Dynamic data decomposition in a message-passing environment

M. Angelaccio, M. Colajanni *

Dipartimento di Informatica, Sistemi e Produzione, Facoltà di Ingegneria, Università di Roma "Tor Vergata", Via della Ricerca Scientifica, Roma, 00133, Italy

Received 8 May 1995; revised 8 October 1996; accepted 28 October 1996

Abstract

The performance of a data parallel program is critically dependent on the data decomposition that the programmer chooses at implementation time. This choice must take into account a combination of different factors such as the kind of the problem, the machine architecture and the data domain size. When these elements are known before execution, the programmer can adopt traditional message-passing languages and optimise performance by means of programs which are dependent on the chosen data decomposition. On the other hand, when the factors that determine the best decomposition are known at run-time only, adequate efficiency can be achieved by a code that dynamically adapts its computation/communication pattern to various decompositions. To assist the programmer in the implementation of a decomposition-independent code, we propose a new programming environment, namely PLUS. It provides the programmer with message-passing primitives that avoid the specification of a data decomposition during the implementation phase, and a run-time support that permits dynamic changes among regular decompositions without affecting the implemented program.

Keywords: Data decomposition-independence; SPMD algorithms; Message-passing primitives; Multicomputers; Run-time support

1. Introduction

The data parallel or Single Program Multiple Data (SPMD) paradigm is the most widely adopted model for implementing scientific programs on multicomputers. By following this programming style, the programmer generates a single code that every node executes on a different portion of the data domain. In traditional message-passing languages, such as the MPI standard [19], the implementation of an SPMD program requires a data decomposition phase in which the programmer maps data onto nodes, and a parallel program generation phase in which the programmer inserts message-passing primitives by taking into account the chosen decomposition. This programming approach produces a decomposition-dependent code, that is a program which works for a single data decomposition (Fig. 1(a)).
When the factors that determine the best data decomposition are static and known at implementation time, a decomposition-dependent code can achieve satisfying performance. However, there are several instances in which the best decomposition can be decided at run-time only. Some examples are the irregular computations in which the workload is intrinsically subject to dynamic changes; the programs running on systems with faulty nodes, provided that a run-time process/data reconfiguration

Fig. 1. (a) Traditional message-passing environments (decomposition-dependent). (b) PLUS programming environment (run-time decomposition-independent). (c) Supercollider programming environments (compile-time decomposition-independent).
support is available; the regular computations in which the domain size and/or target machine architecture are not known during the parallel algorithm design. In these cases, an adequate efficiency can be achieved only by a decomposition-independent code that at run-time adapts its computation/communication pattern to the decomposition that is best. However, the explicit implementation of such a code would require heavy additional efforts to the programmer. For this reason, we present a programming environment called PLUS that is able to support dynamic decomposition changes in a transparent way. In particular, this framework allows the programmer to write an SPMD program by taking into account a meta-decomposition from which it is possible to generate a wide set of regular data decompositions at run-time. PLUS satisfies the decomposition-independence property by means of a twofold contribution:

- A library of new message-passing primitives that, being based on the meta-decomposition, provides the programmer with a level of abstraction higher than traditional languages;
- A run-time support that translates any inquiry to the meta-decomposition into references to the decomposition that is best at that time.

Since the parallel program is implemented through the PLUS primitives, the resulting code can dynamically adapt its computation/communication pattern to a new decomposition without any modification in the program (Fig. 1(b)). This approach is completely different from other proposals for achieving a decomposition-independence code. For example, the first generation of supercompilers such as Superb, Kali, Fortran D, Vienna Fortran, HPF [20,15,14,9,13], take as input a sequential program and some decomposition annotation and produce the parallel code automatically. Even if a program for a different decomposition simply requires a new annotation and re-compilation, the code generated by these supercompilers remains decomposition-dependent (Fig. 1(c)). On the other hand, the second generation of supercompilers [1,10,16] and parallel libraries such as Parti and Toolbox [18] aim at supporting run-time decomposition-independence for irregular computations.

Differently from supercompilers, PLUS hides the actual data decomposition and underlying architecture. However, it requires the programmer to explicitly insert communications. At present, PLUS can be used for regular computations by an experienced programmer which is able to manage message-passing constructs. There is no theoretical limit that prevents an integration of the PLUS primitives with the communication synthesis phase of a supercompiler.

