Supporting Self-Adaptivity for SPMD Message-Passing Applications*

M. Cermele, M. Colajanni, S. Tucci

Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Italy 00133
{cermele, colajanni, tucci}@uniroma2.it

Abstract. Real parallel applications find little benefits from code portability that does not guarantee acceptable efficiency. In this paper, we describe the new features of a framework that allows the development of Single Program Multiple Data (SPMD) applications adaptable to different distributed-memory machines, varying from traditional parallel computers to networks of workstations. Special programming primitives providing indirect accesses to the platform and data domain guarantee code portability and open the way to runtime optimizations carried out by a scheduler and a runtime support.

1 Introduction

The SPMD paradigm is the most widely adopted model for a large class of problems. However, this programming paradigm does not facilitate portability because it requires the choice of a specific domain decomposition, and the insertion of communications and parallel primitives in a decomposition dependent way. If we want a parallel application to be portable with efficiency (*performance portability*), the best domain decomposition and communication pattern cannot be chosen during implementation. Although standard message-passing libraries, such as PVM and MPI, guarantee *code portability*, performance portability requires more sophisticated frameworks. They have to provide an abstract machine for the parallel implementation and a runtime environment for adapting the application to the status of the actual platform. The solution can be at different levels of abstraction going from explicit tools to transparent supports. The former approach leads to parallel programs that we call *reconfigurable* because the adaptivity is achieved at the price of some programming overhead. The latter solution makes a program *self-adaptable* to various platforms without any programming effort greater than the parallelization itself.

This paper proposes a method to add self-adaptivity to SPMD message-passing applications running on multicomputers, clusters or distributed systems, where each (possibly, heterogeneous and nondedicated) node has its own local

* © Springer-Verlag. Proc. of *ACM Workshop on Languages, Compilers and Runtime Systems* (LCR'98), Pittsburgh, PA, May 1998. To appear in Lecture Notes in Computer Science, 1998.

memory. The focus is on the new features of the Dame framework that was initially proposed for cluster computing to dynamically switch from one type of data-distribution to another type [9]. With respect to that version, now Dame includes a new *abstract machine interface* and a new *scheduler* that improve transparency and enlarge the class of applications and target machines. This paper describes the role of those components in achieving self-adaptivity, while does not present other features under implementation, such as the capability of dynamically changing the number of processes that actually carry out computation, the new algorithms for mapping the application's communication pattern onto the active nodes, the automatic method to choose the frequency of activation of the reconfiguration support.

Section 2 describes the basic approach, the programming paradigm and the abstract machine interface. Section 3 outlines how the scheduler adapts the application to the platform status at startup. Section 4 details the operations of the runtime scheduling. Section 5 discusses related works. The paper is concluded by some final remarks.

2 Abstract Machine Interface

The simplicity of the SPMD model is preserved if the programmer can assume a regular and homogeneous environment. The question whether a support can provide easy-to-use parallelism, facilitate the portability and not sacrifice performance requires a decision about the most appropriate level of abstraction. The proposed levels go from an abstract machine that hides only system and network heterogeneity such as in PVM and MPI, to an explicit use of facilities for load balancing such as in ADM [4] and DRMS [17], until the more abstract Linda machine that hides distributed memory [3]. The solution suggested in this paper is an intermediate degree of abstraction that guarantees code portability without sacrificing performance portability. The Dame abstract machine emulates the main features of a distributed memory parallel computer. It hides system heterogeneity, node nonuniformity, actual (typically, irregular) domain decomposition and the underlying architecture, however it still requires the explicit insertion of communication primitives. The programmer or the supercompiler can refer to a *virtual platform* and a *virtual data-distribution*. The virtual platform consists of homogeneous nodes with the same static computational power. Each node executes one SPMD thread that has its own address space. The need for any data placed in the memory of other nodes requires explicit message-passing. The virtual data-distribution is regular because each node is supposed to have the same amount of data items. The programmer can choose the virtual data-distribution which is most appropriate to the application without caring of the platform. Presently, Dame manages one- and two-dimensional decompositions with block, cyclic, or intermediate decomposition formats [5].

