Compilation of I/O Communications for HPF (in Frontiers'95 - also report A-264-CRI)

Fabien COELHO (coelho@cri.ensmp.fr)
Centre de Recherche en Informatique, École des mines de Paris,
35, rue Saint-Honoré, 77305 Fontainebleau CEDEX, FRANCE
voice: (+33 1) 64 69 48 52, fax: (+33 1) 64 69 47 09

Abstract

The MIMD Distributed Memory architecture is the choice architecture for massively parallel machines. It insures scalability, but at the expense of programming ease. New languages such as HPF were introduced to solve this problem: the user advises the compiler about data distribution and parallel computations through directives. This paper focuses on the compilation of I/O communications for HPF. Data must be efficiently collected to or updated from I/O nodes with vectorized messages, for any possible mapping. The problem is solved using standard polyhedron scanning techniques. The code generation issues to handle the different cases are addressed. Then the method is improved and extended to parallel I/Os. This work suggests new HPF directives for parallel I/Os.

Introduction

The supercomputing community is more and more interested in massively parallel architectures. Only these machines will meet the performances required to solve problems such as the *Grand Challenges*. The choice architecture is the MIMD Distributed Memory architecture: a set of loosely coupled processors linked by a network. Such a design insures scalability, at the expense of programming ease. Since the programmer is not given a global name space, distributed data addressing and low level communications are to be dealt with explicitely. Non portable codes are produced at great expense. This problem must be solved to enlarge the potential market for these machines. The SVM (Shared Virtual Memory) approach [31, 8] puts the burden on the hardware and operating system, which have to simulate a shared memory. The HPF Forum chose to put it on the language and compiler technology [22], following early academic and commercial experiments [25, 14, 38, 40, 11, 12]. The user is given a way to advise the compiler about data distributions and parallel computations, through a set of directives added to Fortran 90. The Forum did not address parallel I/O since no concensus was found.

However MPP machines have parallel I/O capabilities [10] that have to be used efficiently to run real applications [35, 20, 32]. For instance, the TMC's CM5 or the Intel's Paragon have so called I/O nodes attached to their fast network. They are used in parallel by applications requiring high I/O throughput. For networks of workstations, files are usually centralized on a server and accessed through NFS. Such a system can be seen as a parallel machine using PVM-like libraries [24]. These systems are often programmed with a host-node model. To allow nodes to access disk data, one solution is to provide the machine with parallel I/O which ensure I/O operation coherency when many nodes share the same data.

Another solution is to rely on program analyses to determine the data needed on each node, and to generate the communications from one or several processes which directly perform the I/O. This paper presents a technique for such an approach, which may be extended to parallel I/Os as shown in Section 3.2. It focuses on the compilation of I/O communications for HPF: data must be collected to or updated from one or several I/O nodes, for any possible mapping. An array may be mapped onto all processors (*i.e.* it is shared), or distributed, or even partially replicated onto some, depending on the mapping directives. Vectorized messages must be generated between the nodes that have data to input or output and the I/O nodes.

```
      program triangle
      real A(30,30)
chpf$ template T(68,30)
chpf$ align A(i,j) with T(2*i,j)
chpf$ processors P(4,2)
chpf$ distribute T(block,cyclic(5)) onto P
      read *, m
      do 1 j=3, m
         do 1 i=3, m-j+1
 1          print *, A(i,j)
      end
```

Figure 1: Example `triangle`

Let us look at the example in Figure 1: a 2D array `A` is mapped onto a 4×2 processor grid `P`. The I/O statement within the loop nest accesses the upper left of `A` using parameter `m` as shown in Figure 2. Several problems must be solved to compile this program: first, the loop nest must be considered as a whole to generate efficient messages. If the sole `print` instruction is considered, and since it outputs one array element at a