The paper is organised as follows. Section 2 focuses on the main contributions underlying the PLUS environment, that is the meta-decomposition and the decomposition-independent primitives. Section 3 outlines the PLUS programming model. Section 4 describes two PLUS programs and demonstrates how it is possible to obtain a parallel code that adapts itself to dynamic decomposition changes. Section 5 shows the advantages of using the PLUS environment when the best decomposition cannot be determined during the implementation phase. Section 6 sketches some future developments and gives our conclusive remarks.

2. The PLUS solution

2.1. SMD meta-decomposition

Matrix decompositions can be classified on the basis of a dimensional approach:

- A 1D matrix decomposition (that is, row or column) is defined by means of a function \( \delta_X : A \rightarrow P \) that maps \( n \times n \) matrix entries \( A = \{a_{ij}\}_{i,j \leq n} \) into a set \( P \) of \( p \) processors. The function \( \delta_X \) can be defined as a composition of two functions \( \delta_X = \pi_X \circ \phi \), where

\[
\pi_X : A \rightarrow P(A)
\]

is the matrix partition that decomposes the matrix in accord to the specified
rule $X$ ($X = R$ for row, $X = C$ for column);  
$\phi : P(A) \rightarrow P$ is the mapping strategy that  
assigns the parts (rows or columns) to the  
processors; common mapping strategies are  
block, cyclic, reflection.

- A 2D matrix decomposition assumes the processors arranged as a grid $P_1 \times P_2$. It can be viewed as the result of a tensor product of two 1D decompositions associated to each co-ordinate. Hence, a 2D decomposition can be denoted by a function having type $\delta : A \rightarrow P_1 \times P_2$ and defined by $\delta = (\pi_p \circ \phi) \otimes (\pi_c \circ \phi)$, where $\pi_p$ and $\pi_c$ apply to row and column indexes and $\phi$ assigns the corresponding entries to $P_1$ and $P_2$, respectively. A 2D decomposition can be seen as a general-purpose decomposition that covers a wide spectrum of regular decompositions (see, for example, the classification carried out in [7]). This approach has been proposed with slight differences as Cartesian clustering scheme [17], Distribute statement with attribute in Fortran D, Square Block Scattered decomposition in ScaLa-PACK [12], DIST() annotation in Vienna Fortran.

We propose a different general-purpose decomposition, namely Subnet Matrix Decomposition (SMD), that represents the crucial support for adding decomposition-independence features to data parallel programs. SMD is a meta-decomposition because it originates several regular data decompositions. It achieves the same generality as traditional 2D decompositions, but in a different way. Basically, SMD is a 2D decomposition that is obtained from the 1D decomposition function $\delta : A \rightarrow P$ by replacing each processor in $P$ with an appropriate subnet. In a hypercube topology, each subnet corresponds to a subcube; in mesh and torus architectures, it corresponds to a row of processors.

In order to describe SMD in a more formal way, let $H_r$ be a partition of $r$-dimensional subnets ($0 \leq r \leq q$). Hence, SMD can be seen as a function $\Delta_r : A \rightarrow H_r$ that returns the subnet identifier to which the entries are assigned. In a way analogous to $\delta$, the function $\Delta_r$ is defined as a composition of two functions $\Delta_r = \pi \circ \Phi_r$, where

- $\pi : A \rightarrow P(A)$ is the matrix partition that carries out a row decomposition of the matrix $A$;
- $\Phi_r : P(A) \rightarrow H_r$ is the subnet mapping strategy that assigns each row to a subnet belonging to $H_r$. This function is characterised by the pair (inter-subnet mapping, intra-subnet mapping) that describes how rows are partitioned among the subnets, and how row entries are assigned to each subnet. Attributes of each component of the pair are block, cyclic, etc.

