Dynamic Load-Balancing Algorithm Porting on MIMD Machines

Francisco J. Muniz

Centro de Desenvolvimento da Tecnologia Nuclear - CDTN/CNEN,
C. P. 941, 30123-970 - Belo Horizonte - MG, Brazil
muniz@cdtn.br

Abstract. This paper describes the porting strategies and the implementation of a dynamic load-balancing mechanism over the PVM library. Such load-balancing mechanism, the Extended Gradient approach, is found in the open literature. The implementation was done using the 'C' programing language, running over Linux/X86 compute nodes. Some results that validate the usefulness of the load-balancing system are presented. The conclusions are general and not restricted to any particular architecture of distributed-memory MIMD (Multiple Instruction, Multiple Data) machines.

1 Introduction

On mono-processing systems, to make decisions of where to place a process (to be executed) is quite obvious. However, on a parallel machine such decision has to be better elaborated. Mechanisms that distribute the processing demand over a cluster are called load-balancing mechanisms. In particular, mechanisms with abilities to start or to move processes at run-time are referred as dynamic load-balancing mechanisms. A comprehensive survey on existing resources management approaches is found in a literature review [1]. The load-balancing mechanism implemented in this paper is dynamic, since it does the distribution of new processes at run-time.

2 Dynamic Load-Balancing Policy

The main objective of this section is to introduce the dynamic load-balancing mechanism whose porting is the major focus on this paper.

2.1 Gradient Model Mechanism

The Gradient Model (GM) mechanism [2] is a locally-distributed scheme with a global placement or migration space. Periodic exchange of load-balancing information is restricted to a small and pre-specified subset of other nodes in the cluster (also referred as valency of a node), therefore a potentially scalable mechanism. The processor status is determined by the local processor availability, i.e.

a measure of the loading level of the processor, and the status of its neighbour nodes - this status represents the logical distance (number of intermediate neighbour nodes plus one) to the nearest idle node in the cluster. The local processor availability is classified as lightly-loaded (idle) or heavily-loaded by comparing each node's current CPU 'busyness'-level with a 'two thresholds information policy' [3] - these thresholds must be previously established. A suitable hysteresis level must be also considered when establishing these thresholds to avoid that a processor changes from the over-loaded state to lightly-loaded state and then back again to over-loaded state too often. The node status information is propagated, from lightly-loaded to over-loaded nodes, through a message interchanging among neighbour nodes.

2.2 Gradient Model Drawbacks

Although the GM scheme is scalable, it has two serious drawbacks [4]: (1) Information from idle (lightly-loaded) nodes propagates to over-loaded nodes through intermediate ones. In the worst-case there is a distance of 'd' hops between possible source and destination processors, where d is the maximum distance between any pair of nodes in the cluster. Since the system load changes dynamically, the processor load status may be considerably out-of-date. (2) If there are only a few lightly-loaded nodes in the cluster, more than one over-loaded source processor may emit a task towards the same lightly-loaded processor. This overflow effect has the potential to transform lightly-loaded processors into heavily over-loaded ones.

2.3 Extended Gradient Algorithm

The Extended Gradient (EG) mechanism [5] overcomes the problems of out-of-date information and overflow effect. Once a processing resource is required and not locally available, the EG method interrogates the remote processor identified by the GM scheme as lightly-loaded to confirm that it is still available and reserves it to receive the new processing demand. Although the EG approach requires communication from any one to all of the other nodes in the cluster (n×n), in order to overcome the above problems, communication facilities are used economically, i.e. reservation requests are allowed only to highly-probable lightly-loaded processors (those previously identified by the GM scheme). Even with this economical use of network communication facilities, it is still possible that excessive network traffic could be generated under over-load conditions. To reduce this occasional excessive network traffic, the introduction of a bound on the number of attempts made to reserve an under-utilised node is proposed. If no lightly-loaded processor can be found after these attempts, then the new process is started locally as the overall system is clearly over-loaded and therefore the probability of existence of an available node in the cluster at that time is likely to be very small. The EG mechanism also has a global placement and a local status information exchange space. The system resource availability can be requested at arbitrary execution times in any node. A detailed description of the

EG algorithm will be further carried out (Section 5), since it is the mechanism being ported in this paper.