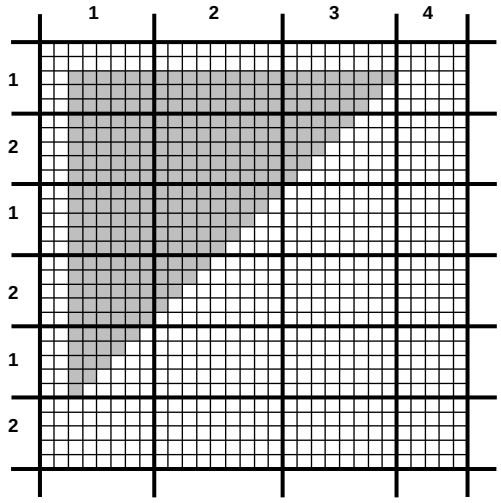

Figure 2: Accessed area for $m = 27$

time, there is no message vectorization opportunity. Second, the accessed array elements are only known at runtime (`m` dependence), so the method must be parametric. Third, the block distribution of the array first dimension is not regular, due to the stride 2 alignment and the odd size of the block: the array block size on the processor grid first dimension oscillates between 8 and 9 elements. Such issues must be managed by the solution. Finally, the generated code must deal with the array local addressing scheme chosen on the nodes to reduce the allocated memory in an efficient way, in order to avoid expensive address computations at runtime.

Section 1 gives the problem a mathematical formulation, and solves it using standard polyhedron scanning techniques. Section 2 describes the generated code to handle the different cases: collect or update of distributed or shared arrays. Section 3 discusses possible improvements and extensions of the technique to parallel I/O. This work suggests new HPF directives to help compile parallel I/O communications.

1 Problem expression

From an I/O statement, a distributed code must be generated. The first step is to formalize the problem in a mathematical way and to show how this problem is solved. This section addresses this point. The next step will be to show how the mathematical resolution is embodied in a generated program, that is how to generate code from the mathematical solution. First, the basic analyses used are presented. Second, the communications required by an I/O statement are formulated in a linear framework. Third, the resolution scheme is briefly outlined.

1.1 Analyses and assumptions

A precise description of the elements accessed by an I/O statement is needed. For each statement and

$$\texttt{A}(\alpha_1,\alpha_2)\texttt{-MUST-READ-}$$
$$\{\alpha_1 + \alpha_2 \leq 1 + m, 3 \leq \alpha_1, 3 \leq \alpha_2\}$$

Figure 3: Region for the I/O loop nest

$$\begin{aligned} \alpha'_1 &= ((2\alpha_1 - 1) \bmod 17) \div 2 + 1 \\ \alpha'_2 &= 5((\alpha_2 - 1) \div 10) + (\alpha_2 - 1) \bmod 5 + 1 \end{aligned}$$

Figure 4: Global $\texttt{A}(\alpha_1, \alpha_2)$ to local $\texttt{A}'(\alpha'_1, \alpha'_2)$

each array accessed within this statement, a set of linear equalities and inequalities describing the region of accessed elements can be automatically derived from the program [36, 37, 6, 7]. A *region* is given an *action* and an *approximation*. The *action* is `READ` if the elements are read and `WRITE` if written. The *approximation* is `MUST` if all the elements are actually accessed and `MAY` if only a subset of these elements will probably be accessed. Figure 3 shows the *region* derived from the I/O loop nest of the running example. The α variables range over the accessed region on each dimension of the array. The linear inequalities select the upper left of the array, except borders.

Moreover the point where to insert the communications to achieve a possible message vectorization must be chosen. The running example may have been expressed with the implied-do syntax, however such loop nests can be dealt with. A simple bottom-up algorithm starting from the I/O instructions and walking through the abstract syntax tree of the program gives a possible level for the communications. For the running example, the I/O compilation technique will be applied to the whole loop nest.

1.2 Linear formulation

The declarations, the HPF directives and the local addressing scheme (Figure 4: each node allocates a 9×15 array $\texttt{A}'$ to store its local part of distributed array `A`, and the displayed formulae allow to switch from global to local addresses) are translated into linear constraints, as suggested in [4]. Together with the *region*, it gives a fully linear description of the communication problem, that is the enumeration of the elements to be sent and received. Figure 5 shows the linear constraints derived for the running example. The α variables (resp. θ, ψ) describe the dimensions of the array (resp. the template, the processors). The α' variables are used for the array local declaration. The γ and δ variables show the implicit linearity within the distribution: the δ's range within blocks, and the γ's over cycles. η_1 is an auxiliary variable.