The complexity of portable parallel computing is effectively hidden in the *Dame Library* (DML). The DML primitives are simpler than standard message-passing constructs because they refer to the virtual platform and virtual data-

distribution. These primitives, providing *indirect accesses* to the platform and data domain, replace any SPMD operation (e.g., system inquiry, domain inquiry, communication) that typically depends on the platform and/or data-distribution. This choice guarantees code portability and allows the dynamic reconfigurations that are at the basis of performance portability. Indirect accesses to data items were previously proposed in CHAOS [19] and Multiblock Parti libraries [1] to manage irregular parallel applications. Here we outline the primitives that are relevant to dealing with data parallelism without taking care of the distributed platform and related issues.

System inquiries. System inquiries give information about the number of nodes involved in computation. Moreover, these primitives are crucial for establishing the role of each node when it executes an SPMD program. The library provides the programmer with primitives for node identification which are based on the subset of data domain they own. A process is not referred in the code through explicit Cartesian coordinates or static identifier such as `pid`, but always through a formula like 'node(s) holding certain data items', where this information is given by the *data inquiry* primitives.

Data inquiries. These are the key primitives that support the decomposition independence of Dame applications since the programmer is never required to specify exactly data locations and bounds of local domain. The *data inquiry* primitives have typically three arguments: <*domain*>, <*row index*>, <*column index*>, where the indexes can refer to the local or global domain or can be the wildcard -1. For example, `dml_f(A,i,j)` applies the primitive f to the local or global entry (i, j) of the matrix A, while `dml_f(A,k,-1)` refers to all the row k. Let us distinguish the data inquiry primitives into five classes.

- *Owner identifiers* return the node or set of nodes that hold a specified data set. The former primitive `dml_which_node(A,i,j)` returns the node identifier to which the global entry (i, j) is assigned. The wildcard parameter is not allowed for this function.
  The latter primitive `dml_which_set(A,i1,i2,j1,j2)` returns the set of node identifiers that contain the subdomain delimited by the rows $i1 - i2$ and by the columns $j1 - j2$. The wildcard parameter -1 can be used in any of the last four input positions. Depending on the position, the meaning is either from the beginning or until the end. For example, `dml_which_set(A,i1,-1,j1,-1)` denotes all nodes containing the submatrix starting from row $i1$ and column $j1$. Moreover, `dml_which_set(A,i,i,-1,-1)` denotes all nodes containing row i.
- *Local data extractors* get from the global data domain the part contained in the address space of the calling node or set of nodes.
- *Index conversion* functions consist of three primitives: *local-to-global*, *global-to-local*, *local loop ranges*.
  Local-to-global primitives take as input a local row (column) index and return the equivalent global row (column) index.
  Dual results are given by the *global-to-local* conversion functions. For

example, `dml_gtol(A,i,-1)` translates the global row index i into the equivalent global row index within the address space of the calling node. Local-to-global and global-to-local primitives return a negative value (that is, error) if the input index is out of the local domain range.
The index conversion class contains also *proximity* primitives that are mainly used to iterate over the local data domain. They compute lower and upper bounds of index loops by taking as input an array and a row (column) global index and returning the closest local index. For example, `dml_close(A,i,-1)` translates the global row index i into the local row index h which is closest to i. Let us consider that the global identifiers of the first and last rows held by the calling node are $i1$ and $i2$, respectively. Three cases are in order. If $i < i1$, the primitive returns $h = 0$; if $i > i2$, it returns the local index of the last row located in the calling node; otherwise, it behaves analogously to `dml_gtol(A,i,-1)`.

- *Owned domain bounds* take the global domain name as input parameter and return the local or global index of the first and last row (column) of the subdomain contained in the address space of the calling node. The primitives returning the global indexes are `dml_gminX(A)`, `dml_gmaxX(A)`, `dml_gminY(A)`, `dml_gmaxY(A)`, while the local indexes are obtained by analogous primitives such as `dml_lminX(A)`.
- *Locality predicates* take as input a global index and execute locality tests. For example, `dml_in_mynode(A,i,-1)` returns true if a part of row i is located in the address space of the calling node.

Communications. Dame communications conform to the PVM/MPI standard by supporting three classes of communications: point-to-point, multicast, gather. All these primitives are based on the send-receive mechanism provided by PVM or MPI.
The use is slightly different and in some sense simplified. Evaluation of the destination nodes and message transmission has to be seen as an atomic action in the sense that no reconfiguration primitive can occur between these two operations. The destination nodes must be always evaluated before the communication through the owner identifier primitives.
A point-to-point communication of a data d to the node owning the (i, j) entry is done through the couple `dest=dml_which_node(A,i,j)` and `dml_send(tag,d,dest)`. A multicast to the nodes having a portion of column h is done through `mdest=dml_which_set(A,-1,-1,h,h)` and `dml_mcast(tag,d,mdest)`, where `mdest` is an integer array containing the process identifiers to be sent to (the number of these processes are specified in the first position of the array). Keeping message transmission and destination evaluation as an atomic action allows to compute at runtime the actual scope of each communication.