3 Hardware and Software Environment

The selected environment is composed of Linux/X86 compute nodes, programed in the 'C' language [6] running over the Parallel Virtual Machine (PVM) library [7]. Fully connected facilities were achieved on the PCs cluster using a Gigaswitch. Valency of two (each PC communicates with two others) was established, therefore organised as a logical ring topology to allow exchange of the EG mechanism load-balancing information.

4 Processor Availability Measurement

Decisions on the availability of the node were determined by the Linux uptime command [8]; particularly, the field that represents the medium node demand in the last minute was selected. The thresholds information policy, mentioned in Section 2.1, is then applied; by comparing the proper field of the uptime command against the thresholds, adequately set, the local node availability is then classified as lightly-loaded or heavily-loaded.

5 Porting Strategies of the EG Mechanism

A schematic representation of the dynamic load-balancing mechanism is shown in Figure 1. The EG approach is composed of three modules: gm, eg_in and eg_an. These three processes are loaded over all nodes of the PVM cluster. The gm implements the GM mechanism previously describe (Section 2.1). The eg_in module establishes the interface between the user (application), which is represented by a pvm_spawn function call of the PVM library, and the EG mechanism. Once that the eg_in module has been activated by the application, if there is no local processing resource available for the local processing demand, it interrogates the eg_an module on a remote node (indicated by the gm mechanism) to confirm that it is still available, in order to overcome the out-of-date information problem (as previously mentioned in Section 2.2). The overflow effect is also avoided by the eg_an module, i.e. an eg_an module issuing a resource commitment (an idle declaration) goes into a state of no answer; this prevents any possibility of overflow. The target process remains in this state for a pre-specified timeout, tailored to ensure that the committed transaction is completed before any further commitment. Both the GM scheme and the EG mechanism provide, at most, availability of one resource in each node at any given time. On the other hand, simultaneous user demand for resources can be generated (asynchronous events). To avoid collisions, the node select operation must be surrounded by a semaphore primitive, represented by objects in rectangular forms (see Figure 1). Two semaphores are represented: one in which the eg_in module is 'put to sleep',

if there is no demand for computational resources, and the other where applications should wait for resources. Simplified versions of these three modules in a C-like pseudo-code language are shown in Appendix A. It was also necessary to develop two other modules: one that initialises the semaphores and makes available shared memory facilities between application and the eg_in module, and another module which starts all these modules into all nodes of the PVM cluster.

6 EG Mechanism Application Programer Interface

The PVM library offers two options of process distribution through a simple procedure call to the pvm_spawn function: (1) it could be established (by the programer) where a process should be initialised, or (2) if an empty string is passed as the node name, a Cyclic Allocation (CA) load-balancing policy is used. In addition to its original features, the pvm_spawn function was custom-designed to support the EG mechanism: if a particular machine name (in the case egm), which was pre-established in a good agreement, is indicated by the user as the machine where the new process sould be started, the spawn primitive inquiries the EG policy and the process is initialised in the node pointed by the EG approach as lightly-loaded. Therefore, the facility made available by the EG mechanism is added to the pvm_spawn function. This modified spawning mechanism enables dynamic verification of processing availability in the cluster.

7 Performance Investigations of the EG Mechanism

For evaluation purposes, it is required a considerably larger number of processes (n) than nodes (m) in the cluster ($\frac{n}{m} \gg 1$), an over decomposition approach, so that the dynamic load-balancing mechanism could effectively manage the system processing capacity. The scheme used was to execute several times a same benchmark process, since it is a relatively straightforward method of spawning a large number of processes. These benchmark investigations were achieved with the number of processes value obtained as shown in Equation 1, therefore for each benchmark experiment, n processes were spawned:

$$n = k \times m^2, \tag{1}$$

where k is depending on the number of nodes. Three benchmark approaches were selected to investigate the experimental performance of the EG mechanism. Two of them, i.e. Hanoi and Queens codes, were downloaded from the 'BENCHWEB' web site [9]. The third benchmark code, also a real application, was chosen from the field of computer graphics (Ray-tracing) [10]. A detailed description of each one of these codes is outside the scope of this paper, however they are well-known algorithms. To obtain unbalanced processing demand, the spawned benchmark processes run the same code different number of times, for example, Hanoi benchmark process numbers 0, 1, 2 and 3 runs 2, 3, 4 and 2