The first three inequalities come from the declarations. The alignment constraints simply link the array (α) to the template (θ) variables, following the affine alignment subscript expressions. The distribution constraints introduce the block variables (δ), and the cyclic distribution on the second dimension re-

$$
\begin{array}{rl}
\texttt{A}(\alpha_1,\alpha_2) & 1 \le \alpha_1 \le 30, \quad 1 \le \alpha_2 \le 30 \\
\texttt{T}(\theta_1,\theta_2) & 1 \le \theta_1 \le 68, \quad 1 \le \theta_2 \le 30 \\
\texttt{P}(\psi_1,\psi_2) & 1 \le \psi_1 \le 4, \quad 1 \le \psi_2 \le 2 \\
\textit{region} & \left\{ \begin{array}{l} \alpha_1 + \alpha_2 \le 1 + m, \\ 3 \le \alpha_1, \quad 3 \le \alpha_2 \end{array} \right. \\
\texttt{align} & \theta_1 = 2\alpha_1, \quad \theta_2 = \alpha_2 \\
\texttt{distribute} & \left\{ \begin{array}{l} \theta_1 = 17\psi_1 + \delta_1 - 16, \\ 0 \le \delta_1 \le 16 \\ \theta_2 = 10\gamma_2 + 5\psi_2 + \delta_2 - 4, \\ 0 \le \delta_2 \le 4 \end{array} \right. \\
\texttt{A}'(\alpha'_1,\alpha'_2) & \left\{ \begin{array}{l} 1 \le \alpha'_1 \le 9, \quad 1 \le \alpha'_2 \le 15 \\ 2\alpha'_1 = \delta_1 - \eta_1 + 2, \quad 0 \le \eta_1 \le 1 \\ \alpha'_2 = 5\gamma_2 + \delta_2 + 1 \end{array} \right.
\end{array}
$$

Figure 5: Linear constraints for `triangle`

$$
\begin{array}{rll}
\mathcal{C} & (m) & 5 \le m \\
\mathcal{P} & \left\{ \begin{array}{ll} (\psi_1) & 1 \le \psi_1 \le 4, \\ & 17\psi_1 - 2m \le 12 \\ (\psi_2) & 1 \le \psi_2 \le 2 \end{array} \right. & \\
\mathcal{E} & \left\{ \begin{array}{ll} (\gamma_2) & 1 \le 2\gamma_2 + \psi_2 \le 6 \\ (\alpha_2) & 0 \le 10\gamma_2 + 5\psi_2 - \alpha_2 \le 4, \\ & 3 \le \alpha_2 \\ (\alpha_1) & 0 \le 17\psi_1 - 2\alpha_1 \le 16, \\ & 3 \le \alpha_1 \le 30, \\ & \alpha_2 - m + \alpha_1 \le 1 \\ (\alpha'_1) & 17 \le 2\alpha'_1 - 2\alpha_1 + 17\psi_1 \le 18 \\ (\alpha'_2) & \alpha'_2 = 5 + \alpha_2 - 5\psi_2 - 5\gamma_2 \end{array} \right. &
\end{array}
$$

Figure 6: Scannable polyhedra for `triangle`

quires a cycle variable (γ) which counts the cycles over the processors for a given template cell. Each distributed dimension is described by an equation that links the template and the processor dimensions together with the added variables. The local addressing scheme for the array is also translated into a set of linear constraints on α' variables. It enables the direct enumeration of the elements to exchange without expensive address computations at runtime. However what is really needed is the ability to switch from global to local addresses, so any other addressing scheme would fit into this technique. Moreover other addressing schemes [33] may be translated into a linear framework.