Common matrix decompositions are generated from SMD by setting the so-called subnet partition parameters: [subnet dimension $r$, inter-subnet mapping, intra-subnet mapping]. In particular, the row matrix decomposition comes from SMD by setting $r = 0$ because each row is assigned to a 0-dimensional subnet, that is a single node. The inter-subnet attribute determines the way in which these rows are assigned among subnets (e.g., block, scattered, reflection). The column matrix decomposition comes from SMD by setting $r = q$ because each row is assigned to a $q$-dimensional subnet, that is the entire set of nodes. In this case, the intra-subnet attribute specifies how to assign the columns to each node. The square matrix decomposition comes from SMD by setting $r = q/2$, while SMD generalises to a grid decomposition, when $r \neq q/2$. Different combinations of the inter/intra-subnet mapping originate other 2D decompositions [3].

### 2.2 The PLUS primitives

SPMD programming requires the specification of several issues such as node identification, data exchange among nodes, owner compute rules, global operations. Traditional programming environments furnish these primitives in a decomposition-dependent form. Conversely, PLUS provides the programmer with a new set of message-passing primitives that he has to use by taking into account the SMD
meta-decomposition. At run-time, PLUS automatically adapts the scope of each primitive to the subnet partition parameters that denote the last data decomposition which is generated from SMD.

The theoretical foundations of PLUS are in [3]. There, the idea of SMD and first examples of PLUS were introduced to achieve unification of parallel programs. At present, PLUS is built as a layer on top of CS-Tools, NX/2 and Vertex. Here, we outline and classify the most representative primitives available in PLUS without aiming to furnish a user manual. More details can be found in [4], while some examples of usage are in Section 4 and Section 5.

In accord to the taxonomy illustrated in Fig. 2, the PLUS primitives have been grouped into two classes and five sub-classes: data decomposition, loop-independent, loop-dependent, communication primitives, global reduction operations.

(1) **Data decomposition.** The decomposition primitives determine the node partition and data decomposition. In particular, the `partition()` routine takes as argument the subnet partition parameters and determines a partition of the active nodes into subnets. The subnet partition setting is dynamic, in the sense that it can be modified at run-time. The `partition()` routine returns two vectors, namely `n_row[]` and `n_col[]`, that contain the identifiers of the nodes belonging to the same row subnet and column subnet, respectively.

These vectors are useful in most of the communication primitives and global reduction operations. For example, a change in the subnet partition parameters affects the scope of the message-passing primitives by means of a modification of these vectors.

Once the partition has been determined, the programmer can call the `mydata()` primitive that takes as argument the global domain (for example, a matrix) and returns the local data entries which have been assigned to the calling node.

(2) **Loop-independent primitives.** The loop-independent primitives give information about node and subnet identifiers. They should be inserted at the beginning of the program since they are independent of a specific iteration step. We use `g_mynode()` and `l_mynode()` as identification primitives. They return absolute and local (that is, relative to `myrownet()` identifier of the node processing the program. Moreover, `myrownet()` and `mycolnet()` return the identifier of the row subnet and column subnet containing `g_mynode()`. Other global information is obtained through the `numnodes()`, `num_subnet()`, `size_subnet()` functions that return the number of active processors, subnets in a partition, and processors in a subnet, respectively. Any modification of the subnet partition parameters affects most of the loop-independent primitives.

(3) **Loop-dependent primitives.** The loop-dependent primitives take as argument a global index that usually corresponds to a loop index. They select the matrix entries which are located on a particular node or identify the node holding certain data. In PLUS these primitives are organised in the following way:

- **Subnet data identifiers**, such as `sub_row()` that takes as argument a global row index and, in accord to the subnet partition parameters, returns the identifier of the subnet holding this row. An analogous effect on columns is obtained through the `sub_col()` primitive.
- **Local and global matrix indexes**, such as `l_row()` that takes as argument a global row
index and returns the equivalent local index in g_mynode(). The \texttt{l_col()} primitive gives the analogous results for the columns, while \texttt{g_row()} and \texttt{g_col()} produce the inverse result of \texttt{l_row()} and \texttt{l_col()}, respectively.

- **Local data extraction**, such as \texttt{myrow()} that takes as argument a matrix name and a global row index and returns the local row entries that correspond to that index in \texttt{g_mynode()}.

- **Global data predicates**, such as \texttt{in_myrownet()} and \texttt{in_mycolnet()} that takes as argument a global row (column) index and returns \texttt{TRUE} if the row (column) entries have been assigned to \texttt{myrownet()} (\texttt{mycolnet()}), \texttt{FALSE} otherwise.

