A TOOL FOR SPMD APPLICATION DEVELOPMENT WITH SUPPORT FOR LOAD BALANCING

A. PLASTINO
Department of Computer Science – Universidade Federal Fluminense
R. Passo da Pátria 156, Niterói 24210-240, RJ, Brazil
E-mail: plastino@dcc.ic.uff.br

C. C. RIBEIRO, N. RODRIGUEZ
Department of Computer Science – Catholic University of Rio de Janeiro
R. Marquês de São Vicente 225, Rio de Janeiro 22453-900, RJ, Brazil
E-mail: {celso,noemi}@inf.puc-rio.br

This paper describes SAMBA, a tool for supporting the development of SPMD applications. A predefined SPMD skeleton invokes application-dependent routines, which must be coded by the programmer, allowing him to concentrate on the specific problem at hand. The tool includes a library of load balancing strategies, all of which can be automatically incorporated to any SPMD application written with SAMBA. In this paper, SAMBA is used to compare several load balancing strategies for two specific applications.

1 Introduction

The SPMD (Single Program, Multiple Data) model has been widely deployed in parallel programming, due to the ease of developing and controlling a single code running on several processors (over different data sets). Moreover, data decomposition is a natural approach for the design of parallel algorithms for many problems [1]. Different SPMD applications typically present a common structure. However, parallel programmers usually write the entire application from scratch, replicating the associated development and debugging work.

The central contribution of this work is SAMBA (Single Application, Multiple Load Balancing), a tool for supporting the development of SPMD applications. The first goal of SAMBA is to allow the reuse of common code in SPMD applications. In SAMBA, a predefined SPMD skeleton invokes application-dependent routines, which must be coded by the programmer, allowing him to concentrate on the specific problem at hand.

Different applications and architectures give rise to specific unbalancing factors which must be taken care of by a *load balancing strategy.* A difficulty in the design of a new SPMD application is to find, among the variety of available strategies [2,3,4,5], the one that best fits the characteristics of the application.

Another important purpose of SAMBA is to help the programmer in identifying an appropriate load balancing algorithm for the application under development. With this purpose, the tool offers a library which contains several load balancing strategies, all of which can be automatically incorporated to any SPMD application written with SAMBA.

Other tools for supporting development of parallel applications have been proposed [6,7]. SAMBA focuses on SPMD applications and on facilitating the experimental evaluation of different load balancing strategies.

2 SAMBA

The code of an SPMD application can typically be structured in three major components: the code which is replicated for the execution of a task (In this work, we use the term *task* to identify a subset of data which corresponds to a unit of work in an SPMD application.); the load balancing strategy; and a skeleton (which initializes and terminates the application, and controls the execution of the other parts). The first of these components is specific to each application, but the other two are usually not.

We designed SAMBA (figure 1) bearing this commonality in mind. Once the programmer has written a specified set of routines – the *user module* –, SAMBA can generate different versions of the associated SPMD application for each load balancing algorithm in its library. In the user module, the programmer must define routines for generating the tasks, executing a single task, and dealing with the results.

The tool uses a predefined *control module* and offers a library of different load balancing strategies. Any one of them may be chosen to define the *load balancing module*. The control module is responsible for initializing and terminating the application, as well as for activating the other modules.

The tasks which compose the application are stored in task pools at each processor. The control module also manages these pools (currently implemented as lists) and transfers tasks between processors as required by the load balancing algorithm. The procedures which manipulate these pools have been isolated in a submodule. This allows the user to easily substitute them, if the application requires a specific data structure.

3 Programming Model

A SAMBA-generated SPMD application executes a control code which contains calls to functions defined by the user. In this section we discuss the routines which must be coded by the user. The central routine is the one for processing a single task (the kernel of the “single” program). Besides this function, the user must necessarily code the generation of tasks. Other routines,

