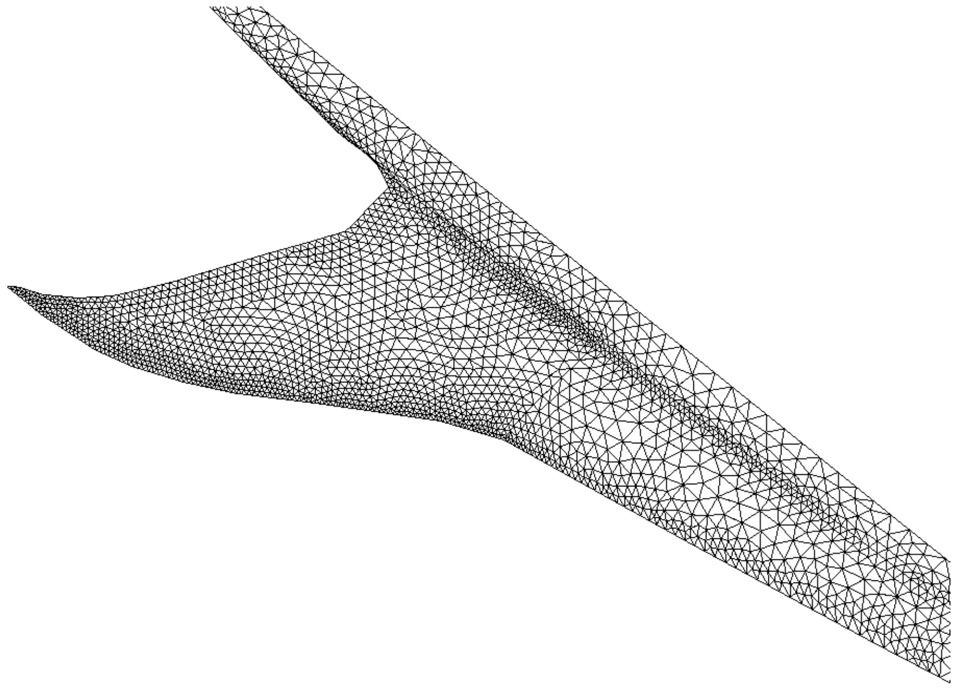

Figure 5. Portion of HSCT unstructured grid.

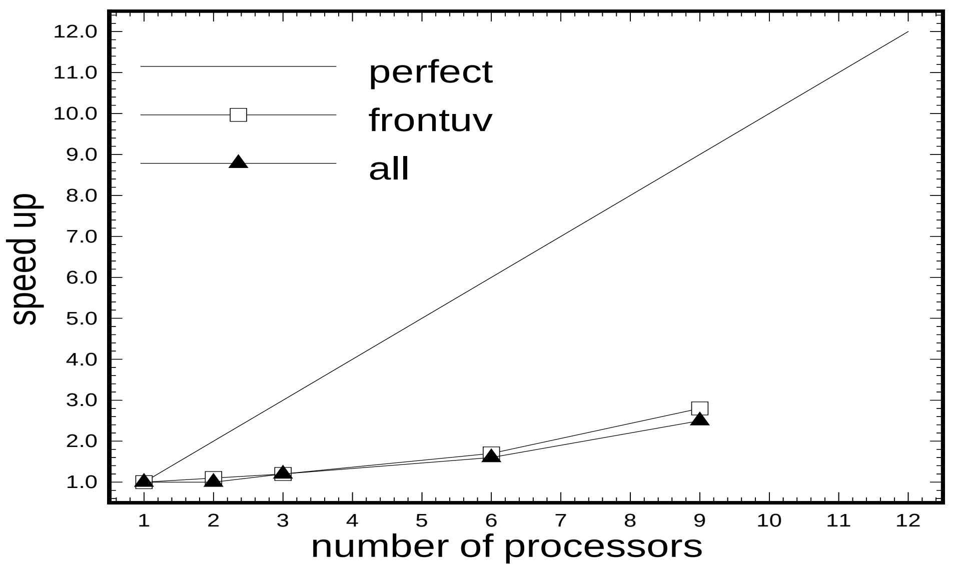

Figure 6. Unbalanced speed up for HSCT using APPL.

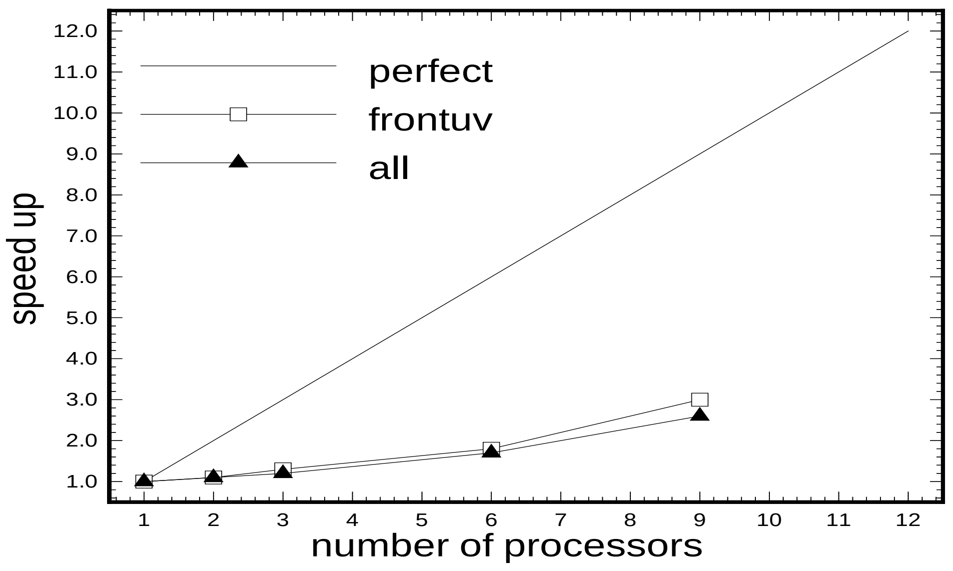

Figure 7. Unbalanced speed up for HSCT using PVM.