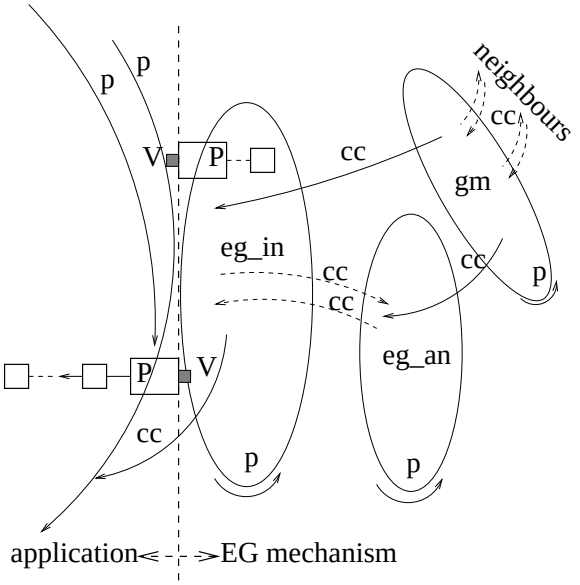

Fig. 1. Graphic representation of the ported EG mechanism
Objects marked with p represent processes: gm is the GM mechanism; eg_in and eg_an modules are extensions of the GM approach; others are application processes. Continuous lines, marked with cc, represent communication channels; in particular, objects in dotted line, marked with cc, represent communication facilities between processes in distinct nodes. Objects in rectangular forms represent the semaphore primitives. It is also represented, in dashed line, the interface between application and machine facilities (EG mechanism part of the operating system).

times the Hanoi code respectively and so on, in cycles (see Table 1). In order to start n processes in a cluster, their codes had to be adequately harnessed into the PVM environment, in a way that the EG facilities could be used. A simplified version of the 'Towers of Hanoi Puzzle' code starting mechanism, written in a C-like pseudo-code language, is shown in Appendix B. The implementations for the other codes are not presented in this paper, however their implementations are similar to the one showed. For each benchmark code, two investigations were done: (1) The execution times were measured and compared to the execution time of a CA load-balancing policy (that one from PVM - see Section 6): the percentage of improvement was then derived from these times (see Table 2). (2) The CPU-processing degradation, due to the EG mechanism, was also evaluated: sequential versions of the three benchmark processes were executed twice in one of the nodes of the cluster (making no use of the load-balancing mechanism), however, one of the benchmark executions had the EG mechanism running as a background process. The overall execution times were also compared (see Table 3).

Table 1. Unbalanced processing demand

process number (pn)	0 1 2 3 4 ...
Hanoi ($pn\%3+2$)	2 3 4 2 3 ...
Queens ($2\times(pn\%3+1)$)	2 4 6 2 4 ...
Ray-tracing ($pn\%3+2$)	2 3 4 2 3 ...

Number of times the same code runs in each process is a function of the process number. In total, Hanoi (running over sets of 16 up to 30 disks) and Ray-tracing (running over 13 scenes) applications execute the same code 216 times (3×72); Queens (15 queens on a 15×15 board) execute 288 times.

Table 2. Percentage of improvement

	execution time EG/CA (in seconds)	% of improvement
Hanoi	2034/2253	10
Queens	1911/2582	16
Ray-tracing	1798/1787	-1

Percentage of improvement derived from the total execution times ($1-\frac{EG}{CA}$). Experiments were conducted using 6 nodes and 72 processes.

8 Related Research

In particular, two related research investigations are mentioned. At first, a research investigation is presented in which the design and the implementation of another extension to the load-balancing GM mechanism were made [11]. In such implementation, to overcome the problems of out-of-date information and overflow effect of the GM scheme, a ticket propagation policy is used. These tickets flow from consumer to producer processors. The ticket policy works in such a way that an over-loaded processor is only allowed to issue a process to a consumer processor if such producer has received a ticket. Another research to be mentioned [12], contrarily to the first one, does not introduce any new balancing approach, neither a modification to an existent policy, however it claims to have made available a library that makes possible prototype evaluations of load-balancing mechanisms.