4) **Communication primitives.** There are two types of communication primitives which are available in PLUS: node-to-node such as \texttt{send(msg, dest)}, and node-to-subnet. Among these latter, \texttt{row\_fan\_out()} broadcasts messages throughout the \texttt{myrownet()} nodes, the identifiers of which are contained in the \texttt{n_row[]} vector. \texttt{col\_fan\_out} sends messages to \texttt{mycolnet()} nodes, that are identified by the \texttt{n_col[]} vector. As example, in a mesh topology, \texttt{col\_fan\_out()} primitive corresponds to a vertical fan-out, while a \texttt{row\_fan\_out()} corresponds to an horizontal fan-out.

5) **Global reduction operations.** The global reduction operations differ from the communication primitives because they originate from a subnet and, usually, involve a reduction operator. The resulting data are left in a single node by the \texttt{subnet-to-node} primitives (\texttt{row\_fan\_in()}), \texttt{col\_fan\_in()}), and in all the subnets by the \texttt{subnet-to-subnet} primitives (\texttt{grow\_fan\_in()}, \texttt{gcol\_fan\_in()}). For example, \texttt{row\_fan\_in(msg, root, red. oper.)}, called by the nodes of a subnet, is the reverse of \texttt{row\_fan\_out()}. In this case \texttt{msg} is gathered into the root node from the nodes of the entire subnet: the leaf nodes originate the flow and the direction is towards the root node; intermediate nodes wait until all of the messages from their immediate descendants have arrived, possibly apply the reduction operator (such as null, maximum, minimum, sum, multiplication, logical), and send the result to their parent.

A subnet-to-subnet primitive such as \texttt{grow\_fan\_in(msg, red.oper.)} behaves analogously to \texttt{row\_fan\_in()} with the difference that now the result is known by all of the nodes involved in this operation.

3. PLUS programming model

One advantage of PLUS with respect to more abstract environments, such as \textit{Linda} [8], is that it guarantees the decomposition-independence of the code still preserving a programming model which is similar to the classic SPMD paradigm. The main difference is that the PLUS programmer has to implement the code on the basis of the SMD meta-decomposition, that is a logical 2D decomposition. The entire mechanism that supports the decomposition-independence (that is, generation of an actual decomposition from SMD, and consequent adaptation of the scope of the primitives) is completely hidden to the programmer. The only visible interface is represented by the \texttt{subnet partition parameters} that the programmer has to set and/or modify at run-time by means of the \texttt{partition()} primitive. A PLUS program conforms to the template which is sketched in Fig. 3.

This programming model has been demonstrated satisfactory in all of the implementations of scientific code our research group has carried out in about three years of experiments. In addition, we have verified that if the programmer adequately adopts the PLUS primitives, the execution times are comparable to those obtained by decomposition-dependent codes written in traditional languages. The PLUS overheads do not exceed 4% when compared to programs
Fig. 3. PLUS program template.

explicitly written for 2D decompositions, while the
difference may reach 10% in the case of programs
dependent on 1D decompositions. This higher over-
head is due to the fact that the PLUS primitives are
introduced on the basis of the SMD, that is essen-
tially a 2D decomposition. Therefore, the addi-
tional cost of a PLUS program is mainly due to the
primitives that become idle when a 1D decomposition is
settled (see Section 4). The satisfying performance of
the PLUS environment is due to a combination of
efficiency in the implementation and accuracy in the
use of PLUS primitives by the programmer. Let us
distinguish the following cases:

• The calls to data decomposition and loop-inde-
pendent primitives occur essentially in the initiali-
sation phase and do not affect the execution time.
In particular, the programmer should assign the
result of a loop-independent call to a variable and
use it in the other parts of the program.

• At run-time, the calls that concern communi-
cations and global reduction operations do not in-
duce additional overheads when a 2D data de-
composition is active, because the scope of their
messages depends on inquiries that require array
accesses only. Conversely, these primitives are
more costly when a 1D decomposition is settled
because some of these calls are unnecessary.