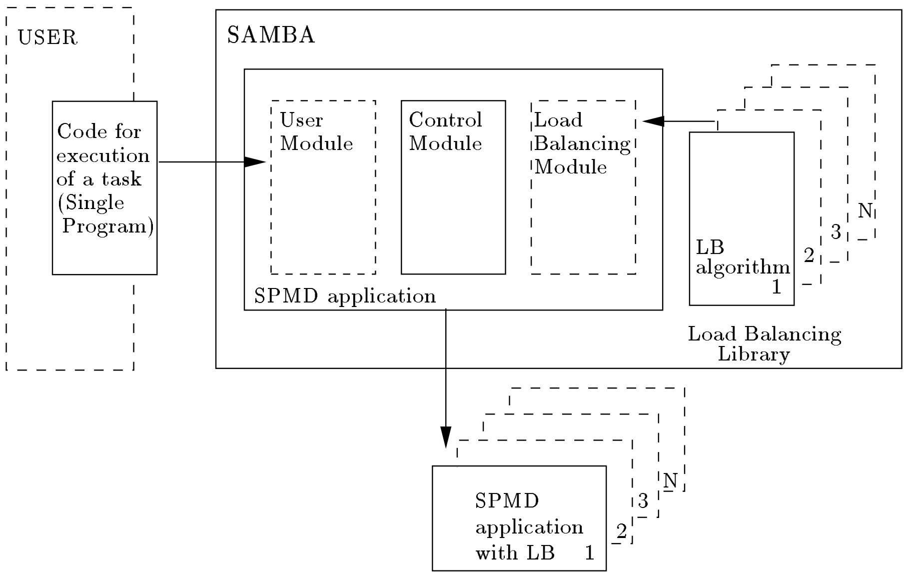

Figure 1: Architecture of SAMBA.

which may be optionally defined with empty bodies, have been specified in an effort to capture the full generality of the SPMD model.

A set of routines is reserved for initialization chores. One of these must take care of task creation. Each new task must be registered by calling a routine in the control module. Before calling this routine, the task description must be packed into a contiguous block of data (The control module knows nothing about the semantics of such description.). Besides the creation of tasks, other common initialization chores include initialization of global variables, parameter verification, and data input and argument checking.

The task execution routine will be continuously invoked at each processor during the application execution. When invoked, it receives the task to be executed as a block of bytes. It unpacks this block to obtain a task description, executes the task, and finally stores results in global variables or sends them to other processors. In some cases, such as in search tree or recursive applications, this function may also register new tasks generated by the current one.

Last, there are routines reserved for finalization procedures, which are invoked after all tasks have been executed. Such procedures may include sending results to a central processor or storing them in files.

4 Load Balancing in SAMBA

One of the assumptions behind the development of SAMBA was that it is possible to use the architecture and programming model described in sections 2 and 3 independently of the chosen load balancing strategy. To validate the hypothesis, we implemented load balancing algorithms with a wide variety of characteristics. The development of SAMBA's load balancing library was guided by a taxonomy of load balancing algorithms for SPMD applications which we had previously proposed [8]. This section describes seven implemented algorithms. SAMBA also allows new algorithms to be included in its library.

1 STATIC: The central processor distributes the initial set of tasks in equal parts among all processors, including itself. Each processor simply executes the tasks it has received, without any dynamic load balancing.

2 DEMAND-DRIVEN: A central processor first allocates a block of tasks to each other processor. Each time a processor finishes executing a block of tasks, it requests and receives a new one.

3 DISTRIBUTED, GLOBAL: A central processor distributes the complete set of initial tasks in equal parts among all processors, including itself. Whenever any processor consumes its tasks, it sends a message to all others, asking them to execute a load balancing step. At this point, each processor sends to all others its internal load index, i.e., the remaining number of tasks to be executed. Load balancing takes place only if the load index of at least one processor exceeds a certain threshold, which is given as an argument in the activation of the application. In this case, each processor computes the necessary task exchanges and performs the exchanges in which it is involved.

4 CENTRALIZED, GLOBAL: This algorithm is similar to the previous one, except that all decisions are taken by a central processor. Whenever any processor consumes all its tasks, it sends a message to all other processors asking them to send their internal load index (number of remaining tasks) to a central processor. After receiving this information from all processors, this central processor checks the need for load balancing, in the same way as before. If load is to be balanced, the central processor defines the necessary exchanges, and informs each processor of the task transfers in which it is involved.

5 DISTRIBUTED, GLOBAL, INDIVIDUAL: Once again, load balancing is triggered by the end of tasks at a processor. However, instead of aiming at balancing the load of all processors, here the goal is to correct the load problem at this single underloaded processor. Whenever a processor consumes all its tasks, it sends a message to all others and they send back their internal load indexes. Based on the received information, the underloaded processor will choose one

from which it will request new tasks. After receiving load information from all processors, the underloaded processor checks whether load balancing is to take place or not, in the same way as in algorithm 3. If load balancing is necessary, a request for load transfer is sent to the most loaded processor.

