# Parallel Implementation of ADVISE on the Intel Paragon

# Parallel Implementation of ADVISE on the Intel Paragon

M. Rezny[1]

Research Report No. 94-10
September 1994

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

[1]Department of Mathematics, University of Queensland, Brisbane 4072, Australia

# Parallel Implementation of ADVISE on the Intel Paragon

M. Rezny[1]

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

Research Report No. 94-10 September 1994

**Abstract**

Environmental models, such as the Pasture Model, require large computing resources to give results within reasonable time frames. This report focuses on the design and implementation of a parallel version of an ADVISE server running the Pasture Model on the Intel Paragon.

In order to make the most efficient use of this computing resource, it is first necessary to perform a thorough analysis of the message passing, file I/O and computation performance of this machine. These results are compared to those obtained on a typical workstation such as a SUN SPARC.

These results are used to design an efficient implementation of a parallel server version of the Pasture Model. This implementation consists of a single master server to process client requests and re-distribute these requests to a number of sub-servers. The Paragon performance results are used to determine what functionality should be implemented in the sub-servers in order to achieve an efficient use of the Paragon and at the same time optimise server response times for client requests.

[1]Department of Mathematics, University of Queensland, Brisbane 4072, Australia

# 1 Introduction

ADVISE [1] is a software package designed to enable large simulation programs, such as those used in environmental modelling, to be run in an efficient, interactive, and user-friendly way. The package is designed using the client-server approach. ADVISE provides the communication interface between a client, typically a graphics user interface and servers which are usually high performance computers.

All previous ADVISE servers have been implemented on uni-processor computers such as SUN SPARC10, ALPHA AXP, SGI workstations and CRAY supercomputers. This paper focusses on the extension of an ADVISE server to run on a parallel machine, in particular, the Intel Paragon. The need for such an implementation becomes clear when one considers the resource requirements of an environmental model, such as the Pasture Model [1]. When run on a SPARC10, the present Pasture Model takes over 720 s to perform the 365 daily simulation steps for one year over the state of Queensland, Australia using a grid size of 10 km. This model computes 29,016 distinct simulations per day. Our aim is to provide the capability to run more complex models over the whole of Australia for 100 years with a grid resolution of 5 km. This will require computational resources equivalent to more than 3,000 such workstations to execute in a similar time to the present model.

Before an optimal porting strategy can be adopted, it is first necessary to understand the performance that can be obtained on the Paragon for the resource required by the Pasture Model. The next three sections discuss these various performance details. It must be assumed that compute nodes will need to receive messages to obtain the necessary input data and send messages to output the state of the model. The first section discusses the performance of the Paragon message passing routines. Environmental models typically access large amounts of input data describing soil, pasture and weather data. Section 3 details the performance of various strategies for efficiently reading these input data files. Section 4 looks at the processor performance with a view to determining some idea of the speed-ups that might be obtainable with the Paragon. Then follows a description of the porting strategy and the performance obtained. The last section summarises the lessons learnt from this project and discusses further areas that could be of interest in future projects.

# 2 Message Passing Performance

This section analyses the message passing performance of the Paragon. Firstly, the message passing performance of **NX**, the native message passing library, is given. Then the implementation of PVM on the Paragon is discussed as well as the performance results. These results are compared with the results obtained using a cluster of SUN workstations.

## 2.1 NX Message Passing

If the times for passing messages between nodes of the Paragon are analysed, the following well known result may be noticed: When several messages of similar length are sent, perhaps

in order to calculate an average message passing time, the first message takes significantly longer to send than the subsequent messages. This initial overhead is associated with the memory paging mechanism. However, the times can usually be made consistent by using the **-plk** option to page-in all the required memory before the program begins execution. This will not work if the message buffers are dynamically allocated at run time since the operating system will not be aware of the dynamic memory requirements at program load time.

It should be noted, however, that it is not usual to send a message with a buffer that has not first been initialised with useful data. If the buffers are first initialised, then the subsequent paging will be done before these data assignment statements and not when the buffer is sent. Hence, the overhead associated in paging on the sending node would not normally be seen in the message passing times. Unfortunately, there is no similar reasoning that can be applied to the receiving node and in a usual program, the paging overhead would normally be incurred in the first message received.

A simple efficient approach that can adopted to remove the paging component from the message passing timings is to allocate memory dynamically using the **calloc** routine on both the sending and receiving nodes. This routine allocates the requested space and then sets all Bytes to zero. In this way all of the necessary paging in both the send and receive nodes is incorporated into the calloc routine. In the case of Fortran timing programs, buffers in both the sending and receiving nodes can be initialised before the timing tests are started.

The message passing times obtained using the NX message passing library are shown in Tables 1 and 2. Table 1 shows the times when the message buffers are assigned using the malloc routine and Table 2 shows the times obtained using the calloc routine. The message passing times shown are round trip times, i.e., the time to send a message of the designated size from the send node to the receive node and for the receive node to send back a 0 Byte message as an acknowledgement. The times have been measured using the Paragon **dclock** timing call. The loop and timing overhead associated with using the timing routine, 1.97 $\mu$s per call, has been accounted for in these timings. All times are in $\mu$s and the average times have been calculated using the times taken for 1000 messages after the initial message.

As can be seen in these two tables, the paging overhead has been shifted from the message passing routines to the allocation routine. It is interesting to note that the paging cost of calling the calloc routine, approximately 48 ms, is less then half the paging cost when sending the first 1,000,000 Byte message of 100 ms. In the case of the calloc routine, the message passing times for the first message are similar to the average times. The 8509 $\mu$s obtained in Table 1 for the first 100 Byte message appears to be unusually high. Further tests were done and the values obtained over a number of runs were all greater than 2600 $\mu$s.