• The loop-dependent primitives are the most cri-
tical calls from the performance point of view
because most of them require some algebra. To
achieve a better efficiency, these primitives are
implemented as macros. The costs are not higher
than those explicitly implemented in a traditional
language. However, we have observed a signifi-
cant performance improvement by a cautious use
of these primitives involving operations that are
hidden to the programmer. For example, if the
programs need more times the result of the same
primitive inside a loop, it is preferable to call this
primitive in the outer loop, assign the result to a
variable, and refer to this latter. In such a way,
we can avoid the additional overheads due to the
multiple computations of the primitive operation.

It should be noted that, besides the decompo-
sition-independent properties, PLUS satisfies also
portability requirements because it provides a ma-
cine-independent programming environment. Even
if there is no theoretical limit to the portability of
PLUS, the programs run efficiently on multicomput-
ers with at least a 2D topology (the sufficient lattice
for most matrix computation).

4. Examples of programs

We illustrate how to use the PLUS primitives in
two representative classes of algorithms that cover
most of the scientific code: a multicast algorithm
and a near-neighbour algorithm. The presentation
focuses on the main aspects influenced by the PLUS
primitives (i.e., loop ranges, communications, locality checks). After each program description, we anal-
alyse the automatic transformation that different de-
compositions induce on the semantic of the PLUS
primitives.
/* program: PLUS_LUP */
for k=0 to n-1 do
  /* Row Partial Pivoting */
  if I own column_k
    then compute(pivot index);
    broadcast(pivot index);
    else receive(pivot index);
  if (k<>pivot index)
    then exchange(row_k, row_pivot);
  /* LU factorisation */
  if I own row_k
    then broadcast(row_k);
    else receive(row_k);
  if I own column_k
    then forall (i>k that I own) do
      multi_k = a_ik/a_ kk;
      broadcast(multi_k);
    else receive(multi_k);
  forall (i>k that I own) do
    forall (j>k that I own) do
      a_ij = a_ij - multi_k a_kj;

(a)

/* program: PLUS_LUP */
for (k=0; k<n-1; k++)
  /* Row Partial Pivoting */
  if (in_mycolnet(k)) /* Compute pivot if column_k is in my node */
    gool_fan_in(mycol(A, k), max_index);
    row_fan_out(pivot index);
    else receive(pivot index);
  if (k!=pivot index)
    a_exchange(A, pivot index, k);
  /* LU factorisation */
  if (in_myrownet(k))
    uprow=myrown(A, k);
    col_fan_out(uprow);
    else receive(uprow);
  if (in_mycolnet(k)) /* Compute and broadcast multipliers */
    for (i=l_row(k); i<l_row(n-1); i++)
      multi[k] = A[i,k]/uprow[k];
      row_fan_out(multi);
    else receive(multi);
  for (j=l_row(k); j<l_row(n-1); j++) /* Update sub-matrix */
    for (j=l_row(1); j<l_col(n); j++)

(b)

Fig. 4. (a) Pseudo-code of LU factorisation. (b) LU factorisation program in PLUS.
4.1. LU factorisation with pivoting

An example of a multicast program, we consider the LU factorisation algorithm with row partial pivoting for dense matrices (Fig. 4(a)). Each stage \( k \) is composed of four main steps: search in column \( k \) for an element greater than the current pivot \( a_{kk} \); possible swap of \( \text{row}_k \) and \( \text{row}_{\text{pivot}} \); multiplier computation; sub-matrix updating. The corresponding version in PLUS is given in Fig. 4(b). The outer loop \( k \) is not affected by the parallelisation, while the ranges of loops \( i \) and \( j \) are modified by \( \text{l\_row}() \) and \( \text{l\_col}() \) in consequence of their operations on local data. The search for pivot is carried out by a global operation \( \text{gcol\_fan\_in}() \) that uses \text{max\_index} as reduction operator. The swap of \( \text{row}_k \) and \( \text{row}_{\text{pivot}} \) is executed by the PLUS macro \( \text{s\_exchange}() \) that involves only the subnet(s) holding \( \text{row}_k \) and \( \text{row}_{\text{pivot}} \). Before the updating phase, a \( \text{col\_fan\_out}() \) of the multipliers and a \( \text{row\_fan\_out}() \) of the \( \text{row}_k \) entries are required.

The PLUS program shown in Fig. 4(b) is explicitly written for SMD, but it runs for any other data decomposition thanks to the flexibility of the PLUS primitives. Let us focus on 1D decompositions that affect both the scope and the semantic of the PLUS primitives.