6 DISTRIBUTED, LOCAL, PARTITION-BASED: This algorithm is basically the same as algorithm 3, except that processors are partitioned in disjoint groups. Load balancing takes place inside groups defined by the user.

7 DISTRIBUTED, LOCAL, NEIGHBORHOOD-BASED, INDIVIDUAL: This strategy is similar to algorithm 5, except that load information and load exchanges occur only between processors in the same group. Groups are defined by a logical neighborhood (which could reflect physical connections) and processors may belong to more than one group. This strategy requires the use of a termination detection algorithm, since there may still be global load to execute even when neighboring processors have no remaining load. Dijkstra's termination detection algorithm [9] was used in our implementation.

5 Experimental Results

The current version of SAMBA was written using C and MPI [10]. It has been executed on an IBM SP2, on a network of Sun workstations, and on a cluster of IBM PCs. As an initial evaluation of the tool, we have used it to develop and analyse load balancing strategies for well known and understood SPMD applications. In this section we report the results obtained with two applications: computation of $\int_0^{15} e^x dx$ by the adaptive quadrature method with a tolerance of 10^{-16} (application #1) and computation of the product of two integer valued square matrices of dimension 2000 (application #2).

The choice of these two applications was due to their simplicity, which allowed us to focus on load balancing issues, and to their different task creation characteristics. While in application #1 tasks are generated dynamically (and their total number is previously unknown) by the execution of other tasks, in application #2 the number of tasks is fixed from the beginning.

The seven strategies described in the previous section have been tested and compared. For the last strategy, two logical topologies have been evaluated: ring (R) and hypercube (H). Each group of the local partition-besed strategy is formed by exactly four processors. The computational experiments we present have been carried out on a cluster of 32 Pentium II-400 running Linux.

Application #1 computes $\int_0^{15} e^x dx$ using the adaptive quadrature method. Numeric quadrature computes the definite integral of a function over an interval by dividing it into subintervals and approximating the area of each subinterval using a method like the trapezoidal rule. The creation of subintervals is dynamically determined. The approximation of the area in an interval is compared to the sum of the two areas corresponding to the two subintervals obtained by dividing this interval at the middle. If the difference is small enough (below the tolerance limit), the approximation is returned. Otherwise, recursive (and parallel) computations proceed over the two subintervals.

In our experiment, the original interval was initially divided into 64 subintervals, generating 64 tasks. The computational effort of these 64 tasks is not the same, since there are different levels of recursion for each subinterval. In the static run with four processors, each processor received 16 tasks. The total number of executed tasks was 110 933 464. However, the number of tasks executed by each of them was quite different, giving rise to the need for dynamic load balancing: 1 793 764, 6 281 112, 22 816 968, and 80 041 620 tasks. Table 1 shows elapsed times in seconds for different load balancing strategies, using 4, 8, 16, and 32 processors. The experiment was performed in exclusive mode (i.e., with no other applications sharing the cluster) - the only unbalancing factor was the dynamic generation of tasks. The demand-driven strategy (lb2) led to the best results, independently of the number of processors. The poor behavior of strategies based on task migration (lb3 to 7) seems to be natural for small granularity applications with a large number of dynamically generated tasks. To illustrate this point, we notice that the load balancing was activated 548 times along the distributed strategy (lb3) using four processors. Also, the very bad behavior of the partition-based strategy (lb6) is due to the fact that different processor groups receive initial subsets of tasks with different computational demands, but are not able to redistribute them to other groups.

Table 1: Application #1 using 4, 8, 16, and 32 processors.

Proc.	lb1	lb2	lb3	lb4	lb5	lb6	lb7R	lb7H
4	248	114	126	132	115	126	117	117
8	162	49	82	82	74	122	91	81
16	103	23	49	51	56	88	68	58
32	52	13	32	32	39	53	49	36

Application #2 computes the product of two integer valued square matrices of dimension 2000. The basic parallel multiplication algorithm is used. The second matrix is replicated on all processors. Each task is defined by a row in the first matrix and corresponds to computing one of the 2000 rows of the resulting matrix. In this case, all tasks present roughly the same computational demand, and the only unbalancing factor is the external load.