## 2.2 PVM Message Passing

The next tests conducted, measure the times using the message passing routines in PVM [3]. This software package has a master/slave pair of timing programs for this purpose in the suite of example programs supplied. Tests were conducted using:

| | Message Bytes | First $\mu$s | Average $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|---|
| allocate memory (send node ) | | 2905 | N/A | N/A |
| allocate memory (receive node) | | 786 | N/A | N/A |
| | 0 | 108 | 92 | N/A |
| | 100 | 8509 | 116 | 862.1 |
| | 1000 | 131 | 120 | 8333.3 |
| | 10000 | 2520 | 279 | 35842.3 |
| | 100000 | 30691 | 1691 | 59136.6 |
| | 1000000 | 115759 | 15734 | 63556.6 |

Table 1: NX message passing times using malloc

| | Message Bytes | First $\mu$s | Average $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|---|
| allocate memory (send node ) | | 50261 | N/A | N/A |
| allocate memory (receive node) | | 48590 | N/A | N/A |
| | 0 | 111 | 90 | N/A |
| | 100 | 153 | 116 | 862.1 |
| | 1000 | 136 | 121 | 8264.5 |
| | 10000 | 320 | 269 | 37174.7 |
| | 100000 | 1660 | 1624 | 61576.3 |
| | 1000000 | 15206 | 15178 | 65884.8 |

Table 2: NX message passing times using calloc

| Routine | Machine | $\mu$s/call |
|---|---|---|
| gettimeofday | SUN LX | 39 |
| | SPARC2 | 28 |
| | SPARC10 | 12 |
| | Paragon | 497 |
| dclock | Paragon | 1.97 |

Table 3: Loop and timing overhead

- a pair of SUN SPARC2 workstations,
- a pair of SUN SPARC10 workstations,
- a SUN LX and a compute node of the Paragon, and
- a service node and a compute node on the Paragon.

Performance times, as implemented in the PVM timing programs, are obtained using the timing routine **gettimeofday** which is a standard UNIX routine. The loop and timing overhead associated with this call was also measured and these times are shown in Table 3. This loop and timing overhead has been accounted for in all of the message passing timings. What should be noted is the huge overhead (497 $\mu$s) associated with using this routine on the Paragon compared with the times on the SUN workstations and the time for using the Paragon dclock routine (approx 2 $\mu$s). Although not measured, there is also a large variance associated with calling this routine on the Paragon which may impact on accurately comparing program performance.

### 2.2.1 SUN Workstation Timings

Two sets of timings are shown for each of the workstations used for comparisons. Table 4 shows the message packing and message passing performance of a SUN SPARC2 in the case where both the master and slave tasks are on the same workstation. This version of PVM uses UNIX domain sockets for communication between the PVM daemon (pvmd) and tasks as well as between tasks on the same processor when the **Direct Routing** option is specified. It is claimed that a performance increase of from 1.5 to 2 can be expected over previous versions. Table 5 gives the results obtained in the typical situation where the master and slave tasks are on distinct SPARC2 workstations. In this case, messages are passed between the tasks over an Ethernet connection. The maximum throughput of this network is 1.25 M Bytes/s.

Tables 6 and 7 are the results obtained using SUN SPARC10 workstations. These results are interesting in two ways. Firstly, PVM tasks communicating with the TCP Direct Routing option, are now able to utilise up to 80% of the Ethernet network bandwidth. Secondly, tasks

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 2266 | N/A |
| 100 | 270 | 2317 | 43.2 |
| 1000 | 686 | 2451 | 408.0 |
| 10000 | 5299 | 5836 | 1713.5 |
| 100000 | 59544 | 49888 | 2004.5 |
| 1000000 | 734082 | 522441 | 1906.9 |

Table 4: PVM local message passing on SUN SPARC2

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 3534 | N/A |
| 100 | 283 | 3607 | 27.7 |
| 1000 | 711 | 4628 | 216.1 |
| 10000 | 5394 | 13100 | 763.4 |
| 100000 | 55203 | 105875 | 944.5 |
| 1000000 | 512396 | 1013311 | 986.9 |

Table 5: PVM remote message passing on SUN SPARC2

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 1004 | N/A |
| 100 | 75 | 1009 | 99.1 |
| 1000 | 99 | 1059 | 944.3 |
| 10000 | 387 | 2366 | 4226.5 |
| 100000 | 4386 | 18018 | 5550.0 |
| 1000000 | 59159 | 179489 | 5571.4 |

Table 6: PVM local message passing on SUN SPARC10

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 2050 | N/A |
| 100 | 75 | 2119 | 47.2 |
| 1000 | 114 | 3089 | 323.7 |
| 10000 | 532 | 10828 | 923.5 |
| 100000 | 4378 | 99785 | 1002.2 |
| 1000000 | 78776 | 992106 | 1008.0 |

Table 7: PVM remote message passing on SUN SPARC10

on the same processor are now able to achieve over 5.5 M Bytes/s. This may be indicative of the performance that may be achieved by multiple processor SUN workstations using shared memory. The average times shown were calculated over 20 messages.

### 2.2.2 Paragon Timings

In the case of sending messages between the SUN LX workstation and the Paragon, two further situations must be taken into account with respect to messages passed between ADVISE clients and servers. This is due to the different machine representations for integers and floating point data. Firstly, image data sent from the Paragon server to the SUN client is in Byte format and can be transferred without translation. However, floating point data sent from the Paragon or vice-versa must first be translated into a machine independent format before being sent to the other machine. This extra overhead appears in the message packing times. On the Paragon, the paging times associated with the first message are negligible when compared with the message passing times and have not been reported. Table 8 gives the times for passing Byte messages from a SUN LX workstation to the Paragon. Table 9 shows the extra overhead in the packing times when the message contains integer data which must be translated. Table 10 gives the times for the case when the master is a UNIX task on a service node and the slave is on a compute node.

Messages in all three cases are passed as TCP/UDP packets and are routed through the pvmd on a user node of the Paragon. In all cases the message passing times are extremely poor and will have a major impact on the performance that can be expected from the ADVISE package. These performance figures have been passed to the person who is responsible for the Paragon MPP port, but so far no feedback has been received.