Interface primitives. They represent the only non-transparent interface between the abstract machine and the support that guarantees adaptivity. The `dml_init` and `dml_end` primitives initialize and close the Dame environment,

while `dml_check_load` activates the runtime scheduler. This last function will be discussed in Section 4.

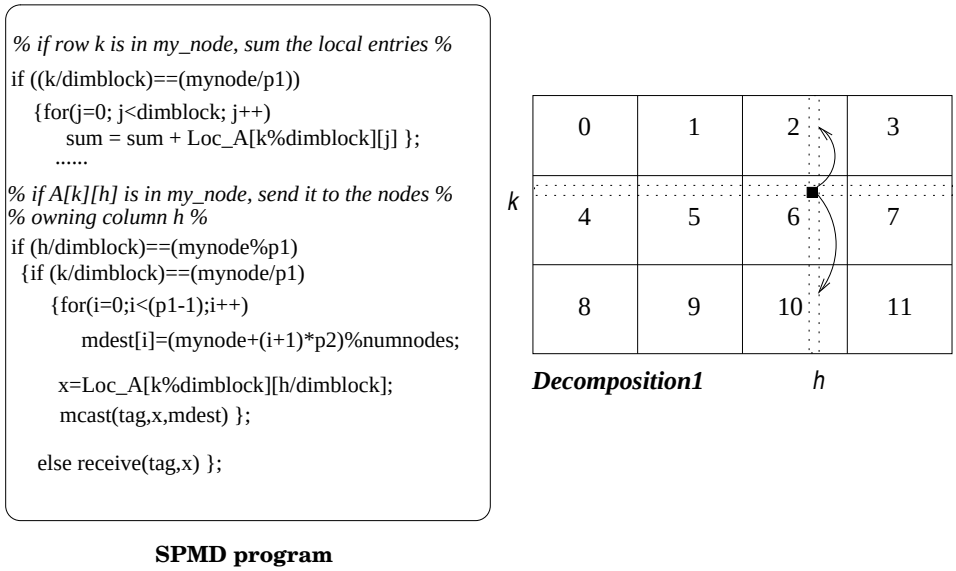

Fig. 1. An example of SPMD program.

There is no complexity of converting legacy MPI/PVM applications to the Dame library because self-adaptivity simplifies SPMD programming. Let us consider the example in Figure 1. It contains a naive SPMD code where the nodes owning a portion of row k evaluate the sum of their local entries, and the node having the $A(k,h)$ entry sends it to each node containing elements of column h. Note that tedious and error-prone operations, such as the evaluation of the iterative loop bounds, conditional checks and communications, are replaced by simple calls to DML primitives. Moreover, if we want to execute the program of Figure 1 on an irregular platform where the nodes have different speeds, the SPMD program does not work well because it strictly refers to a block decomposed matrix which is uniformly mapped onto a $p_1 \times p_2$ mesh.

On the other hand, the Dame version of the same algorithm (Figure 2) is independent of the actual platform and data-distribution. The same program can be executed without requiring changes and recompilations. If necessary, Dame scheduler achieves the nonuniform data-distribution shown in *Decomposition2*, and the same Dame code is able to self-adapt to this balanced decomposition. Node behavior is different depending on data-distributions. As example, for *Decomposition1* the $A(k,h)$ entry is sent by node 6 to nodes 2 and 10, for *Decomposition2*, the same entry is sent by node 4 to nodes 1 and 11.