1.3 Resolution

The integer solutions to the previous set of equations must be enumerated to generate the communications. Any polyhedron scanning technique [19, 3, 5, 39, 17, 27, 13, 30, 28] can be used. The key issue is the control of the enumeration order to generate tight bounds and to reduce the control overhead. Here the word polyhedron denotes a set of constraints that defines a subspace of integer points. Different sets of constraints may define the same subspace, and some allow loop nests to be generated to scan the points. A *scannable* polyhedron for a given list of variables is such that its constraints are ordered in a triangular way which suits loop bound generation. One loop bound level must only depend on the outer levels, hence the triangular structure.

The technique used in the implementation is algorithm *row_echelon* [5]. It is a two stage algorithm that takes as input a polyhedron and a list of variables, and generates a scannable polyhedron on these variables, the others being considered as parameters. The first stage builds the scannable polyhedron through successive projections, and the second stage improves the quality of the solution by removing redundant constraints while preserving the triangular structure. This algorithm generates a remainder polyhedron by separating the constraints that do not involve the scanning variables.

For the running example, this algorithm is applied twice. The first step addresses the actual element enumeration for a given processor. The processor identity is considered as a fixed parameter to enumerate the elements ($\mathcal{E}$ with γ_2, α and α' variables). The second step is applied on the remainder polyhedron of the first. It addresses the processor enumeration ($\mathcal{P}$ with ψ variables), to characterize the nodes that have a contribution in the I/O statement. The final remainder polyhedron ($\mathcal{C}$) is a condition to detect empty I/Os. The resulting polyhedra are shown in Figure 6: each line gives the variable and the linear constraints that define its lower and upper bounds. The algorithm insures that each variable has a lower and upper bounds which only depend on the outer variables and the parameters. Unnecessary variables (variables that are not needed to generate the communications) are eliminated when possible to simplify the loop nest and speed up the execution. θ, δ and η variables are removed, but γ_2 cannot. The order of the variables to enumerate the elements is an important issue which is discussed in Section 3.1.

2 Code generation

In the previous section, it has been shown how to formalize an HPF I/O statement into a linear framework, and how to solve the problem. The resolution is based on standard polyhedron techniques. From a set of equalities and inequalities, it allows to build scannable polyhedra that suit loop bound generation. The method presented on the running example is very general, and any HPF I/O statement may be formalized and solved as described in Section 1. The only point which is not yet addressed is the replication. In this section, the polyhedra generated in the very general case (replication included) are presented. Then the different code generation issues are discussed. First the macro code to handle the different cases is outlined. Second the particular problems linked to the actual generation of the communications are addressed, that is how data is to be collected to or to updated from the host.

	READ effect (print)	WRITE effect (read)
shared	host: I/O	host: I/O broadcast
distributed	collect host: I/O	if MAY collect host: I/O update

Figure 7: Generated code

2.1 General case polyhedra

In the general case, replication must be handled. Since several processors share the same view of the data, the same message will be sent on updates. For collects, only one processor will send the needed data. Thus four polyhedra are generated which define constraints on the variables between parentheses as described:

condition $\mathcal{C}$ (parameters): to guard empty I/Os.

primary owner $\mathcal{P}_1$ (ψ's): one of the owner processors for the array, which is arbitrarily chosen on the replication dimensions. The HPF replications occur regularly on specific dimensions of the processor grid. One processor is chosen by fixing the ψ variable on these dimensions. This subset describes the peculiar processors that will send data on collects, if several are able to do it.

other owners $\mathcal{P}_r$ (also ψ's): for a given primary owner in $\mathcal{P}_1$, all the processors which share the same view of the data. This is achieved by relaxing the previously fixed dimensions. This polyhedron synthetises the replication information. If there is no replication, then $\mathcal{P}_r = I$, thus the simpler $\mathcal{P}$ notation used in Figure 6 for $\mathcal{P}_1$. This polyhedron is used to send the message to all the processors which wait for the same data.

elements $\mathcal{E}$ (scanners): a processor identity (ψ) being considered as a parameter, it allows to enumerate the array elements of the I/O statement that are on this processor. The scanning variables are the variables introduced by the formalization, except the parameters and the ψ's.