The final message passing tests consist of timing communication between two compute nodes on the Paragon. In this case, messages are passed directly between nodes using the Paragon NX message passing routines. The gettimeofday routine was found to be impractical to use in this case because the variance of the time take for this routine was much larger than the message passing times. The test programs had to be rewritten to use the dclock routine. Table 11 shows the performance figures obtained from these programs.

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 63352 | N/A |
| 100 | 247 | 61413 | 1.6 |
| 1000 | 304 | 57726 | 17.3 |
| 10000 | 979 | 160022 | 62.5 |
| 100000 | 7830 | 1304522 | 76.7 |
| 1000000 | 78754 | 12893763 | 77.6 |

Table 8: PVM SUN LX master and Paragon slave, Byte messages

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 59125 | N/A |
| 100 | 276 | 57814 | 1.7 |
| 1000 | 595 | 58345 | 17.1 |
| 10000 | 3737 | 199532 | 50.1 |
| 100000 | 35663 | 1274377 | 78.5 |
| 1000000 | 354405 | 13033763 | 76.7 |

Table 9: PVM SUN LX master and Paragon slave, integer messages

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 110561 | N/A |
| 100 | 443 | 122850 | 0.8 |
| 1000 | 486 | 125525 | 8.0 |
| 10000 | 1325 | 256759 | 38.9 |
| 100000 | 10972 | 1532505 | 65.3 |
| 1000000 | 117681 | 13699936 | 73.0 |

Table 10: Paragon master and slave

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 438 | N/A |
| 100 | 761 | 727 | 137.2 |
| 1000 | 795 | 540 | 1844.4 |
| 10000 | 771 | 1075 | 9281.3 |
| 100000 | 4732 | 6090 | 16415.3 |
| 1000000 | 32844 | 50804 | 19682.5 |

Table 11: Paragon master and slave using PVM send and recv

| Length Bytes | Pack $\mu$s | Send $\mu$s | Transfer Rate K Bytes/s |
|---|---|---|---|
| 0 | N/A | 89 | N/A |
| 100 | N/A | 123 | 801.3 |
| 1000 | N/A | 128 | 7687.3 |
| 10000 | N/A | 279 | 35552.4 |
| 100000 | N/A | 1588 | 62905.9 |
| 1000000 | N/A | 14729 | 67886.3 |

Table 12: Paragon master and slave using PVM psend and precv

The results are rather disappointing when compared to those obtained from using the NX routines directly. The 0 Byte round trip times are 438 and 90 $\mu$s respectively, and the data transfer rates for 1,000,000 Byte messages are 19.7 and 65.9 M Bytes/s, respectively, for the PVM and NX message passing versions.

The **pvm_send** and **pvm_recv** calls are for general use and incur some overhead in transferring data to and from message buffers. There are two routines that are optimised for message passing by utilising the underlying message passing library more efficiently. The **pvm_psend** and **pvm_precv** routines do not allow the packing of various arrays of data but immediately send to or receive from the designated user-defined area. Table 12 shows the timing results obtained by replacing the standard send and receive routines by the more efficient psend and precv routines. Compared with Table 11, the round trip times for 0 Byte messages has decreased by more than a factor of 4 and the data transfer rate has increased by over a factor of three. If these results are compared with those using the NX message passing routines (Tables 1 and 2), it is seen that the performance of the PVM message passing routines is similar to the NX message passing routines.

## 2.3 Conclusions

The following summarises the important points of interest with regard to using PVM on the Paragon and, in particular, how these results impact on porting an ADVISE server to the Paragon:

- It is important to understand the timing routine that is used to gain performance measurements. The implementation on the Paragon of the gettimeofday routine takes on average 497 $\mu$s in comparison with the Paragon dclock routine which takes only 2 $\mu$s. Completely misleading performance results could be obtained if the timing overhead of these calls is not taken into account. In particular, the message passing performance programs supplied in the PVM package use the gettimeofday routine and no attempt has been made in these programs to account for the timing call overhead.

- The message passing times between a SUN workstation and a compute node on the Paragon are unsatisfactory. Using Ethernet, the average 0 Byte round trip time between the Paragon and a SUN LX is about 60 ms compared to about 2ms between two SPARC10 workstations, a factor of 30 times slower. The data transfer times for 1,000,000 Byte messages using the Paragon are slower by a factor of 13 when compared to two SPARC10 workstations.

  In the ADVISE package there are usually at least two distinct calls between a client running on a graphics workstation and a server running a model. One to step the model and at least one other to receive selected data describing the current state. In the Pasture Model, although each image contains 29016 Bytes of data, compression reduces the size of the returned message to about 5 K Bytes. If the message passing times shown in Table 8 are interpolated, the request and subsequent transfer times are about 100 ms per image. If the Pasture Model is run over 365 days and only one parameter is displayed at each step, there will be 730 calls between the client and server. This will take at least (365 x (60 + 100)ms = ) 58.4s to execute. As will be shown in subsequent discussion, this will be the major limitation in obtaining satisfactory performance from the Paragon.

  The implementation of PVM on the Paragon needs further investigation to determine what is causing this poor performance and to make the necessary changes to obtain performance at least equal to that obtained between workstations. It is possible that the problem is not with the PVM implementation. The times obtained using the UNIX **ping** command are also quite poor in comparison to ping times between SUN workstations. The ping times between two SPARC10 workstations is less than 2 ms compared with 8 and 22 ms to ping the Paragon from a SUN workstation.

- Using PVM, the message passing performance between compute nodes on the Paragon is extremely good. It was not possible to measure any overhead associated with passing messages using the PVM psend and precv routines compared with the Paragon native

NX message passing routines. Thus, it is possible to write portable MPP applications using the PVM routines without incurring a message passing performance penalty.

# 3 File I/O Performance

The File I/O performance of the Paragon is detailed in this section. More specifically, the file reading performance is measured using the input datafiles used by the ADVISE Pasture Model server.