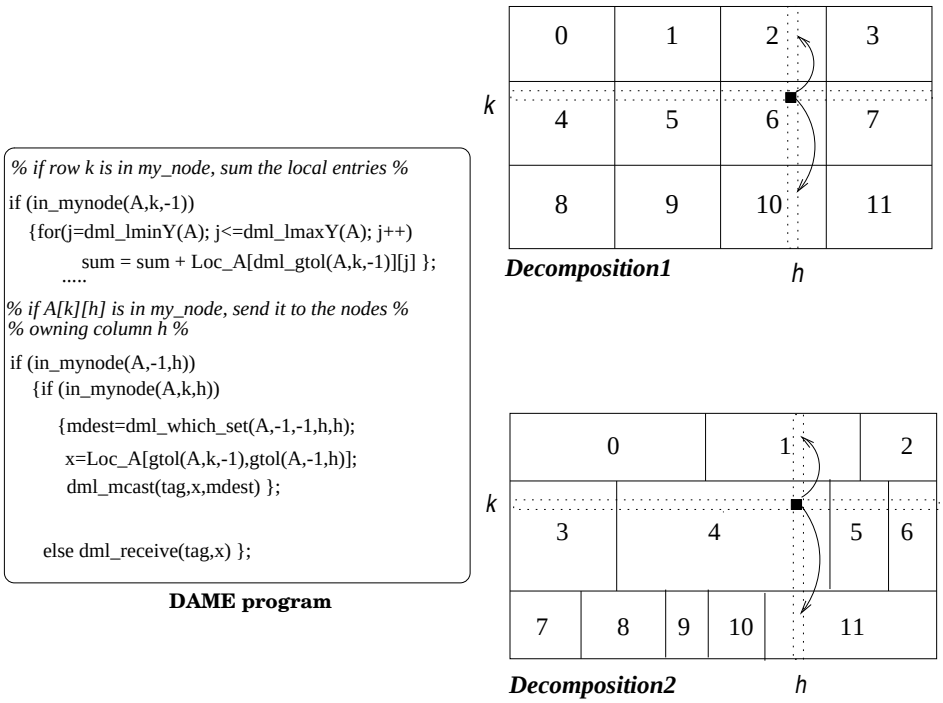

Fig. 2. The same program of Figure 1 written in Dame.

Dame achieves self-adaptivity by intervening in all phases of program's life. We have already seen the peculiarities of the Dame implementation style. The compilation does not alter the program's independence, because it generates a code that leaves unset all indirect accesses to the platform and data domain. The runtime support translates the Dame primitives referring to the abstract parallel machine into explicit accesses to actual data locations and active nodes. Modifications of the active platform and data-distribution do not require any adjustment of the high level code because the semantic flexibility of the indirect accesses allows the library runtime support to take different actions at different time for the same primitive call. When the scheduler modifies the actual platform and/or data-distribution, the application is indirectly informed through the updated results of the Dame primitives.

An interesting property is that the semantic flexibility does not introduce big overhead to the underlying layers. Table 1 shows that the cost of the decomposition-independent Dame primitives is equivalent to the cost of analogous constructs that are explicitly implemented in PVM and MPI on the basis of a single data-distribution.

The slightly higher time of some Dame primitives is entirely rewarded by the simplicity of implementation and self-adaptivity of the code. Moreover, that

overhead becomes quite negligible when the primitives are used in a real contest. To this purpose, we compare the execution time of a *template* implementing a portion of the LU factorization algorithm (identification of the nodes owning the pivot column and evaluation of the multipliers). The transmission times are quite comparable because the Dame communications are based on PVM or MPI layers. The cost due to the dynamic evaluation of the destinations is much lower than any message transmission and does not affect the overall performance.

Table 1. Execution time (μsec., as average of 100 runs) of some programming primitives.

Class	**Primitive**	PVM	Dame-PVM	MPI	Dame-MPI
System inquiry	`dml_my_node`	< 0.001	< 0.001	< 0.001	< 0.001
	`dml_which_set(`A`,`k`,-1)`	0.23	0.48	0.37	0.59
Data inquiry	`dml_lmaxX(A)/dml_gmaxY(A)`	0.01	0.01	0.01	0.01
	`dml_gtol(`A`,-1,`k`)`	0.18	0.23	0.22	0.28
	`if(dml_in_mynode(`A`,`k`,-1))`	0.8	0.9	0.8	0.9
	template	33	34	33	34
Communication	`dml_send` (single data)	$1.3E+3$	$1.3E+3$	$1.1E+3$	$1.1E+3$

3 Initial adaptivity

The *initial adaptivity* is guaranteed at startup time by the scheduler through three main activities:

- choice of the *active platform* that is, the set of nodes that actually carry out computation at a certain time;
- mapping of the communication pattern of the application onto the topology of the active platform;
- mapping of the virtual data-distribution onto the nodes of the active platform.