By setting the \text{subnet\_partition\_parameters} to \text{row\_decomposition}, the subnets are reduced to nodes. As a consequence, \( \text{myrow\_net}() \) becomes equivalent to \( \text{g\_mynode}() \); local column indexes become global indexes, that is \( \text{l\_col}(k) - k \) and \( \text{l\_col}(n) = n \); \( \text{row\_fan\_out}() \) is redundant; \( \text{col\_fan\_out}() \) is equivalent to a global broadcast; test whether a column index belongs to \( \text{g\_mynode}() \) is always satisfied; \( \text{myrow}(A, k) \) selects all of the \( \text{row}_k \) entries.

In the instance of \text{column\_decomposition}, the row subnets are equivalent to the entire set of processors. Therefore, the primitives related to these subnets (such as broadcasting elements between subnets and testing whether an entry is in \( \text{myrow\_net}() \)) become redundant. In addition, local row indices become global indices; \( \text{mycol}(A, k) \) selects all of the \( \text{column}_k \) entries; test whether a row index belongs to \( \text{myrow\_net}() \) is always satisfied. With regard to communications, \( \text{row\_fan\_out}() \) becomes a global broadcast, while \( \text{col\_fan\_out}() \) and \( \text{col\_fan\_in}() \) become useless because node destination is \( \text{g\_mynode}() \) itself.

The PLUS environment allows the programmer to develop a single program for any decomposition and machine architecture. Fig. 5(a) and Fig. 5(b) show the experimental results obtained by the same PLUS code (that is, the LU factorisation shown in Fig.

![Efficiency](a)

![Efficiency](b)

Fig. 5. (a) Efficiency of LU factorisation with row partial pivoting on hypercube Intel iPSC with 16 nodes for different values of the subnet dimension \( r \). The values \( r = 0, r = 2 \) and \( r = 4 \) correspond to \text{row\_square} and \text{column\_decomposition}, respectively. (b) Efficiency of LU factorisation on Meiko Computing Surface with 8 nodes for different values of the subnet dimension \( r \).
4(b)) running on Intel iPSC/2 and Meiko Computing Surface that have hypercube and 2D torus topology, respectively. For the experiments, we chose dense matrices that for any decomposition required a row swap in the pivoting phase of each step k. The efficiency curves relative to different subnet dimensions confirm the optimality of the square decomposition [17,5] for the matrix sizes we have considered. An interesting trade-off among the matrix decompositions exists, instead, if we focus on the pivoting phase only [5]. In particular, for some matrix sizes the row decomposition could result better than square decomposition, while the serial pivot search for the column decomposition usually causes poor efficiency.

4.2. SOR algorithm

We consider the SOR method as example of near-neighbour communication algorithm. It works on a grid domain and computes at each iteration a grid value as linear combination of the four adjacent points. Since, in the update step, each processor requires neighbour data, at the beginning of i and j loops, it occurs to receive adjacent rows (from north and south subnets) and adjacent columns (from east and west nodes). The adjacent indexes are computed by means of the g_row() and g_col() primitives, while the adjacent node destinations are obtained through the n_col[] and n_row[] vectors. The PLUS program, described in Fig. 6, is explicitly

/* program: PLUS_SOR */

for (k=0; k<convergence; k++)
    for (i=1_row[1]; i<1_row[n-1]; i++)
        if [(!in_myrownet(i+1)) /* if row i+1 not in myrownet */
          dest=n_col[sub_row[g_row(i)+1]]; /* homologous node in next subnet */
          send(myrow(A,i), dest);
          receive(myrow(B,i+1));
        ]
        if [(!in_myrownet(i-1)) /* if row i-1 not in myrownet */
          dest=n_col[sub_row[g_row(i)-1]]; /* homologous node in previous subnet */
          send(myrow(A,i), dest);
          receive(myrow(B,i-1));
        ]
    for (j=1_col[1]; j<1_col[n-1]; j++)
        if [(!in_mycolnet(j+1)) /* if column j+1 not in mynode */
          dest=n_row[sub_col[g_col(j)+1]]; /* successive node */
          send(mycol(A,j), dest);
          receive(mycol(B,j+1));
        ]
        if [(!in_mycolnet(j-1)) /* if column j-1 not in mynode */
          dest=n_row[sub_col[g_col(j)-1]]; /* previous node */
          send(mycol(A,j), dest);
          receive(mycol(B,j-1));
        ]
        <test for convergence>
    for (i=1_row[1]; i<1_row[n-1]; i++)
        for (j=1_col[1]; j<1_col[n-1]; j++)
            B[i,j] = A[i,j];