Firstly, the datafiles and data storage formats used by the Pasture Model are presented. Next, the file reading performance obtained with a SUN SPARC10 is discussed. The next section gives the file reading performance obtained with the Paragon. Finally, these results are analysed in order to determine the best method to use with regard to a MPP implementation of the ADVISE Pasture Model server.

## 3.1 ADVISE Data File Formats

There are ten data files used as input to the Pasture Model. Three of these files store the constant parameters associated with the pasture and soil and are read in during the model initialisation. One file stores the initial state of the model and is read in when the model is restarted. Another file stores the rainfall data and is accessed for each daily step of the model. To reduce the size of this file, data is stored as 2 Byte integers and converted by the Pasture Model to 4 Byte floating point values. The last five files hold other weather parameters. However, because the Pasture Model is not as sensitive to this data, these parameters are read in only on a weekly basis to reduce the I/O and data storage requirements. The data in these files is stored as Byte values and converted by the Pasture Model to 4 Byte floating point values. This further reduces the size of these data files by a factor of 4.

File compression techniques have also been applied to the data in these files to further reduce the amount of storage needed and to increase the file I/O reading performance. The need for these techniques will become more important when the Pasture Model is run over finer grids and longer time periods. Running this model over Australia using a 5 km grid for 100 years will require a 3,200 fold increase in data storage requirements.

Table 13 shows the sizes of the input datafiles in both uncompressed and compressed format. The compressed versions of the data files require only 25% of the storage required for the uncompressed versions. This lower storage requirement will be of more importance when data storage for the larger models must be considered. There are two factors that influence the file reading performance of the compressed versions of the files in comparison to the uncompressed versions. Firstly, the smaller file sizes will require less file I/O resources due to their smaller size. In this case, the file read time should be only 25% of the time taken to read in the uncompressed versions. Secondly, there is a computation overhead in decompressing the data to its original format. There will be an improvement in the compressed version only if the decompression overhead is less than the improvement in the

| File Name | Read Frequency | Uncompressed Size Bytes | Compressed Size Bytes |
|---|---|---|---|
| pgrass | Initialisation | 3,748 | 642 |
| psoil | Initialisation | 88,144 | 18,752 |
| range | Initialisation | 232,128 | 52,343 |
| input | Restart | 1,392,768 | 726,258 |
| rain | Daily | 21,181,680 | 3,952,370 |
| evaporation | Weekly | 1,508,832 | 593,041 |
| radiation | Weekly | 1,508,832 | 400,699 |
| vapour | Weekly | 1,508,832 | 600,520 |
| mintemp | Weekly | 1,508,832 | 597,294 |
| maxtemp | Weekly | 1,508,832 | 586,815 |
| Total Size | | 30,442,628 | 7,528,734 |

Table 13: Pasture Model Input Files

file reading times. The compression method will thus be more successful when used on a computer with a fast processor and relatively slow disk I/O performance.

## 3.2 SPARC10 File I/O Performance

This section discusses the file read performance results obtained using a SUN SPARC10 to read the ADVISE data files. Programs were written to read in both the uncompressed and compressed files containing the input data for the Pasture Model. These programs read the data in a similar way to the way in which the ADVISE server would also access these data files. i.e. the rain file containing daily data is read using 365 read calls. In this way, it is possible to determine when the file I/O performance will become a bottleneck in obtaining better overall performance from the ADVISE server.

It should also be possible to extrapolate from these performance results to obtain some idea of the performance that might be obtained in running larger models. When determining file I/O performance, it is important to consider what effect disk buffering has on the read times. The SPARC10 workstation used in these tests reads data from files stored on a remote SUN file-server. This workstation mounts a remote file system using **NFS** (Network File System). The SPARC10 has 64 M Bytes of main memory and is able to cache all of the data files in either uncompressed or compressed format. Thus, it is expected that the first timing tests will show the cost of accessing the data files remotely and transferring the data across the network into memory. Subsequent reading of these files will be serviced locally. One extra overhead is that the local NFS must first check with the remote file system that the locally cached data is still valid. If this is not the case, then the data must be transferred from the remote server.

| File Name | Remote access | Local access |
|---|---|---|
| pgrass | 12 | 6 |
| psoil | 107 | 15 |
| range | 430 | 20 |
| input | 3,149 | 92 |
| rain | 34,082 | 1,281 |
| evaporation | 2,274 | 96 |
| radiation | 2,108 | 94 |
| vapour | 2,095 | 94 |
| mintemp | 2,273 | 94 |
| maxtemp | 2,205 | 99 |
| Total Time | 48,615 | 1,891 |
| Transfer Rate | 0.63 M Bytes/s | 16.1 M Bytes/s |

Table 14: SPARC10 Uncompressed file reading performance (ms)

In performing these tests it is necessary that the local NFS cache mechanism be disabled. The reason for doing this is so that the read performance for data files much too large to be cached in local memory can also be determined.

Disabling the cache mechanism is easily achieved by using the UNIX **touch** command from a different workstation. This ensures that subsequent file reading will be done remotely.

Table 14 shows the file reading performance obtained using the uncompressed version of the data files with both local and remote NFS access. The loop and timing overhead of using the gettimeofday routine (11 $\mu$s) has been accounted for in these results. The overall data transfer rates have also been included for reference.

Table 15 shows the file reading times for the compressed data files in the situation where the files have been cached locally. The time to read the data files has decreased from 1,891 ms for the uncompressed files down to only 409 ms for the compressed files. The file reading times have decreased to only 22% of the time taken to read the uncompressed data. This is consistent, since the compressed files are only 25% of the size of the uncompressed files.

However, the time required to decompress this data is 3,677 ms. The total time to read and decompress the data, 4,086 ms, is much greater than the time taken for the uncompressed data. On the basis of these results, one could conclude that the decompression technique takes 216% longer and is thus not time-efficient.