3.1 Active platform

The user chooses the maximum set of machines that participate in computation at startup time. This set, called *maximum platform*, may differ from the active platform. For example, a node of the platform could be excluded because it does not provide adequate efficiency or because an external application with higher priority requires exclusive use of this node. If the platform is static that is, we are guaranteed that all the initial nodes provide the same computational power, the active platform does not change at runtime. Otherwise, if the real platform is dynamic, we could have various active platforms during computation.

The algorithm that chooses the initial active platform works on the basis of three kinds of startup information. Some of them are provided by the user at startup (U), others are typically set once and for all (O), others are automatically get by the scheduler (A).

Maximum platform (U).
This is the maximum set of machines that can participate in the computation. Typically, there is a list of default machines from which the user can select the maximum set for that computation. The set chosen at startup does not vary at runtime. The Dame scheduler, differently from the Piranha scheduler [3], does not try to dynamically acquire machines external to the maximum set. Hence, the active platform is always a subset of the maximum platform.

Startup platform conditions (A).
The scheduler evaluates the status of each node of the maximum platform such as availability, computational power and, if the nodes belong to a platform distributed among multiple subnetworks, mean inter-communication times.

Exclusion thresholds (O).
There are three set of thresholds that are useful when the actual platform is not a traditional parallel machine: *power threshold* Π, *latency threshold* Λ, *sharing thresholds* ϑ_i^D and ϑ_i^N.
In a heterogeneous platform, the nodes may have different computational powers. The scheduler excludes from the initial active platform the nodes that offer a relative power so small that the ratio between that value and the average platform speed is below Π.
In a platform where the nodes have different inter-communication times, the threshold Λ is used to exclude from the initial active platform the nodes that are connected so slowly to cause a bottleneck for the application.
When the platform is used as a parallel compute server and by individual users, the scheduler requires for each node i a couple of *sharing threshold* ϑ_i^D and ϑ_i^N that is, the maximum percentage of power that the parallel application is allowed to employ on node i during day-time and night-time, respectively. When an external application requires more power than that specified by the sharing threshold value, the scheduler withdrawals the parallel application from that node. Typically, each node has its default ϑ_i^D and ϑ_i^N values that can be changed only upon explicit request at startup. These values are negotiated when the parallel user asks for accounts on non-property machines. The use of shared thresholds provides a twofold contribution. They reduce suspicious look at a shared use of resources, and allow a flexible degree of obtrusiveness of the Dame applications. For example, if the shared threshold for each node is close to zero, Dame behaves similarly to frameworks such as Condor [16] and Piranha [3] that leave a node as soon as an interactive user logs in.

3.2 Mapping algorithms

Once defined the active platform, central to the scheduler role are the notions of *embedding* and *data mapping*. The scheduler binds the application's communication pattern to the active platform topology, and the virtual data-distribution to the data-distribution that best adapt itself to the status of each node.

The problem of embedding the communication pattern onto the active platform is highly simplified thanks to the assumption that the number of nodes of the active platform coincides with the number of active application threads. Analogously, the data mapping is simplified because the computational domain has a fine granularity. Hence, data entries can be almost optimally assigned with no computational effort. Embedding and data mapping require the following user's information.

1. *Virtual data-distribution.*
   The programmer has implemented the SPMD application on the basis of this distribution. Two parameters characterize the division that has to be made on the computational domain: the *decomposition dimension* and the *decomposition format*. Presently, Dame manages one-dimensional (by row or column) and two-dimensional decompositions, and pure block, cyclic, or intermediate combinations of decomposition formats. Other frameworks that provide nonuniform data-distributions for cluster-based computing consider only static decompositions [8], block decompositions [12] or are not transparent to the user [4].
   One-dimensional decompositions typically lead to applications with lower number of communications. The advantage of two-dimensional decompositions is a smaller granularity of the items that can be reconfigured and, hence, a potential better load balancing. This choice is especially recommended for irregular applications. It should be noted that a parallel program implemented on the basis of a two-dimensional decomposition can run even for one-dimensional decompositions, whereas the opposite does not hold.
2. *Actual topology.*
   Various topologies are proposed in literature. However, the topologies of existing platforms are rather restricted. Presently, the scheduler manages most common topologies such as *hypercube* (e.g., Ncube2), *two-dimensional mesh* (e.g., Paragon), *single-line cluster* (e.g., nodes in the same physical network connected through one line such as Ethernet), *multiple-lines cluster* (e.g., IBM SP-2, or a Myrinet cluster where multiple lines connect the nodes), *multi-cluster* (e.g., a distributed system where the nodes belong to different physical subnetworks).
3. *Communication pattern.*
   Each SPMD message-passing application follows a prevalent pattern to exchange information among nodes. For the embedding task, the scheduler takes into account common patterns: *master/slave*, *near-neighbor*, *multicast*, *irregular*.