Table 2 presents execution times for application #2 with different load balancing strategies, using 4, 8, 16, and 32 processors. External load was simulated by load processes which continuously computed arithmetic expressions. In each group of four processors, a first processor executed no load process, the second executed one process, the third executed two processes, and the fourth, three processes. The static case (lb1) reflects the external load - with four processors, each of them executing 500 tasks, execution times at each processor were 170, 412, 618, and 762 seconds.

Table 2: Application #2 using 4, 8, 16, and 32 processors with external load.

Proc.	lb1	lb2	lb3	lb4	lb5	lb6	lb7R	lb7H	lb8	lb9
4	762	496	391	383	379	391	386	386	374	374
8	410	261	247	252	244	252	245	260	241	248
16	265	168	189	190	196	177	191	195	187	171
32	182	128	171	163	175	145	153	166	166	140

With four and eight processors, the algorithms based on task migration (lb3 to 7) presented comparable results, and were more appropriate than the demand-driven algorithm (lb2). This possibly occurs due to the fact that, with few processors, the communication overhead imposed by the exchange of load indexes is low. On the other hand, in the demand-driven strategy, one processor is dedicated to task distribution and does not contribute to the computation itself. With few processors, this represents a heavy loss in computing power.

As the number of processors grows, the demand-driven strategy again achieves the best results. However, it is interesting to note the relatively good performance of the partition-based strategy (lb6). In this experiment, the same load unbalance was generated inside each group. In this setting, a group strategy can attain the same balancing results as a global one and implies less communication overhead. The comparison of strategies 3 and 6 thus reflects the communication overhead due to a totally distributed computation.

We modified algorithms 3 and 6 to take the external load into account, originating strategies 8 and 9, respectively. The external load index used is the number of tasks executed per unit of time, which reflects the computational capacity of each processor. The gain in performance obtained with the new algorithms is still very small. However, we believe this gain can be higher if we eliminate the observed “ping–pong” effect. When a processor is handling little external load, there is a tendency for too many processors to send it their load, causing the previously underloaded processor to become a bottleneck in the system, and eventually to have to send back part of the load it received.

6 Final Remarks

We presented a tool for supporting the development of SPMD applications and the evaluation of different load balancing estrategies. SAMBA's implementation has shown that it is possible to integrate different load balancing strategies to the single code of an SPMD application. Once the user has written the set of routines described in section 3, several different versions of his SPMD application can be generated with no further programming effort.

The analyses reported in section 5 show how the tool can help in identifying the appropriate strategies for specific applications and environments. The computational results also show significant reduction in elapsed times due to the use of appropriate load balancing algorithms.

References

1. T. G. Mattson, "Scientific computation", *Parallel and Distributed Computing Handbook* (A. Y. Zomaya, editor), 981–1002, McGraw-Hill, 1996.
2. M. A. Franklin and V. Govindan, "A general matrix iterative model for dynamic load balancing", *Parallel Computing* 22 (1996), 969–989.
3. S. Lifschitz, A. Plastino, and C. C. Ribeiro, "Exploring load balancing in parallel processing of recursive queries", *Proceedings of the III EUROPAR*, LNCS 1300, 1997, 1125–1129.
4. M. A. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers", *IEEE Trans. on Parallel and Dist. Systems* 4 (1993), 979–993.
5. M. J. Zaki, W. Li, and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations", *Journal of Parallel and Distributed Computing* 43 (1997), 156–162.
6. T. Decker, "Virtual Data Space – a universal load balancing scheme", *Proceedings of the IRREGULAR IV*, LNCS 1253, 1997, 159–166.
7. C. Fonlupt , P. Marquet, and J. Dekeyser, "Data-parallel load balancing strategies", *Parallel Computing* 24 (1998), 1665–1684.
8. A. Plastino, C.C. Ribeiro, and N. Rodriguez "Uma taxonomia de algoritmos de balanceamento de carga para aplicações SPMD", Technical Report PUC-RioInf.MCC49/98, Department of Computer Science, Catholic University of Rio de Janeiro, Brazil, 1998.
9. E. Dijkstra, W. Seijen, and A. Gasteren, "Derivation of a termination detection algorithm for a distributed computation", *Information Processing Letters* 16 (1983), 217–219.
10. Message Passing Interface Forum, "MPI: A message passing interface", *Proc. Supercomputing'93*, 878–883, IEEE Computer Society, 1993.