Table 16 gives the results in the case where the compressed files are accessed remotely. Once again, the file read time, 5,866 ms is much lower then the total time, 48,615 ms to remotely read the uncompressed files. The decrease in the file read time is much more than can be accounted for by the decrease in the file sizes. This result is, at present, unexplained.

In this case, the total time taken by the compressed version, 9,543 ms is much lower than that taken by the uncompressed version, 48,615 ms. It can be seen that the decompression

| File Name | Read Time | Decompress Time | Total |
|---|---|---|---|
| pgrass | 4 | 1 | 5 |
| psoil | 6 | 17 | 23 |
| range | 3 | 28 | 31 |
| input | 44 | 200 | 244 |
| rain | 210 | 2,402 | 2,612 |
| evaporation | 29 | 209 | 238 |
| radiation | 22 | 187 | 209 |
| vapour | 31 | 209 | 240 |
| mintemp | 31 | 209 | 240 |
| maxtemp | 29 | 215 | 244 |
| Total Time | 409 | 3,677 | 4,086 |
| Transfer Rate | | | 7.5 M Bytes/s |

Table 15: SPARC10 Local compressed file reading performance (ms)

technique is time-efficient in the case where the data files are not cached locally. This method will therefore also be suitable in the case where the data files are so large that the data cannot be cached locally.

## 3.3 Paragon File I/O Performance

In this section, the file reading performance obtained on the Paragon is given and the results are discussed. Firstly, the times obtained on a single compute node are detailed. The next section discusses the performance obtained on multiple nodes using three different methods. Lastly, these results are used to determine an optimal file reading method for use in the ADVISE server.

### 3.3.1 Single Node Performance

The Paragon computers at ETH and KFA Jülich, on which the tests were conducted, have four different file systems for data storage.

1. a standard UNIX File System (UFS),
2. a Parallel File System (PFS) with a 8 K Byte block size striped over 4 I/O nodes,
3. a PFS with a 64 K Byte block size striped over 3 I/O nodes, and
4. a PFS with a 64 K Byte block size striped over 6 I/O nodes.

The performance of these file systems for various read requests is discussed in [2, Section 3]. However, the results are only listed for read requests that are multiples 8 K Bytes.

| File Name | Read Time | Decompress Time | Total |
|---|---|---|---|
| pgrass | 7 | 1 | 8 |
| psoil | 28 | 17 | 45 |
| range | 42 | 28 | 70 |
| input | 726 | 200 | 926 |
| rain | 2,722 | 2,402 | 5,124 |
| evaporation | 490 | 209 | 699 |
| radiation | 318 | 187 | 505 |
| vapour | 521 | 209 | 730 |
| mintemp | 504 | 209 | 713 |
| maxtemp | 508 | 215 | 723 |
| Total Time | 5,866 | 3,677 | 9,543 |
| Transfer Rate | | | 3.2 M Bytes/s |

Table 16: SPARC10 Remote compressed file reading performance (ms)

The size of the read requests for the input file are 116,064 Bytes. The rain data is read in using 58,032 Bytes requests and the other five weather data files are read using 29,016 Bytes requests. It is necessary to measure the performance of these file systems in the case where the read requests are not in multiples of the stripe size.

Table 17 shows the file reading performance for the UNIX file system. Tables 18, 19, and 20 show the file reading times for the uncompressed data files using all three parallel file systems. These tables show the normal results as well as the results where disk buffering was disabled by touching all the data files.

The following points can be deduced from these tables:

- The read transfer rate of approx 1 M Byte/s for the UFS is consistent with those shown in [2].
- No significant differences were obtained on any of the file systems by disabling the disk caching.
- No single file system gives the best read performance for all data files. From these results, the best performance can be obtained by placing the input and weather files on the 4 stripe 8 K Byte block size PFS and the other files on the UFS.

The poor performance of the 64 K Byte block size PFS on the rain data file is due to the fact that the read requests are for only 58,032 Bytes. This is less than the stripe size of 64 K Bytes. The read requests for the other weather files are only for 29,016 Bytes and the read performance for these files is also poor.

Table 21 gives the results obtained from reading the compressed files from the UFS with caching disabled. Table 22 shows the corresponding results with caching enabled. Although

| File Name | Cache disabled | Cache enabled |
|---|---|---|
| pgrass | 79 | 89 |
| psoil | 123 | 126 |
| range | 249 | 258 |
| input | 1,357 | 1,246 |
| rain | 18,356 | 18,137 |
| evaporation | 1,382 | 1,360 |
| radiation | 1,380 | 1,359 |
| vapour | 1,390 | 1,362 |
| mintemp | 1,401 | 1,482 |
| maxtemp | 1,410 | 1,388 |
| Total Time | 27,127 | 26,806 |
| Transfer Rate | 1.12 M Bytes/s | 1.14M Bytes/s |

Table 17: Paragon UFS uncompressed file reading performance (ms)

| File Name | Cache disabled | Cache enabled |
|---|---|---|
| pgrass | 211 | 204 |
| psoil | 155 | 181 |
| range | 191 | 242 |
| input | 494 | 507 |
| rain | 8,912 | 8,574 |
| evaporation | 1,202 | 1,128 |
| radiation | 1,448 | 1,109 |
| vapour | 1,255 | 1,143 |
| mintemp | 3,020 | 1,203 |
| maxtemp | 1,274 | 1,131 |
| Total Time | 18,161 | 15,421 |
| Transfer Rate | 1.68 M Bytes/s | 1.97 M Bytes/s |

Table 18: Paragon 4 stripe 8 K PFS uncompressed file reading performance (ms)

| File Name | Cache disabled | Cache enabled |
|---|---|---|
| pgrass | 298 | 221 |
| psoil | 208 | 291 |
| range | 271 | 359 |
| input | 538 | 534 |
| rain | 11,122 | 11,107 |
| evaporation | 1,475 | 1,445 |
| radiation | 1,465 | 1,504 |
| vapour | 1,503 | 1,425 |
| mintemp | 1,487 | 1,410 |
| maxtemp | 1,433 | 1,436 |
| Total Time | 19,801 | 19,731 |
| Transfer Rate | 1.54 M Bytes/s | 1.54 M Bytes/s |