4 Dynamic adaptivity

The dynamic adaptivity of the SPMD application is guaranteed by a *runtime environment* that consists of a dynamic activation of the scheduler to manage load and platform reconfigurations, and a runtime support that adapts the effects of Dame primitives in a transparent way, thereby masking any modification to the high-level code. The runtime scheduler aims to solve dynamic load imbalance due to the platform and/or to the application.

In particular, at runtime the scheduler copes with external factors such as active set modifications and relative node power variations when the platform is shared among multiple users. The internal factors are due to irregular applications. (As example, we have investigated the WaTor algorithm [5] and large scale molecular dynamics.) The overall process of platform and workload reconfiguration is hidden to the programmer. Moreover, the new version of the runtime scheduler uses a decision support system that automatically chooses the frequency of activation on the basis of information monitored at runtime [6].

The runtime scheduler maintains the workload proportional to the computational power of each node through migrations of data items from overloaded to underloaded nodes. Although the decision is centralized, a possible remapping is not performed by the master node that only indicates to each process which data are to be sent or received. The SPMD paradigm allows a very light-weight reconfiguration: only the local data of a process need to be moved during the migration phase. At the end of this phase, the data domain is partitioned in proportion to the power of each node.

We designed the support in a modular way so that different load balancing strategies can be easily plugged in. The dynamic load balancing model is similar to that given in [21]. It consists of four phases that can be implemented in different ways:

1. *Activation mechanism.*
   Various protocols such as synchronous/asynchronous and implicit/explicit have different degrees of interference between the support and internal processes. We propose a partial transparent framework that does not leave any responsibility for data reconfiguration to the programmer, however it requires the programmer to specify the place of the application where the support has to be activated through the `dml_check_load()` primitive. The activation interval can be chosen by the programmer or automatically set by the support [6].
2. *Load monitoring.*
   Once the support has been activated, each process evaluates the status of the external workload on its node. This phase is usually executed in a distributed way. We use two active methods for load monitoring. The alternatives regard the ways in which the load parameter can be evaluated. The first method exploits Unix system information. One can obtain different kinds of information about current computational power such as average number of tasks in the run queue, one-minute load average, rate of CPU context switches.

Following Kunz's results [15], we use the number of tasks in the run queue as basis for measuring the external workload.
An available alternative estimates the current load through a synthetic micro-benchmark that is, a purposely implemented function which gives an immediate estimate about the available capacity. The code of the micro-benchmark is specifically designed for scientific-based programs. Other classes of parallel applications would require different micro-benchmarks. The Unix call is faster in the evaluation of the load but requires some additional computations to evaluate the available capacity. Moreover, this estimate causes some approximation in the load information.

3. *Decision.*
This phase takes two important decisions: *whether* to redistribute and *how* to redistribute. We implemented policies which are based on a centralized approach. The reconfiguration master is responsible for collecting the load parameters, executing one decision policy, and broadcasting the results to the internal processes. This message consists of three parts: *operation* (to reconfigure or not), *node information* (map of sender and receiver nodes), *data information* (map of data to transmit). Even if a centralized approach tends to be more time consuming and less feasible than distributed strategies as the number of processors in the system becomes large, we preferred a central scheme such as in [18, 2, 20] because it guarantees the consistency of a generic SPMD application, and allows the master to keep track of the global load situation to evaluate the relative capacities. Various decision policies can be used in Dame. They use different state information, such as instantaneous workload variation, present and past load, maximum system imbalance.
4. *Data reconfiguration.*
When the runtime scheduler decides a data reconfiguration, each application process has to be blocked. Data migrations occur in a distributed way among the nodes that own parts of neighbor data domain. In this way, the scheduler ensures the locality of data-distribution after the reconfiguration. The scheduler is in charge of the flow control of messages for data migrations. If a one-dimensional data distribution is used, each node communicates with the two neighbor nodes. For two-dimensional data distributions, data exchanges occur first horizontally and then, if necessary, vertically.

In our current implementation, the framework follows a master/slave policy. It is activated explicitly by a call to the `dml_check_load()` function that we provide together with the run-time support. When this function is activated, each process independently evaluates the load status of its node, and transmits the new parameters to the reconfiguration master process.