Table 3. CPU-processing degradation

	without EG	with EG
Hanoi	186	187
Queens	278	278
Ray-tracing	149	149

Numbers in second.

9 Discussions and Further Works

Table 2 shows that a substantial machine performance improvement (execution time reduction) was achieved for Hanoi (of 10%) and Queens (of 16%) real applications for the experiments whose processes average granularity ranges from values of one and half minute (under the condition described previously). The observed small performance degradation (of 1%) for Ray-tracing experiment when using the EG mechanism suggests that, in this case, the CA load-balancing policy is marginally better than the EG scheme. From the Table 3 can be inferred that CPU-processing degradation due to the EG mechanism is small than 1% for the workload model established in the experiments. Possible further works should be mentioned: (1) utilisation of prototype evaluation tools (similar to the one presented in Section 8) in order to have better evaluation of the EG mechanism; (2) implementation of additional extensions to the EG load-balancing mechanism.

10 Conclusions

It is worthwhile mentioning that the algorithm implemented: (1) is a decentralised (a locally-distributed) mechanism, therefore a scalable method; (2) has global placement space; yet, (3) can be easily required by application programer. This paper therefore contains the porting of a dynamic load-balancing mechanism on a distributed-memory MIMD machine. Experiments were conducted and results were presented to make evident that a system-level user-independent dynamic load-balancing mechanism is feasible, practical and can significantly improve performance.

11 Acknowledgements

I am grateful to the whole CENAPAD-MG/CO ('Centro Nacional de Processamento de Alto Desempenho para Minas Gerais e o Centro-Oeste') group, who made computing facilities available for the benchmark experiments and to the FAPEMIG ('Fundação de Amparo a Pesquisa do Estado de Minas Gerais') foundation for providing financial resources to support my paper presentation. I wish to mention John E. Stone for providing me the Ray-tracing code. I want also to mention the colleagues Carlos, Orozimbo and Salvador for their suggestions and comments during the development of this research.

References

1. D. G. Feitelson. Job scheduling in multiprogrammed parallel systems. Technical report, IBM T. J. Watson Research Center, August 1997. (IBM RC 19790 (87657), October 1994, Second Revision, August 1997, 175 pages).
2. F. C. H. Lin and M. R. Keller. Gradient model: a demand-driven load balancing scheme. In *IEEE Conf. on Distributed Systems*, pages 329–336, 1986.

3. K. G. Shin and Yi-Chieh Chang. Load sharing in distributed real-time systems with broadcast of state changes, 1988. TR.88-006, Int. Computer Science Institute, 1941 Center Street, Suite 600, Berkeley, CA94704, 48 pages.
4. S. Nishimura and T. L. Kunii. A decentralized dynamic scheduling scheme for transputers networks. In T. L. Kunii and D. May, editors, *Proc. of the 3^{rd} Transputer/occam Int. Conf.*, pages 181–194, Tokyo, Japan, May 1990.
5. F. J. Muniz and E. J. Zaluska. Parallel load-balancing: An extension to the gradient model. *Parallel Computing*, 21(2):287–301, February 1995.
6. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall International, 1988. 2^{nd} Edition, ANSI, INB 0-13-110362-8 (paper), 0-13-110370-9 (hard).
7. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3.0 User's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, February 1994.
8. Larry Greenfield and Michael K. Johnson. UNIX uptime comand, January 1993. fie@gauss.rutgers.edu and sonm@sunsite.unc.edu, respectively.
9. BENCHWEB - High Performance Computers, 2003. http://www.netlib.-org/benchweb/hpc.htm - aburto@nosc.mil.
10. John E. Stone. Tachyon (tm) parallel/multiprocessor ray tracing software, 2001. johns@megapixel.com; j.stone@acm.org; johns@ks.uiuc.edu.
11. Liu Feixong, Thomas Peikenkamp, and Werner Damm. An extended gradient model for NUMA multiprocessor systems. In *Algorithms, Concurrency and Knowledge*, volume 1023, pages 210–224, Thailand, December 1995. (Asian Computing Science of Lecture Notes in Computer Science, Asian Computing Science Conference, Springer).
12. K. J. Barker, N. P. Chrisochoides, and B. Holinka. Dynamic load balancing for message passing asynchronous adaptive applications. In *International Conference on Supercomputing*, Williamsburg, Va USA, 2002. (College of William and Mary, Dept. of Computer Science, 10 pages).