Table 19: Paragon 3 stripe 64 K PFS uncompressed file reading performance (ms)

| File Name | Cache disabled | Cache enabled |
|---|---|---|
| pgrass | 343 | 297 |
| psoil | 251 | 278 |
| range | 326 | 337 |
| input | 696 | 1,204 |
| rain | 16,927 | 15,778 |
| evaporation | 1,842 | 1,854 |
| radiation | 1,883 | 3,805 |
| vapour | 1,859 | 1,829 |
| mintemp | 1,875 | 3,038 |
| maxtemp | 2,960 | 1,949 |
| Total Time | 28,962 | 30,369 |
| Transfer Rate | 1.05 M Bytes/s | 1.00 M Bytes/s |

Table 20: Paragon 6 stripe 64 K PFS uncompressed file reading performance (ms)

| File Name | Read Time | Decompress Time | Total |
|---|---|---|---|
| pgrass | 92 | 12 | 104 |
| psoil | 57 | 71 | 128 |
| range | 80 | 51 | 131 |
| input | 687 | 367 | 1,054 |
| rain | 2,706 | 4,517 | 7,223 |
| evaporation | 489 | 394 | 883 |
| radiation | 311 | 350 | 661 |
| vapour | 478 | 397 | 875 |
| mintemp | 477 | 396 | 873 |
| maxtemp | 474 | 395 | 869 |
| Total Time | 5,850 | 6,951 | 12,801 |
| Transfer Rate | | | 2.38 M Bytes/s |

Table 21: Paragon UFS compressed file reading performance without cache (ms)

| File Name | Read Time | Decompress Time | Total |
|---|---|---|---|
| pgrass | 58 | 13 | 71 |
| psoil | 21 | 75 | 96 |
| range | 18 | 51 | 69 |
| input | 92 | 372 | 464 |
| rain | 361 | 4,539 | 4,900 |
| evaporation | 62 | 399 | 461 |
| radiation | 46 | 354 | 400 |
| vapour | 60 | 394 | 454 |
| mintemp | 60 | 397 | 457 |
| maxtemp | 57 | 389 | 446 |
| Total Time | 835 | 6,983 | 7,818 |
| Transfer Rate | | | 3.89 M Bytes/s |

Table 22: Paragon UFS compressed file reading performance with cache (ms)

not included in this report, the read performance using the compressed files mounted on the three parallel file systems was also tested. The results obtained were very poor. This is to be expected because the read requests sizes are approx 25% smaller for the compressed files and thus too small to be efficient on these file systems. Clearly, these tables demonstrate that the fastest read times are obtained using the compressed files mounted on a UFS.

### 3.3.2 Multiple Node Performance

This section discusses methods used to perform parallel file access. For all of these methods it is required that all of the the data in the input files must be available on all nodes. The following three methods were implemented:

1. data is requested on each node using standard UNIX file access routines,
2. data is requested on each node using the Paragon parallel file access routines using the M_GLOBAL option, and
3. one node is designated as the file read node. This node requests data using standard UNIX file access routines and then broadcasts this data to all other nodes.

Table 23 contains the results obtained from these three methods. The total time taken to access the data from all ten input files on each node was measured and the times shown are the maximum of the times taken by each node. The results for the UNIX method were so poor in comparison to the other methods that the tests were stopped after using only 4 nodes. From these results it is clear that this method cannot be used as a means of spreading data across multiple nodes.

The performance of the parallel file access routines is also disappointing when compared with the third method. Parallel file access takes over 30 s longer for 96 nodes than for 1 node. The hand-coded third method using broadcast messages takes only 3 s more.

## 3.4 Conclusions

The following conclusions can be made with regard to choosing an optimal file access method for reading in the data files for the ADVISE server:

- The compressed data file format gives the fastest read performance.
- The UFS gives better read performance than the three Parallel File Systems. This is because the read request sizes are too small to enable the PFS routines to execute efficiently. When larger models are run, the suitability of using the PFS will have to be re-evaluated.
- The most efficient method to spread the input data across multiple nodes is to assign a master node to do all file reading and then broadcast the necessary data to other nodes.

| Number Nodes | UNIX file I/O | Parallel file I/O | Broadcast |
|---|---|---|---|
| 1 | 26.34 | 25.85 | 26.10 |
| 2 | 73.73 | 27.77 | 26.38 |
| 4 | 143.75 | 30.19 | 26.50 |
| 8 | | 34.93 | 27.28 |
| 16 | | 37.06 | 29.76 |
| 24 | | 42.98 | 28.50 |
| 32 | | 42.27 | 30.52 |
| 40 | | 46.39 | 28.78 |
| 48 | | 45.51 | 28.60 |
| 56 | | 48.09 | 28.47 |
| 64 | | 51.34 | 29.05 |
| 72 | | 52.54 | 29.00 |
| 80 | | 53.33 | 30.46 |
| 88 | | 54.93 | 29.17 |
| 96 | | 56.59 | 29.14 |

Table 23: Paragon UFS parallel file read performance (s)

Although not tested, there are further areas where potential speed-ups may be possible.

- Spreading the datafiles across several I/O nodes and assigning more than one node to perform file reading.
- Use asynchronous file access so that file reading and decompression can be overlapped.

# 4 Paragon Processor Performance

This section discusses the performance that might be expected from the I860 processors used on each node of the Paragon. These results will give an insight into the performance that could be expected from the parallel version of the Pasture Model.