Although an explicit activation of the runtime scheduler partially violates the transparency requirement that frameworks such as Dame should satisfy, the programmer's task is very limited. The check-load points are specified by the programmer if the application is parallelized by hand, or may be automatically chosen by the compiler if the application is automatically parallelized by

frameworks such as HPF. However, even in this latter case, an explicit intervention of the user through some directive seems the most efficient choice because the admissible check-load points risk to be too many and the insertion of a `dml_check_load` call in each of them is impractical because of excessive overhead. Additional considerations for the automatic insertion of check-load points are discussed in [12].

5 Related Work

Dame provides a new *abstract machine interface* consisting of a programming model, a language to use it and a runtime support. From the portability point of view, it is not important if the parallelization is done by hand or automatically because the main goal is that the programmer or the supercompiler refers the implementation to the abstract machine. The code is adapted to the real platform by the *scheduler* that monitors the platform status and workload conditions. As the runtime environment does not force any adjustment on the high level code and does not require programmer's intervention, the Dame program can be considered *self-adaptable*. This property differentiates this version of Dame from other projects oriented to SPMD message-passing applications [17, 9, 12].

The focus on the SPMD model is the main difference among Dame solutions and the approaches of other projects that aim at a self-adaptable parallel code, but in the context of task migration [4, 11] and thread migration [10] solutions, object-oriented parallelism [2, 13], special languages such as Linda [3] and Dataparallel C [18], or multidisciplinary applications [7].

The DRMS support [17] achieves reconfigurable SPMD codes but not a true self-adaptivity because most responsibility for node and data repartitioning is still left to the programmer. More related to Dame's goals is the Adaptive Multiblock Parti (AMP) project [12]. An AMP programmer explicitly activates the reconfiguration support through the `remap` function. Unlike the `dml_check_load` primitive that specifies only the reconfiguration points and the domains to be reconfigured, the `remap` function requires also an explicit update of all variables and data structures that reconfiguration may affect. For this reason, a complete self-adaptivity could be achieved by the ongoing project of integrating AMP with a supercompiler, such as HPF [14], that can autonomously indicate the variables in the `remap` function.

The integration of Dame with a supercompiler would mainly address programming issues than reconfiguration aspects. The main reason is that the `dml_check_load` function is designed in such a way that it does not require the explicit specification of the data structures subject to modifications. Indeed, the semantic flexibility of Dame primitives already allows the code to self-adapt to runtime variations thus masking them to the programmer and high-level program. Another important difference of Dame from related projects is the possibility of automatically choosing the load balancing interval. Such a solution is useful to increase the degree of transparency and to optimize the performance of the load balancing support.

6 Conclusions

The importance of frameworks that guarantee self-adaptivity of SPMD applications has a twofold motivation. This paradigm is the most widely adopted for parallel programs (some estimations say about 80%) and represents the typical output of supercompilers. Moreover, the typical platform on which executing a parallel application is changing with respect to that of some years ago. Interconnected or shared platforms are becoming popular. Therefore, it is likely that a modern programmer or a supercompiler does not (want to) know the exact platform on which its application will run, but wants that code and performance portability is guaranteed at least for some large class of platforms.

This paper presents the Dame solution that enriches SPMD message-passing applications with self-adaptivity for the broad class of platforms consisting of nodes with local memory. This attribute is achieved with no additional efforts on the programmer. To preserve the simplicity of the SPMD model, we provide the programmer with an *abstract machine interface*, a *scheduler*, and a *runtime environment*. The abstract machine interface provides the programmer with a regular platform and data decomposition model. The scheduler adapts the computation written for the abstract machine to the real platform. The runtime environment performs platform reconfiguration and translates the primitives referring to the virtual platform and virtual decomposition into accesses to the actual platform.

Dame simplifies SPMD programming and provides code portability without sacrificing performance. The experimental results demonstrate that using the Dame primitives does not add significant overheads to a PVM/MPI code that is not even reconfigurable. Moreover, we verified that the benefits of Dame adaptivity are more appreciable when the platform is subject to intense power variations.

We are examining how to enrich Dame with a dynamic fault recovery mechanism. Another interesting development is the integration of Dame adaptivity mechanisms into a supercompiler such as HPF to generate portable SPMD code that does not require explicitly message passing and local memory management.

Acknowledgements

This research is supported by the Italian Ministry of University and Scientific Research in the framework of the project *High Performance Systems for Distributed Applications*.