2.2 Generated code

The target programming model is a host-node architecture. The host processor runs a specific code and has to perform the I/O, and the nodes execute the computations. However the technique is not restricted to such a model, and the host processor could be one of the nodes with little adaptation. The generated codes for the distributed and shared arrays are shown in Figure 7. For the shared array case, it is assumed that each processor (host included) has its own copy of the array, so no communication is needed when the array is read. When it is written, the message is sent to all nodes: the accessed elements defined by the I/O statement are broadcasted.

HOST:
if ($\sigma \in \mathcal{C}$) – *non-empty I/O*
  – *get and unpack the messages*
  for $p_1 \in \mathcal{P}_1$
    receive from p_1
    – *enumerate the received elements*
    for $e \in \mathcal{E}(p_1)$
      unpack A(global(e))

SPMD node: – *p is my id*
– *if non-empty I/O and I am in $\mathcal{P}_1$*
if ($\sigma \in \mathcal{C} \bigwedge p \in \mathcal{P}_1$)
  – *enumerate the elements to pack*
  for $e \in \mathcal{E}(p)$
    pack A'(local(e))
  send to host

Figure 8: Distributed collect

For the distributed array case, the whole array is allocated on the host, so the addressing on the host is the same as in the original code. However this copy is not kept coherent with the values on the nodes. It is used just as a temporary space to hold I/O data. The only issue is to move the needed values from the nodes to the host when the array is read, and to update the nodes from the host when the array is written. If the region's approximation is MAY and the array is defined within the statement (distributed row and WRITE effect column in Figure 7), a collect is performed prior to the I/O and the expected update. The rationale is that some of the described elements *may* not be defined by the statement, while the update generates a full copy of that region. This added collect insures that the elements that are not defined by the I/O are updated with their original value.

2.3 Scanning code

The next issue is the generation of the collects and updates, and how the polyhedra are used. For the host-node model, two codes are generated: one for the host, and one SPMD code for the nodes, parameterized by the processor identity. The two codes use the processor polyhedron ($\mathcal{P}$) differently. While the host scans all the nodes described by the polyhedron, the nodes check if they belong to the described set to decide whether they have to communicate with the host. On the host side, the polyhedron is used to enumerate the remote nodes, while on the node side it enables an SPMD code.

The distributed collect code is shown in Figure 8. First, each part checks whether an empty I/O is going to occur. If not, the host enumerates one of the owner of the data to be communicated through the $\mathcal{P}_1$ polyhedron. The messages are packed and sent from the contributing nodes, and symetrically received and unpacked on the host. The local and global functions

HOST:
if ($\sigma \in \mathcal{C}$) – *non-empty I/O*
  – *for each primary owner*
  for $p_1 \in \mathcal{P}_1$
    – *enumerate the elements to pack*
    for $e \in \mathcal{E}(p_1)$
      pack A(global(e))
    – *send the buffer to all owners*
    for $p \in \mathcal{P}_r(p_1)$
      send to p

SPMD node: – *p is my id*
– *if non-empty I/O and I am an owner*
if ($\sigma \in \mathcal{C} \bigwedge p \in \mathcal{P}_1 \times \mathcal{P}_r$)
  receive from host
  – *enumerate the elements to unpack*
  for $e \in \mathcal{E}(p)$
    unpack A$'$(local(e))

Figure 9: Distributed update

used model the addressing scheme for a given point of the polyhedron which represents an array element. For the running example, it is a mere projection since both global (α) and local (α') indices are directly generated by the enumeration process ($\mathcal{E}$). The distributed update code is shown in Figure 9. The replication polyhedron $\mathcal{P}_r$ is used to send a copy of the same message to all the processors that share the same view of the array.