A Extended Gradient Mechanism

A.1 gm Module

```
while (1) {
  sleep(tick);
  local_st = determine_local_load();
  // calculate the node status
  st = node_st(local_st, l_r_st);
  // pack the status to be sent
  pvm_pkint(&st, 1, 1);
  // send status to neighbours and to
  // other processes of EG mechanism
  pvm_send(l_r_neighbour, ...);
  pvm_send(gm_eg_an, gm_eg_in, ...);
  // receive status from neighbours
  l_r_st = pvm_nrecv(l_r_neighbour, ...);
}
```

A.2 eg_in Module

```
while (1) {
  // Do a semaphore P-operation
  semop(id1, P_operations, 1);
  // verify, with gm, the idle node
  pvm_nrecv(tids_gm, gm_eg_in_type);
  pvm_upkfloat(&idle_h.load, 1, 1);
  pvm_upkint(&idle_h.tid, 1, 1);
  pvm_upkstr(idle_h.name);
  strcpy(sbuf.mtext, &idle_h.name);
  if ((idle_h.load == 0.0) &&
    (mytid != tids[2][idle_i])) {
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&mytid, 1, 1);
    pvm_send(tids[1][idle_i], eg_in_an_type);
    pvm_recv(tids[1][idle_i], eg_in_an_type);
    pvm_upkfloat(&idle_h.load, 1, 1);
    pvm_upkint(&idle_h.tid, 1, 1);
    pvm_upkstr(idle_h.name);
    if ((idle_h.load < th_high) &&
      (idle_h.how_far <= nhost/2))
      strcpy(sbuf.mtext, &idle_h.name);
  }
  // The message will be sent
  msgsnd(msqid, &sbuf, ...);
  // Do a semaphore V-operation
  semop(id2, V_operations, 1);
}
```

A.3 eg_an Module

```
while (1) {
  pvm_recv(-1, eg_in_an_type);
  pvm_upkint(&tid_inquiry, 1, 1);
  pvm_nrecv(tids_gm, gm_eg_an_type);
  pvm_upkfloat(&idle_h.load, 1, 1);
  pvm_upkint(&idle_h.tid, 1, 1);
  pvm_upkstr(idle_h.name);
  pvm_initsend(PvmDataDefault);
  pvm_pkfloat(&idle_h.load, 1, 1);
  pvm_pkint(&idle_h.tid, 1, 1);
  pvm_pkstr(idle_h.name);
  pvm_send(tid_inquiry, eg_in_an_type);
}
```

B Hanoi Benchmark Implementation

B.1 Master_Hanoi Module

```
#define nprocess ...
int main() {
  int tids[2]; // tasks ids
  master = pvm_parent();
  if (master == PvmParentNotSet)
    ntasks = nprocess;
  else {
    // receive message from the master
    pvm_recv(master, 0);
    pvm_upkint(&ntasks, 1, 1);
  }
  if (ntasks > 0) {
    // start up master Hanoi task
    pvm_spawn(m_h, (char**)0, 0, "", 1, &tids[0]);
    ntasks--;
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&ntasks, 1, 1);
    // send message to the new master
    pvm_send(tids[0], 0);
    // start up slave Hanoi task
    pvm_spawn(s_h, (char**)0, 1, "egm", 1, &tids[1]);
    // send message to the new slave
    pvm_send(tids[1], 1);
  }
}
```

B.2 Slave_Hanoi Module

```
int main() {
  pvm_recv(master, 1);
  pvm_upkint(&ntasks, 1, 1);
  loop = (ntasks % 3)+n;
  system("date >> out");
  for (i=0; i<loop; i++)
    system("hanoi >> out"); // execute hanoi code
  system("date >> out");
}
```