Fig. 6. SOR program in PLUS.
written for SMD but it runs for any regular decompositions. In the instance of subnet partition parameters set to column decomposition, the following issues hold true: test whether a row index in myrownet() is always satisfied because !in_myrownet(i+1) is false; communications are carried out only when local column boundaries are reached, that is !in_mycolnet(j+1) is true for j-1_col(1) and j-1_col(#-2) only.

5. Computations requiring run-time decomposition changes

In this section we analyse some examples of regular problems that require a dynamic data decomposition to achieve adequate performance. In particular, we show how a PLUS program can easily evaluate at run-time the convenience of switching from a decomposition to another.

5.1. Dynamic data size

There are many regular computations for which the best decomposition depends on the data size of the problem to solve. When this value is an input parameter, any choice of the decomposition during the parallel algorithm design risks to cause serious inefficiencies. In these instances, the best solution is to postpone the data decomposition choice to runtime. Let us consider the LU factorisation and SOR algorithms that have been described in Section 4.

```c
/* HPL host program for LU pivoting */
getcube(..);
while (end of input file) do
{
   <read N>;
   p=numnodes(); rp=SQRTP(p);
   if ((comp_time() N/p+comm_time() *p) >
       (comp_time() N/rp+comm_time() *rp))
      then load_prog("LUP_square");
      else load_prog("LUP_column");
   load_data(A,N);
   <wait node program termination>;
}

/* PLUS host program for LU pivoting */
getcube(..);
load_prog("PLUS_LUP");
while (end of input file) do
{
   <read N>;
   p=numnodes(); rp=SQRTP(p);
   if ((comp_time() N/p+comm_time() *p) >
       (comp_time() N/rp+comm_time() *rp))
      then decomposition_par=nodedim()/2; /* square decomposition */
      else decomposition_par=0; /* row decomposition */
   partition(A,N,decomposition_par);
}
```

Fig. 7. Host programs that call node programs (LU factorisation) with variable decomposition in the case of a supercompiler and PLUS, respectively.
The pivoting phase of LU factorisation algorithm could present a trade-off between square and row decomposition for increasing values of the matrix size $N$. In the former case, the cost for the pivoting phase is given by $N/\sqrt{p}$ comparisons and $\sqrt{p}$ communications, while in the latter the cost is given by $N/p$ comparisons and $p$ communications. Let $\alpha$ denote the comparison time and $\beta$ the communication time in a given parallel machine. If $N$ is an input parameter, we obtain the following dynamic decomposition requirement that can be added to the PLUS program [5]:

$$\text{if } (\alpha N/p + \beta p) > (\alpha N/\sqrt{p} + \beta \sqrt{p})$$

then decomposition = square block
else decomposition = row block

In Fig. 7 we compare two host programs for the LU factorisation codes generated by a supercompiler and the PLUS environment, respectively. An evident advantage of PLUS is that a single code works well for different decompositions. Conversely, a supercompiler needs to generate and load various programs to allow a code to change decomposition at run-time.

An analogous problem may occur if we consider the SOR algorithm. Since the grid domain can be distributed among the $p$ processors in several ways, the decomposition choice has a crucial influence on the SOR algorithm too. For 1D decompositions (row/column), each processor requires two $N$-length messages per iteration step, while 2D decompositions (e.g., square block) require four messages having dimension $N/p$. As a consequence, once known the communication cost in a given machine (that is, startup time $S$ and data propagation time $\delta$), the best decomposition will be simply determined by the $N/p$ ratio.