The uni-processor version of the ADVISE Pasture Model server was ported to run on one node of the Paragon. Tests were conducted using various optimiser compilation flags. The best performance was obtained using the following set of flags:

-O4 -Mvect -Knoieee -Mstreamall

Table 24 gives the time taken to run the Pasture Model over 365 days. As can be seen, the performance of the I860 is somewhere between the SPARC2 and SPARC10. It must be pointed out that these results should not be used as a benchmark. No attempt was made to optimise the code to suit the I860 with regard to instruction pipelines, data caches, etc.

| Processor | Run Time (s) |
|---|---|
| SPARC2 | 878.4 |
| SPARC10 | 414.9 |
| Paragon I860 | 660.8 |

Table 24: Pasture Model run times for 365 days

From these results it can be deduced that if the Pasture Model were to be run on 64 or 96 processors then the minimum computation time would be 6.5 or 4.3 s respectively. This is an upper bound on the performance that can be expected on the Paragon when executing the Pasture Model. Of course, the overhead of passing messages and unequal load sharing between processors may add considerably to these times.

# 5 The Parallel Pasture Model

This section details the strategy used to implement a parallel version of the Pasture Model on the Paragon. The performance obtained from this implementation is then shown and improvements and limitations to this implementation are discussed.

The major computation part of the Pasture Model consists of a simulation routine that is applied to a large number of grid cells. There is no communication between grid cells. This type of code is often referred to as being *embarrassingly* parallel. The parallelisation of such programs usually involves partitioning the set of grid cells into distinct subsets and then assigning each subset to a processing node. However, there are some extra complications with the Pasture Model and the ADVISE server that must also be taken into account.

Firstly, there is the problem of getting the input data to each node. If the data were stored in uncompressed form, it may be possible to make each node responsible for reading the necessary data with the parallel file I/O routines and using the **M_SYNC** option. However, as discussed in a previous section, the most efficient method for reading this input data is to use compressed versions of the files. This compressed format will not be suitable for use with this file reading method. The most promising method seems to be to assign an extra node to handle all of the file I/O operations and then to farm the data out to the respective compute nodes. This has further advantages in that it may then be possible to overlap the file reading and decompression tasks with the computation tasks. Furthermore, if the file access operations become a bottleneck to obtaining better performance, it may be possible to assign two or more processors to the file I/O tasks. For example, one processor would read only the rain data file and another processor would read all the other data files. The reading times shown in Table 21 indicate that this would result in a reasonable sharing of the file reading and decompressing operations.

Secondly, the parallel version of the ADVISE server must provide exactly the same services and use the same interface protocol as the uni-processor versions. One solution is to

adopt a master-slave approach in which one node would appear to ADVISE clients as a normal sequential ADVISE server. However, this node would farm out the computation load among a number of other nodes.

One complication, with this simple approach, is that at the end of each computation step all of the exportable data that may be requested from a client is only available on these computation nodes. A solution would be for each computation node to send back all exportable data at the end of each computation step. However, one of the original goals for ADVISE was to implement an efficient environment. Sending back the complete data set back to the master node may be a significant overhead, especially in the cases where the client only makes subsequent data requests for a small number of parameters. In the case of the Pasture model, there are 18 exportable parameters. Passing all of this data back would involve over 2 M bytes of message passing per step, that is, 730 M Bytes per year.

Another solution is to implement a two level server structure to the ADVISE server. The master server, in this case is responsible for handling all client requests. Each compute node is also a Pasture Model server. However, each of these servers is only responsible for a small subset of the overall model. The master server then has the task of reformulating each client request into a set of requests to be sent to the pasture Model sub-servers and, as well, concatenating the replies from these servers into a single reply to the client. The disadvantage with this is that each client request will result in further messages to one or more sub-servers. As well, formulating all of the sub-requests may be a non-trivial task.

The approach adopted was to implement a solution somewhere between the simple master-slave and the complete sub-server approach. There is a master node that is responsible for handling all client requests. Each compute node acts as a sub-server. However, these sub-servers provide only a subset of the services that the master server must provide. These services are: Initialise, Restart, Step, Terminate and a modified GetData service.

The master server handles all of the data requests from a client in the following way: There is storage allocated on the server for all exportable parameters as well as a flag to indicate the validity of each parameter. After an Initialisation, Restart or a Step request all parameters are marked as invalid. When a data access request is made for a parameter marked as invalid, the master server must first request this parameter from all of the sub-servers. Once this is done, the parameter is marked as valid and all further data requests for this parameter can then be serviced entirely within the master server.

This solution minimises the message traffic between the master and sub-servers by only requesting data from the sub-servers in response to client requests. The disadvantage is that the master server has a computation component in the case where image data is requested. The conversion, which is parallelisable, is done only by the master server.

The master server is also responsible for handling all of the file I/O and decompression tasks.

The first step is to determine the compute performance of the parallel server for different numbers of sub-servers. A test client was written to test the performance of the different remote procedure calls (**rpc**). The times taken to execute the step routine for a step size of 365 days for different numbers of sub-servers was measured. The results are shown in Table

25. The best speed-up of 22.1 was obtained using 60 sub-server nodes.

The file reading and decompression tasks done by the master server are a bottleneck to providing better speed-ups. Table 22 shows that the time taken to perform these tasks on the weather data files is 7.1 s in the case where disk caching is enabled. If this sequential component is subtracted from the 1 and 60 processor times a speed-up of (538.8 / 17.6 = ) 30.6 is obtained.

All of the arrays in the sub-server were dimensioned to the full length of 29016. This was so the server could be tested with different numbers of sub-servers. However, it is possible that this will cause more cache-misses in the processor data cache. The effect of array dimension was also measured. All data arrays in the program were re-dimensioned to (29016 / 63 = ) 461. The execution time then decreased from 25.0 to 22.5 s. If allowance is made for the file I/O and decompression times, the speed-up becomes (538.8 / 15.4 = ) 35.4.

One significant problem encountered is the long times associated with loading programs onto nodes. If the Paragon is rebooted the initialisation times range from 2.35 s for 1 sub-server to 9.8 s for 63 sub-servers. However, after the Paragon has been running for an extended period of time executing several other users programs, the initialisation time goes up to over 30 s.

The poor message passing performance between a workstation and the Paragon has already been discussed in a previous section. The impact that this performance has on the interactive response was also measured. The average time taken to perform the 100 Getinfo requests was 65.8 ms compared to 5.8 ms for a server running on a SPAR10 workstation. The time taken to execute the model using 63 sub-servers over 365 days using 365 Step calls with a step size of 1 day was 59.7 s compared to 22.5 s for a single Step request with a step size of 365 days. The difference in these times is the overhead in performing the extra 364 rpc's. This gives an average of 102 ms per call.

There are still several areas where performance may be improved.

1. The file I/O and decompression tasks for the weather data could be moved out of the master server and performed on two other nodes. These nodes could read and decompress data ahead of the sub-servers.

2. The dataset used in these tests is for the area of Queensland, Australia. The rectangular grid contains about 50% ocean grids. There is no computation done on ocean grid points. However, the allocation routine assigns grid points to sub-servers without taking ocean grid points into account. This results in an uneven distribution of computation amongst the sub-servers. A better method is to give each sub-server the same number of non-ocean grid points. This would result in better load-balancing between sub-servers.

3. A major modification to the compressed file format should make much faster file I/O possible. In the present format 2 read operations are required to read each compressed vector. The first, to obtain the length of the vector and the second to read in the compressed data. A possible improvement is to store a large table at the head of the

| Processor | Run Time (s) | Speed-up |
|---|---|---|
| 1 | 545.9 | 1.00 |
| 2 | 369.0 | 1.48 |
| 3 | 253.3 | 2.16 |
| 4 | 189.3 | 2.88 |
| 6 | 143.9 | 3.79 |
| 8 | 111.0 | 4.92 |
| 12 | 78.2 | 6.98 |
| 16 | 62.1 | 8.79 |
| 20 | 52.1 | 10.48 |
| 24 | 45.3 | 12.05 |
| 28 | 39.4 | 13.86 |
| 32 | 36.9 | 14.79 |
| 36 | 32.5 | 16.80 |
| 40 | 31.4 | 17.39 |
| 44 | 29.3 | 18.63 |
| 48 | 28.2 | 19.36 |
| 52 | 26.2 | 20.84 |
| 56 | 25.8 | 21.16 |
| 60 | 24.7 | 22.10 |
| 63 | 25.0 | 21.84 |

Table 25: Parallel Pasture Model, Step rpc times for step size = 365 days

file containing the number as well as the lengths of all the compressed vectors in the file. This table could be read in with a single read request and the each compressed vector could be read using only a single read. It would also then be possible to read in several vectors at once, reducing even further the number of read requests. This would have a major impact when using the Parallel File System, since read requests could be made large enough to ensure efficient use of this file system.

# 6 Summary

One lesson learnt in this report is to pay careful attention to the timing routines used to measure performance and to take the overhead in using these calls into account. The PVM 0 Byte Round Trip Time was originally reported as over 1,000 $\mu$s. However, about 600 $\mu$s of this time was due to the cost of calling the gettimeofday routine.

One must also be careful in using performance measuring programs written for general use. The above message passing times were reduced from 438 $\mu$s to only 89 $\mu$s by using more efficient routines. Thus, the times eventually obtained were better by over a factor of 10 over those reported by the original program!

The node-node PVM message passing performance is excellent. It was not possible to measure any overhead in using the PVM routines over the native NX routines.

On the other hand, messages through a pvmd on the Paragon suffer a large performance penalty when compared with passing messages between workstations. Further work should be done to investigate and, if possible, remedy the problem. This performance is the major bottleneck to achieving adequate performance with the ADVISE software.

This report has shown that speed-ups of over 22 have been obtained with a parallel version of the Pasture Model. Speed-ups of over 35 could be achieved by allocating the file I/O and decompression tasks to separate nodes. Further speed-ups might be achieved by using a better load-balancing method.

The main obstacle to getting further performance increases is that the size of the problem is too small. The file I/O performance will improve if the size of the data sets is such that compressed vectors contain more than 64 K Bytes. This will allow the use of the Parallel File System where a peak performance of over 9 M Bytes/s is possible compared to about 1 M Byte/s for the standard UNIX File System.

The weather data sent to sub-servers contains either 1 or 2 Bytes per grid point. If 63 sub-servers are allocated, each rain message is only 922 Bytes. From the results in the message passing section, the time taken to send this message will be 45 $\mu$s for the latency + 13 $\mu$s to transfer the data, a total of 58 $\mu$s per message. Other weather data, containing only 1 Byte per grid point will take 52 $\mu$s to send. However, if a model containing four times more grid points is executed, the message passing times should only increase to 97 $\mu$s and 73 $\mu$s respectively. This is an increase of less than 2 in the message passing overhead.

It is possible then that much greater speed-ups will be possible when the number of grid points is increased.

## 6.1 Acknowledgements

All of this work was done whilst I was an academic guest of the Seminar für Angewandte Mathematik at ETH Zürich for three months. I gratefully acknowledge the financial support provided by the Intel Corporation.

I would like to thank Prof. Dr. Rolf Jeltsch for helping to arrange this visit, and for the support and advice of one of my supervisors, Prof. Kevin Burrage.

I would also like to thank Dr. Bert Pohl from the Seminar für Angewandte Mathematik and Dr. Michael Vollmer from Intel for many fruitful discussions on parallel and high performance computing.

Many thanks also go to Nicholas Ironmonger, Bruno Löpfe and Dr. Hanspeter Scherbel for attending to the many many requests in system and network administration that were needed to provide me with such an efficient computing environment in which to do this research.

# References

[1] ADVISE - Agricultural Developmental Visualisation Interactive Software Environment L. Lau, M. Rezny, J. Belward, K. Burrage, B. Pohl, CONPAR 94 - VAPP VI conference proceedings, 1994.

[2] Paragon System Software Release 1.2 Release Notes. June 1994.

[3] PVM 3.3 Users'guide and Reference Manual, A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, ORNL/TM-12187, Oak Ridge National Laboratory, Tennessee, 1994.

# List of Tables