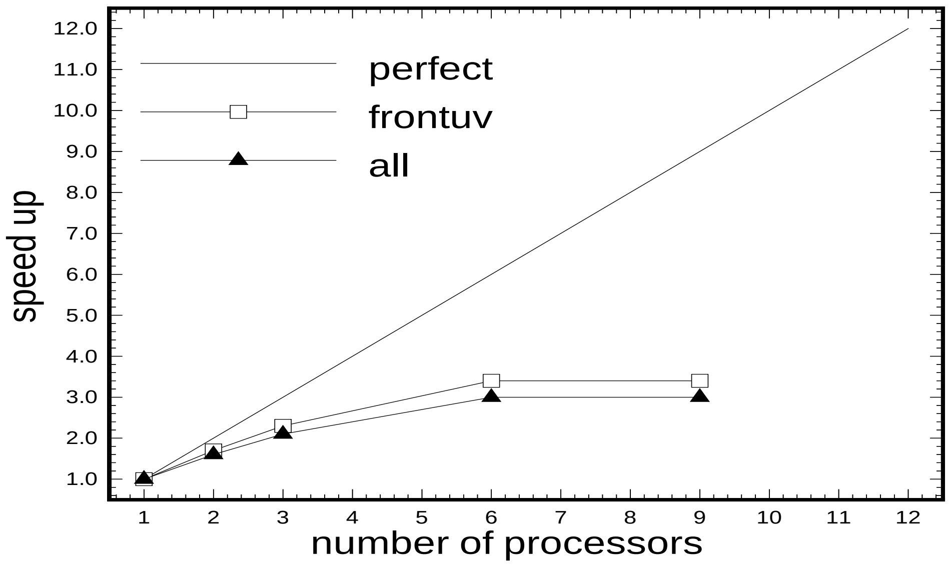

Figure 8. Load Balanced speed up for HSCT using APPL.

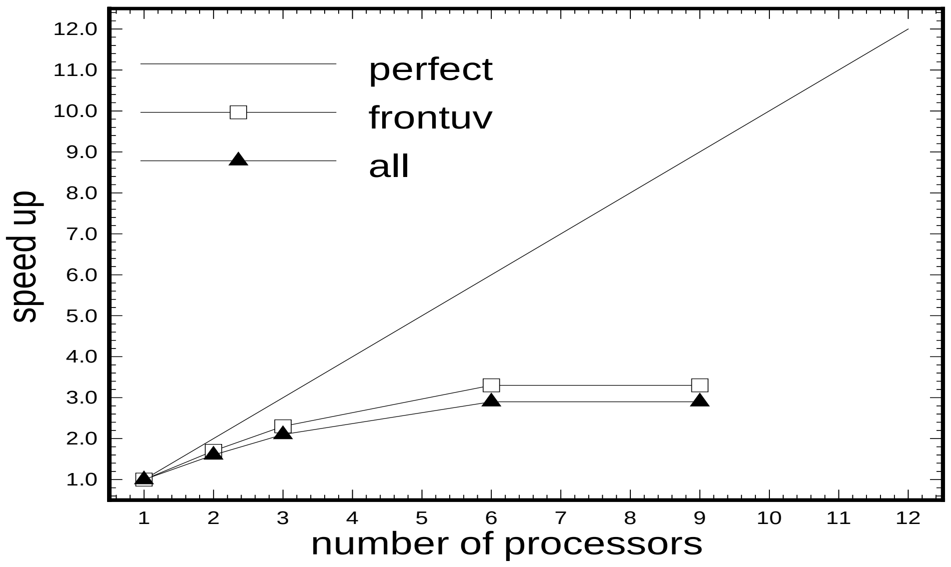

Figure 9. Load Balanced speed up for HSCT using PVM.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE January 1993	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE A Comparison of Using APPL and PVM for a Parallel Implementation of an Unstructured Grid Generation Program	5. FUNDING NUMBERS C NAS1-19038 WU 505-90-53-02
6. AUTHOR(S) Trey Arthur Michael J. Bockelie	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Computer Sciences Corporation 3217 N. Armistead Ave. Hampton, VA 23666-1379	8. PERFORMING ORGANIZATION REPORT NUMBER TAO 60322
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA CR-191425

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Samuel A. McPherson ACD-CMB

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified–Unlimited Subject Category 61	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
In this report we describe our efforts to parallelize the VGRIDSG unstructured surface grid generation program. The inherent parallel nature of the grid generation algorithm used in VGRIDSG has been exploited on a cluster of Silicon Graphics IRIS 4D workstations using the message passing libraries Application Portable Parallel Library (APPL) and Parallel Virtual Machine (PVM). Comparisons of speed up are presented for generating the surface grid of a unit cube and a Mach 3.0 High Speed Civil Transport. It was concluded that for this application, both APPL and PVM give approximately the same performance, however, APPL is easier to use.

14. SUBJECT TERMS Distributed Computing; APPL; PVM; Surface Grid Generation; Unstructured Grids	15. NUMBER OF PAGES 15
	16. PRICE CODE AO3

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT