Supercomputer Performance with a Workstation Cluster on Dense Matrix Multiplication

Alberto Ferreira de Souza [*] and Fábio Daros de Freitas [†]

Abstract

This paper shows that a *Workstation Cluster* - a group of networked workstations - with a few number of workstations can achieve supercomputer performance on dense matrix multiplication problem. Dense matrix multiplication is used in a variety of applications and is one of the core components in many scientific computations. Using experiments it was possible to show that a *Workstation Cluster* compound of four DEC-2000 Alpha AXP could reach a performance only 4.2 times lower than a CRAY Y-MP2E/232 supercomputer, when applied to dense matrix multiplication problem with more than 700 lines.

Key words: Algorithms, Scientific computing, High performance computing.

1 Introduction

Using a proper software infrastructure, a *Workstation Cluster* - a group of networked workstations - can emulate a multiprocessor. In this way, some projects are being conducted by the goal of developing software tools that make it possible to use a set of interconnected machines as a concurrent programming platform [1,2,3,4].

The PVM (Parallel Virtual Machine) is one of such software tools [5,6]. The PVM system is a software platform that supports concurrent programming in a network environment. PVM emulates a multiprocessor with distributed memory at a heterogeneous collection of UNIX computers linked by a network, by means of supporting a complete message-passing model.

In this work, the PVM was used as a concurrent programming tool for the dense matrix multiplication task. The dense matrix multiplication is used in a wide variety of applications, and is the core of the numerical solutions of many scientific problems. The multiplication of two NxN matrices requires $O(N^3)$ floating point operations in a sequential machine. Because it is a hard computing problem, the development of efficient distributed algorithms is taking great interest [7,8,9]. This work also presents a parallel algorithm for the dense matrix multiplication in a *Workstation Cluster* using a bus topology network. In the largest used configuration - four DEC-2000 Alpha AXP workstations - a performance only 4.2 times lower than a CRAY Y-MPE2E/232 was achieved.

This paper is divided in five parts. After the introduction, the PVM tool is presented in a more formal way. Next, the proposed matrix multiplication algorithm is discussed, followed by the presentation of the experiment and results. In the last part some remarks about the experiment results are presented.

2 The PVM

The PVM is a software tool that enables a set of networked UNIX computers to act as a single parallel computer. PVM supplies the functions to start tasks and lets the computers communicate and synchronize with each other. It survives the failure of one or more connected processors and allows users to make their applications fault tolerant. Users can write applications in C or FORTRAN and parallelize them by calling simple PVM message-passing routines such as *pvm_send()* and *pvm_recv()* [10].

The PVM supports heterogeneity at the application, machine and network level. All the necessary data conversion for the communication between two different machines, like different integer or floating point representations, are handled by the PVM. It also provides routines for packing and send messages between tasks.

2.1 The PVM Computing Model

The computation model of PVM assumes that any PVM task can send messages to any other PVM task, and that there is no limit to the number or size of such messages. The communications model of PVM provides asynchronous blocking send, asynchronous blocking receive and non-blocking receive functions: an asynchronous blocking send returns as soon as the send buffer is free for reuse regardless of the state of the receiver; a non-blocking

[*]Universidade Federal do Espírito Santo, Centro Tecnológico, Departamento de Informática, PO Box 01-9011, 29060-970 - Vitória - ES - Brazil, alberto@inf.ufes.br.

[†]Universidade Federal do Espírito Santo, Centro Tecnológico, Departamento de Informática, PO Box 01-9011, 29060-970 - Vitória - ES - Brazil, freitas@inf.ufes.br.

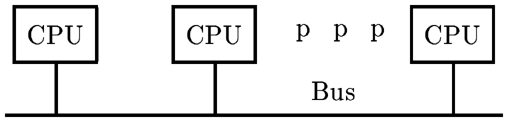

Figure 1: Bus Topology Network

receive immediately returns with either the data or with a flag indicating that data has not arrived yet; and a blocking receive returns only when data is in the receive buffer. In addition to these point-to-point communication functions, the model supports multicast for a set of tasks and broadcast to a user defined group of tasks. PVM guarantees that messages order is preserved between any pair of communicating entities.

2.2 The PVM Implementation

The PVM system is composed of two parts [11]. The first one is an UNIX daemon called **pvmd3**, that resides in all the computers making up the virtual machine - a parallel computer with distributed memory logically implemented by the interconnection of various networked machines. The second part of the system is a library of PVM interface routines. This library contains user callable routines for message passing, process creation, coordinating tasks, and monitoring and modifying the virtual machine. Application programs must be linked with this library to use PVM.

In the next section, the algorithm developed for dense matrix multiplication in a *Workstation Cluster* using a bus topology network will be presented.

3 The Algorithm

The dense matrix multiplication algorithm developed for the implementation of the experiments allows the overlap of processing and communication.

The designed algorithm follows a *master/slave* architecture for a distributed system with bus topology (see Figure 1). In this kind of network topology a unique physical communication media is shared among all the machines, in a way that just one message can be transmitted at a time. If two CPUs are trying to transmit a message at the same time, a collision occurs. The collisions degenerate the network performance, hence, mechanisms for minimizing their occurrences were included in the algorithm.

The task of the *master* in the algorithm is basically to distribute the matrices **A** and **B** to the *slaves* and to catch

```
void
master ( float A[MDIM][MDIM], float B[MDIM][MDIM],
         float C[MDIM][MDIM], int N, int nproc)
{
  int LI, LIN, LOUT, i, j, fulano;
  int *tids;
   tids = (int *) malloc (sizeof (int) * nproc);
   pvm_spawn ("slave", (char**) 0, 0, "", nproc, tids);
   pvm_initsend (PvmDataInPlace);
   pvm_pkint (&N, 1, 1);
   pvm_mcast (tids, nproc, 102);
   for (i = 0; i < N; i++) {   /* Send B Matrix */
      pvm_initsend (PvmDataInPlace);
      pvm_pkfloat (B[i], N, 1);
      pvm_mcast (tids, nproc, 101);
   }
   LIN = 0;
   LOUT = 0;
   LI = 0;
   for (j = 0; j < 2; j++) {    /* Send Initial Lines of A */
      for (i = 0; (i < nproc) && (LOUT < N); i++) {
         pvm_initsend (PvmDataInPlace);
         pvm_pkint (&LOUT, 1, 1);
         pvm_pkfloat (A[LOUT], N, 1);
         pvm_send (tids[i], 99);
         LOUT++;
      }
   }
   do {
      pvm_recv (-1, 100);       /* Receive a Line of C */
      pvm_upkint (&LI, 1, 1);
      pvm_upkint (&fulano, 1, 1);
      pvm_upkfloat (C[LI], N, 1);
      LIN++;
      if (LOUT < N) {          /* Send a Line of A */
         pvm_initsend (PvmDataInPlace);
         pvm_pkint (&LOUT, 1, 1);
         pvm_pkfloat (A[LOUT], N, 1);
         pvm_send (fulano, 99);
         LOUT++;
      }
      else {                   /* Terminate Slaves */
         for (i = 0; i < nproc; i++) {
            pvm_initsend (PvmDataInPlace);
            pvm_pkint (&LOUT, 1, 1);
            pvm_send (tids[i], 99);
         }
      }
   }while ((LOUT != N) || (LIN != N));
   pvm_exit ();
}
```

Figure 2: The Implementation of the *Master* in C and PVM

from them the resulting matrix **C**. The slaves are reserved to the task of realizing the numerical operations related to the multiplication. The algorithm is very simple and can be outlined by the following steps:

Master's **Algorithm**

1. Create *slave* processes;
2. Send the matrix **B** to all the *slaves* (*broadcast*);
3. Send two distinct lines of the matrix **A**, $A[i]$ and $A[i+1]$, to each *slave*, indicating its order in the message, i;
4. Receive a specific line of the matrix **C**, $C[j]$, from one of the *slaves* (the line of **C** computed from matrix **B**

and line $A[j]$);

5. Send a new line of **A** to the *slave* that has sent the line of **C** in the previous step;

6. Repeat the steps 4 and 5 until the whole matrix **A** has been sent and the whole matrix **C** has been received;

7. Send a message to the *slaves* terminating them.

Slaves's Algorithm

1. Receive matrix **B** from the *master*;

2. Receive a line of matrix **A**, $A[i]$, from the *master*;

3. Multiply line $A[j]$ by the matrix **B** producing one line of **C**, $C[j]$;

4. Send line $C[j]$ to the *master*;

5. Repeat the steps 2, 3 and 4 until receiving the terminate message;

The overlap of processing and communication is achieved by the distribution scheme of the lines of matrix **A**. Sending at first, sequentially, two lines to each slave, implies that each *slave* will receive its lines in a distinct time, and that differences between the times each pair of *slaves* receives its lines is equal to the transmission time of two lines. At the time of receiving one of the two initial lines, the *slave* i (any of them) begins immediately to compute one line of matrix **C**; the other line sent remains in the message buffer of the local PVM daemon (**pvmd3**). When the *slave* i sends the first line of matrix **C** computed by it to the *master*, this immediately sends it a new line of matrix **A**. When this line arrives at the *slave*, it, in fact, is already computing a new line of matrix **C**, with the second line of matrix **A** initially received from *master*. The new line of matrix **A** send by the *master* will remain in the **pvmd3** buffer.

Through the artifice of sending two lines at first, the algorithm achieves the overlap of processing and communication. When supposed that the processing times of the *slaves* are equal, the master will receive the *slave's* lines at the same cadence it sends them, which guarantee absence of collisions in the network, since there are no other traffic sources.

If the time necessary to compute one line of **C** were greater than TO:

$$TO = 2.tl.nprocs \tag{1}$$

where tl is the time to send one line and $nprocs$ is the number of processors, then the theoretical speedup, S, will be:

```
void
slave ( float A[MDIM], float B[MDIM][MDIM],
        float C[MDIM])
{
  int N, LI, LF, BL, nproc, i, j, k;
  int tids[10];
  int master, me;

   master = pvm_parent ();
   me = pvm_mytid ();
   pvm_recv (master, 102);
   pvm_upkint (&N, 1, 1);
   for (i = 0; i < N; i++) {      /* Receive B Matrix */
       pvm_recv (master, 101);
       pvm_upkfloat (B[i], N, 1);
   }

   while (1) {
       pvm_recv (master, 99);   /* Receive a Line of A */
       pvm_upkint (&LI, 1, 1);
       if (LI == N)             /* Terminate Mensage ? */
           break;
       pvm_upkfloat (A, N, 1);

       for (k = 0; k < N; k++) {
           C[k] = 0.0;              /* Makes a Line of C */
           for (j = 0; j < N; j++)
               C[k] += A[j] * B[j][k];
       }
       pvm_initsend (PvmDataInPlace);
       pvm_pkint (&LI, 1, 1 );
       pvm_pkint (&me, 1, 1 );
       pvm_pkfloat (C, N, 1);
       pvm_send (master, 100); /* Send a Line of C */
   }
   pvm_exit ();
   return (0);
}
```

Figure 3: The Implementation of the *Slave* in C and PVM

$$S = \frac{N.tpC}{tB + 2.tl.nprocs + tT.nprocs + \frac{N.tpC}{nprocs}} \tag{2}$$

where tB is the time necessary to send the matrix **B**, tT is the time to send a terminating message, N is the number of lines of the resulting matrix **C**, and tpC is the time needed to compute one line of matrix **C**. In the equation 2, the numerator is the time needed for the sequential multiplication. In the denominator, the first term of the sum represents the broadcast of matrix **B**, the second one is owing to mechanism that guarantees the overlap of the processing and communication, the third one is owing to the terminating procedure of the *slaves*, and the fourth one is owing to the computing of the lines of matrix **C** on the *slaves*.

If the time to compute one line of **C** were less than TO, the speedup would depend strongly on the network performance.

It is easy to verify with the help of the equation 2 that the speedup would be equal to the number of processors if the time necessary to send one line of matrix **A** were much less than the time needed to compute one line of matrix **C**.

```
void
mul_seq (float A[MDIM][MDIM], float B[MDIM][MDIM],
         float C[MDIM][MDIM], int N)
{
   int i,k,j;

    for (i = 0; i < N; i++)
        for (k = 0; k < N; k++) {
            C[i][k] = 0.0;
            for (j = 0; j < N; j++)
                C[i][k] += A[i][j] * B[j][k];
        }
}
```

Figure 4: Sequential Algorithm in C

N	Time Seq.	Time Par.	Speedup
100	0.333792	1.828224	0.182577
150	1.149328	3.118096	0.368599
200	2.813584	4.280112	0.657362
250	5.691584	6.029280	0.943991
300	10.329488	6.799344	1.519189
350	17.070272	9.185440	1.858405
400	26.015615	12.505168	2.080389
450	37.629520	16.306065	2.307701
500	52.235214	20.346081	2.567335
600	90.984383	31.564129	2.882525
700	144.513382	46.895569	3.081600
800	216.649033	67.098579	3.228817
900	335.889709	92.379662	3.635970

Table 1: Execution Time on *Workstation Cluster* with 4 DEC-2000 (seconds)

This algorithm was designed to adjust to a bus topology network, although it may present equal or better results with networks with more favorable topology, for example the star topology.

In the next section the experiments and the obtained results are presented.

4 Experiments and Results

In Figure 2 is shown the implementation of the *master* process of the matrix multiplication algorithm in C language and PVM. The implementation of the *slave* process is presented in Figure 3.

As can easily be seen, the number of messages used by the algorithm is equal to $3 * N^2$ plus one termination message for each *slave* process. The number of floating point operations executed by the *slaves* (added) is equal to the number of operations executed by the sequential matrix multiplication algorithm: $2 * N^3$, as can be seen in the Figure 4. Memory usage is $2 + N + N^2$ in each *slave*; in the sequential algorithm, $3 * N^2$.

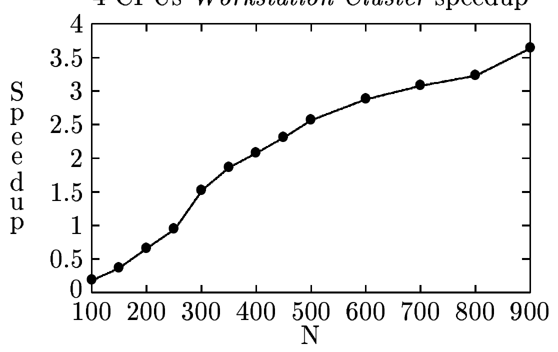

Figure 5: *Speedup* versus N

The codes of the *master*, the *slaves* and the code of the sequential algorithm were compiled on the native compiler of one of the DEC-2000 Alpha AXP machines that were used in the experiments, with the optimization flag $-O3$. The execution times on a *Workstation Cluster* with 4 DECs for various matrix dimensions are presented in the Table 1. In the graphical presentation of Figure 5, can be observed the speedup obtained with the concurrent implementation versus the sequential implementation. The communication cost is very high - the concurrent implementation only is superior to the sequential implementation in the case of square matrices with more than 250 lines. However, beyond this point the *speedup* quickly approaches the theoretical maximum. The experiments were carried out on the workstations without load or additional network trafic. One of the machines was runing the code of the *master* and the code of one of the *slaves*.

In the Table 2 are presented the execution times of the sequential version on a CRAY Y-MP2E/232 machine; a supercomputer with 2 processors, 32 Mwords of memory, clock cycle of 6 nanoseconds and maximum velocity of 330 MFLOPS per CPU [12]. The machine utilized in the experiments is the CRAY that is installed on the Centro Nacional de Supercomputação - CESUP, at Universidade Federal do Rio Grande do Sul.

The sequential program was slightly modified in order to allow vectorization. It was compiled on the CRAY's native compiler with the optimization flag $-O3$. The compiling directives were not inserted in the source code. Using optimization tools, it was verified that the code was not parallelized, this means that just one CPU was used during the experiments.

The graphical presentation of Figure 6, shows the relationship between the time spent to multiply matrices of different sizes in a *Workstation Cluster* with 4 DECs and the CRAY. As the graph shows, for square matrices with more than 700 lines, a *Workstation Cluster* with 4 DECs is just 4.2 times slower than the CRAY, approximately. This result shows that *Workstation Cluster* with about 17 CPUs may have a CRAY performance.

N	Time	N	Time
100	0.053500	400	2.330000
150	0.153000	450	3.270000
200	0.342000	500	4.210000
250	0.581000	600	7.350000
300	0.980000	700	11.10000
350	1.540000	-	-

Table 2: Execution Time on a CRAY Y-MP2E/232

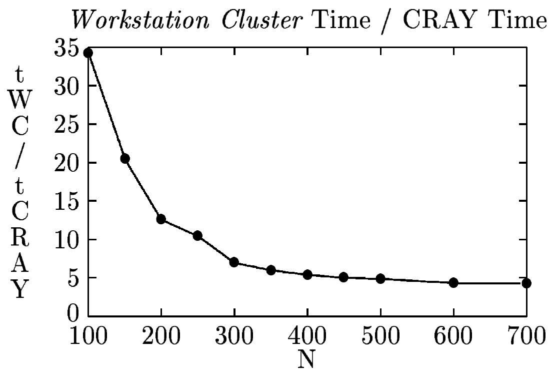

Figure 6: *W. C.* Time / CRAY Time, versus N

5 Conclusion

Using the presented experiments it was possible to show that a *Workstation Cluster* compound of four DEC-2000 Alpha AXP workstations and PVM could reach a performance only 4.2 times lower than a CRAY Y-MP2E/232 supercomputer, when applied to dense matrix multiplication problem with more than 700 lines.

Dense matrix multiplication problem is of special interest to Grupo de Redes Neurais of Departamento de Informática at Universidade Federal do Espírito Santo (GRN/DI/UFES) because it forms the central functional part in trained neural networks. The main motivation of this work was to make available a machine with high processing capabilities to be used in neural networks experiments.

Workstations Clusters are not a panacea, but they are, today, a real choice to high performance processing. Some time ago, because of the high communication latencies on networks and the low rate of this communication, in relation to the processing capabilities of CPUs, *Workstation Clusters* were just an alternative to MPPs [13]. Nowadays, with the advancing of network technologies, they are a threat to the existence of MPPs [14].

Acknowledgments

We thank Hannu Ahomen, Jane Méri Santos and Adilson Bento Pereira for their help in preparing this article.

References

[1] C. Kaliher, "*Cooperative Processes Software*", Proc. 209 Am. Inst. of Physics Conf., AIP, New York, 1990, pp. 364-371.

[2] N. Carriero, D. Gelernter, "*Linda in Context*", Communications of the ACM, 32(4), April 1989, pp. 444-458.

[3] E. Lusk et al., "*Portable Programs for Parallel Processors*", Holt, Rinehart and Winston, New York, 1987.

[4] J. Flower, A. Kolawa and S. Bharadwaj, "*The Express Way to Distributed Processing*", Supercomputing Review, May 1991, pp. 54-55.

[5] V. S. Sunderam, "*PVM: A Framework for Parallel Distributed Computing*", Journal of Concurrency: Practice and Experience, 2(4), December, 1990, pp. 315-339.

[6] V. S. Sunderam, G. A. Geist, J. Dongarra and R. Manchek, "*The PVM Concurrent Computing System: Evolution, Experiences, and Trends*", to be published on the Parallel Computing magazine, also available via ftp on netlib2.cs.utk.edu at name pvm3/pvm-eet.ps.

[7] H. Gupta, P. Sadayappan, "*Communication Efficient Matrix Multiplication on Hypercubes*", 6th Annual ACM Symposium on Parallel Algorithms and Architectures, New Jersey, June, 1994.

[8] A. Gupta, V. Kumar, "*Scalability of Parallel Algorithms for Matrix Multiplication*", Proceedings of the 1993 International Conference on Parallel Processing, vol. 3, pp 115-123.

[9] J. Berntsen, "*Communication Efficient Matrix Multiplication on Hypercubes*", Parallel Computing, vol. 12, 1989, pp. 335-342.

[10] A. Beguelin, J. Dongarra, A. Geist, V. Sunderam, "*Visualization and Debugging in a Heterogeneous Environment*", Computer, June 1993, pp. 88-95.

[11] A. Geist, at al., "*PVM 3 User's Guide and Reference Manual*", Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, September, 1994.

[12] R. Teodorowitsch, "*Otimização do Uso do Supercomputador CRAY Y-MP2E/232, Segunda Edição*", Internal publication of the CESUP/UFRGS, October, 1993.

[13] M. Furtney, G. Taylor, "*Of Workstations & Supercomputers*", IEEE Spectrum, May 1994, pp. 64-68.

[14] B. Furht, "*Parallel Computing: Glory and Collapse*", Computer, November 1994, pp. 74-75.