When the SOR code should work for different grid dimensions (Fig. 8) or when $N$ is an input parameter, the best decomposition can be determined at run-time only. In these instances, a PLUS code is able to dynamically adapt its computation/communication pattern to the decomposition that is considered most efficient. To this purpose, the following condition has to be added to the host program (as done for the LU decomposition in Fig. 7):

$$\text{if } (4'(S + \delta N/p)) < (2'(S + \delta N))$$

then decomposition = square block
else decomposition = column block

Let us now investigate for which cases a SOR program with 2D decomposition is preferable to a 1D decomposition in three actual multicomputers:

- In an Intel iPSC/860, the startup and propagation time are given by $S = 470 \mu s$ and $\delta = 0.77 \mu s/\text{byte}$, respectively. Therefore, the square decomposition is preferable when the grid dimension is $N > 620$ for $p = 128$, or when $N > 630$ for $p = 64$.
- An nCUBE2 has $S = 210 \mu s$ and $\delta = 0.58 \mu s/\text{byte}$. Therefore, the threshold value that makes preferable the square decomposition is lower than in an iPSC/860, that is $N > 367$ for $p = 128$, and $N > 373$ for $p = 64$.
- An Intel Paragon has $S = 60 \mu s$ and $\delta = 0.5 \mu s/\text{byte}$. Therefore, on this machine the square decomposition is preferable only when $N > 1219$ for $p = 128$, and $N > 1238$ for $p = 64$.
5.2. Further applications of PLUS

The decomposition-independent properties and related transparent supports make the PLUS environment very useful in several applications. In addition to the instances shown in Section 5.2, the following applications represent other interesting examples where PLUS can be actually applied.

PLUS is a powerful tool that can be used to write a library of decomposition-independent linear algebra subroutines such as those proposed in [12] and [11]. The main advantage is that each algorithm would require a single PLUS code instead of a code for each feasible decomposition.

PLUS is also a framework that can be adopted for the performance analysis, both analytical and experimental, of parallel programs [2]. Through PLUS, a programmer can write a single decomposition-independent program, and then find via experimentation the most satisfactory decomposition for this algorithm on a given architecture. A simple settling of the PLUS decomposition parameters avoids the drawback of rewriting a different test program for each decomposition as required by other environments.

PLUS may also represent the basis for building fault-tolerant SPMD algorithms in multicomputers without hardware redundancy. Usually, this requires run-time supports that carry out workload and process migration. On the other hand, a fault-tolerant support for SPMD algorithms can restrict workload reconfigurations to data migrations because the same code is loaded on all of the processors. However, since any data reconfiguration induces a new data decomposition at run-time, these supports can be applied to decomposition-independent algorithms such as those provided by PLUS. By combining PLUS with the data reconfiguration algorithms described in [6], it is possible to achieve a programming environment that produces SPMD codes for which when no processor fails the performance is fully guaranteed, whereas when some processor fails the performance degrades gracefully until at least one processor is working. Moreover, PLUS hides to the programmer most part of the support that allows an SPMD program to continue its execution in case of some faulty processor.

6. Conclusions

In this paper we have illustrated main capabilities of PLUS, a new programming environment that adds decomposition-independent property to traditional message-passing languages. PLUS consists of a collection of high-level primitives which are based on the SMD meta-decomposition, and a run-time support that is able to dynamically adapt the PLUS primitive calls to the best regular decomposition.

PLUS is useful to any regular computations that require dynamic modifications of the domain decomposition. However, the examples demonstrate that the implementation of decomposition-independent algorithms still requires a certain programming effort. For this reason, it is interesting to study the combination of the PLUS environment with a supercompiler. The goal should be a tool that automatically transforms serial programs to decomposition-independent codes by adding explicit calls to the PLUS library. Another on-going activity is the implementation of PLUS on top of libraries such as PVM and MPI.

References


Michele Angelaccio received his Doctorate degree in Computer Science from the University of Pisa, Italy, in 1986. Since 1988 he has been a researcher in Computer Science in the Department of Computer Engineering, University of Roma “Tor Vergata”. His current research interests include parallel algorithms, compiler optimization for multicomputers, and fault-tolerance.

Michele Colajanni received his Laurea degree in Computer Science from the University of Pisa, in 1987, and his Ph.D. degree in computer engineering from the University of Roma “Tor Vergata”, in 1992. He is currently with the Department of Computer Engineering, University of Roma “Tor Vergata” as a researcher. His research interests include parallel processing, fault-tolerance, and performance analysis. Dr. Colajanni is a member of the IEEE Computer Society, the Association for Computer Machinery, and the Society for Computer Simulation.