The correctness of the scheme requires that as many messages are sent as received, and that these messages are packed and unpacked in the same order. The balance of messages comes from the complementary conditions on the host and the nodes: the host scans the very node set that is used by the nodes to decide whether to communicate or not. The packing/unpacking synchronization is ensured since the same scanning loop ($\mathcal{E}$) is generated to enumerate the required elements on both sides. The fact that for the distributed update case the messages are packed for the primary owner and unpacked on all the owners is not a problem: the area to be enumerated is the same, so it cannot depend on the ψ variables of the replication dimensions. Thus the enumeration order is the same.

This technique is implemented in `hpfc`, a prototype HPF compiler [16, 15]. An excerpt of the automatically generated code for the running example is shown in Figure 10. The integer division with a positive remainder is used. The array declaration is statically reduced on the nodes. The SPMD code is parameterized by the processor identity (ψ_1, ψ_2) which is instanciated differently on each node. The inner loop nest to enumerate the elements is the same in both codes, as needed to ensure the packing/unpacking synchronization.

3 Extensions

The previous sections address the compilation of HPF I/O communications for a host-node model. From

```
// HOST
ARRAY A(30,30)
IF (m ≥ 5) THEN
  DO ψ1 = 1, min((12+2m)/17, 4)
  DO ψ2 = 1, 2
    CALL pvmfrecv(P(ψ1, ψ2)...)
    DO γ2 = (2-ψ2)/2, (6-ψ2)/2
    DO α2 = max(-4 + 10γ2 + 5ψ2, 3),
            5ψ2 + 10γ2
    DO α1 = max((17ψ1-15)/2, 3),
            min(17ψ1/2, 30, -α2 + m + 1)
    DO α1' = (18-17ψ1+2α1)/2, (18-17ψ1+2α1)/2
      α2' = α2 - 5ψ2 + 5 - 5γ2
      CALL pvmfunpack(A(α1, α2)...)
  ENDDOs
ENDIF

// SPMD node, (ψ1, ψ2) is my id in P
ARRAY A'(9,15)
IF (m ≥ 5 AND 17ψ1 ≤ 2m + 12) THEN
  CALL pvmfinitsend(...)
  DO γ2 = (2-ψ2)/2, (6-ψ2)/2
  DO α2 = max(-4 + 10γ2 + 5ψ2, 3),
          5ψ2 + 10γ2
  DO α1 = max((17ψ1-15)/2, 3),
          min(17ψ1/2, 30, -α2 + m + 1)
  DO α1' = (18-17ψ1+2α1)/2, (18-17ψ1+2α1)/2
    α2' = α2 - 5ψ2 + 5 - 5γ2
    CALL pvmfpack(A'(α1', α2')...)
  ENDDOs
  CALL pvmfsend(host, ...)
ENDIF
```

Figure 10: Collect host/SPMD node codes

an I/O statement, it has been shown how to formalize and to solve the problem, then how to generate code from this mathematical solution. In this section, possible improvements are discussed first, then the technique is extended to handle parallel I/O communications.

3.1 Improvements

The implementation of the technique in `hpfc` already includes many basic optimizations. Let us look at the automatically generated code again. Non necessary variables (δ, η, θ) are removed from the original system through exact integer projections when legal, to reduce the polyhedron dimension. The condition checked by the node to decide whether it has to communicate is simplified: for instance constraints coming from the declarations on ψ's do not need to be checked. Moreover the variables used to scan the accessed elements are ordered so that the accesses are contiguous if possible in the memory, taking into account the row major allocation scheme of Fortran. Thus they are ordered by dimension first, then the cycle before the array variables...However the generated code may still be improved.