References

1. G. Agrawal, A. Sussman, J. Saltz, "An integrated runtime and compile-time approach for parallelizing structured and block structured applications", *IEEE Trans. on Parallel and Distributed Systems*, v. 6, n. 7, pp. 747–754, July 1995.
2. J.N.C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, P. Stephan, "Dome: Parallel programming in a distributed computing environment", Proc. *10th Int. Parallel Processing Symposium* (IPPS'96), Honolulu, April 1996.
3. N. Carriero, D. Kaminsky, "Adaptive parallelism and Piranha", *IEEE Computer*, v. 28, n. 1, Jan. 1995.
4. J. Casas, R. Konuru, S.W. Otto, R. Prouty, J. Walpole, "Adaptive load migration systems for PVM", *Proceedings of Supercomputing '94*, Washington, DC, pp. 390–399, Nov. 1994.
5. M. Cermele, M. Colajanni, "Nonuniform and dynamic domain decomposition for hypercomputing", *Parallel Computing*, v. 23, n. 6, pp. 697–718, June 1997.
6. M. Cermele, M. Colajanni, and S. Tucci, "Check-load interval analysis for balancing distributed SPMD applications", Proc. of *Int. Conf. on Parallel and Distributed Techniques and Applications*, Las Vegas, v. 1, pp 432–442, June 1997
7. B. Chapman, M. Haines, P. Mehrotra, H. Zima, J. Van Rosendale, "Opus: A coordination language for multidisciplinary applications", Tech. Rep. TR-97-30, ICASE, June 1997.
8. A.L. Cheung, A.P. Reeves, "High performance computing on a cluster of workstations", Proc. *1st Int. Symp. on High-Performance Distributed Computing*, Syracuse, NY, pp. 152–160, Sept. 1992.
9. M. Colajanni, M. Cermele, "Dame: An environment for preserving efficiency of data parallel computations on distributed systems", *IEEE Concurrency*, v. 5, n. 1, pp. 41–55, Jan.-Mar. 1997.
10. D. Cronk, M. Haines, P. Mehrotra, "Thread migration in the presence of pointers", Tech. Rep. TR-96-73, ICASE, Dec. 1996.
11. P. Dinda, D. O'Hallaron, J. Subhlok, J. Webb, B. Yang, "Language and run-time support for network parallel computing", Proc *8th Int. Work. on Languages and Compilers for Parallel Computing* (LCPC'95), Columbus, OH, Aug. 1995.
12. G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, J. Saltz, "Runtime and compiler support for programming in adaptive parallel environments", *Scientific Programming*, v. 6, Jan. 1997.
13. A.S. Grimshaw, J.B. Weissman, W.T. Strayer, "Portable run-time support for dynamic object-oriented parallel processing", *ACM Trans. on Comp. Systems*, v. 14, n. 2, pp. 139–170, May 1996.
14. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1993.
15. T. Kunz, "The influence of different workload descriptions on a heuristic load balancing scheme", *IEEE Trans. on Software Engineering*, vol. 17, no. 7, pp. 725–730, July 1991.
16. M. Litzkow, M. Livny , M.W. Mutka, "Condor - A Hunter of Idle Workstations", *Proc. of the 8th International Conference of Distributed Computing Systems*, pp. 104-111, June 1988.
17. J.E. Moreira, K. Eswar, R.B. Konuru, V.K. Naik, "Supporting dynamic data and processor repartitioning for irregular applications", Proc. *IRREGULAR'96*, Santa Barbara, pp. 237–248, Aug. 1996.

18. N. Nedeljkovic, M.J. Quinn, "Data-parallel programming on a network of heterogeneous workstations", *Concurrency: Practice and Experience*, v. 5, n. 4 , pp. 257–268, June 1993.
19. R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, G. Fox, "Runtime support and compilation methods for user-specified irregular data distributions", *IEEE Trans. on Parallel and Distributed Systems*, v. 6, n. 8, pp. 815–829, Aug. 1995.
20. B.S. Siegell, "Automatic generation of parallel programs with dynamic load balancing for a network of workstations", Tech. Report CMU-CS-95-168, Carnegie Mellon University, May 1995.
21. M.H. Willebeek-LeMair, and A.P. Reeves, "Strategies for dynamic load balancing on highly parallel computers", *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 9, pp. 979–993, Sept. 1993.

This article was processed using the LaTeX macro package with LLNCS style