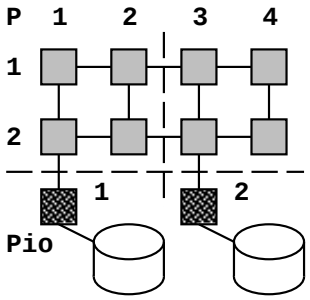

Figure 11: 2 I/O processors

$$\begin{array}{ll} \texttt{Pio}(\psi_{\text{io}}) & 1 \le \psi_{\text{io}} \le 2 \\ \texttt{P}(\psi_1,\psi_2) & \left\{ \begin{array}{l} \psi_1 = 2\psi_{\text{io}} - 1 + \delta_{\text{io}}, \\ 0 \le \delta_{\text{io}} \le 1 \end{array} \right. \end{array}$$

Figure 12: `Pio` to `P`

First, Hermite transformations could have been systematically applied to minimize the polyhedron dimension, as suggested in [4]. The simplification scheme used in the implementation may keep unnecessary variables in some cases.

Second, the generated code would benefit from many standard optimizations such as strength reduction, invariant code motion, dag detection and constant propagation [1], which are performed by any classical compiler at no cost for `hpfc`. For instance, in Figure 10, a compiler may notice that α'_1 loop lower and upper bounds are equal, that α'_2 computation may be moved outside of the α_1 loop, or even that α'_1 and α'_2 computations are not necessary on the host.

Third, improving the analyses used by the technique would help enhance the generated code. For instance, the current implementation of the region analysis only computes a polyhedron around the accessed elements. Thus, it fails to detect accesses with a stride, and useless elements may be transmitted. Equalities may be added to the analysis to catch the underlying lattice which is lost in such cases. Another issue is to detect more `MUST` regions in general. Indeed, if the region approximation is `MAY` while all elements are accessed, then unuseful communications are generated when distributed arrays are defined.

3.2 Parallel I/O

The technique was presented for a host/node architecture. It can be extended to parallel I/O. The basic idea of the method is to use standard polyhedron scanning techniques to generate vectorized messages. A linear description of the problem is needed. The reciprocal would be that if given a linear description of a parallel I/O problem, an efficient vectorized message passing code can be generated. So it is. Let us consider the running example again. Let us now assume that instead of a host/node architecture, we have 2 I/O processors which are regularly attached to the processor grid as depicted in Figure 11. This kind of arrangement is representative of I/O capabilities of some real world machines. However, there is no need for I/O nodes to be physically mapped as depicted. This regular mapping may only be a virtual one [18] which allows to attach groups of processors to I/O nodes.

$$\begin{array}{lll} \mathcal{C} & (m) & 5 \le m \\ \mathcal{P}_{\text{io}} & \left\{ \begin{array}{ll} (\psi_{\text{io}}) & 1 \le \psi_{\text{io}} \le 2, \\ & 17\psi_{\text{io}} - m \le 16 \end{array} \right. & \\ \mathcal{P} & \left\{ \begin{array}{ll} (\psi_1) & 2\psi_{\text{io}} - 1 \le \psi_1 \le 2\psi_{\text{io}}, \\ & 17\psi_1 - 2m \le 12 \\ (\psi_2) & 1 \le \psi_2 \le 2 \end{array} \right. & \\ \mathcal{E} & \ldots & \end{array}$$

Figure 13: Parallel I/O polyhedra

```
// SPMD I/O node, ψio is my id in Pio
IF (m ≥ 5 AND 17ψio ≤ m + 16) THEN
  DO ψ1 = 2ψio − 1, min((12+2m)/17, 2ψio)
  DO ψ2 = 1, 2
    // receive from P(ψ1, ψ2) and unpack
    ...
ENDIF

// SPMD node, (ψ1, ψ2) is my id in P
// ψio is my attached I/O node
IF (m ≥ 5 AND 17ψ1 ≤ 2m + 12) THEN
  // pack and send to Pio(ψio)
  ...
ENDIF
```

Figure 14: Parallel I/O collect

Figure 12 suggests a possible linear formulation of the link between the I/O nodes and their attached processors. ψ_{io} ranges over the I/O nodes. An auxiliary dummy variable (δ_{io}) is used to scan the corresponding processors. This description looks like the one presented in Figure 5. The same scheme to enumerate the elements of interest (the contributing nodes, namely polyhedron $\mathcal{P}$) can be applied, with some external parameters (ψ_{io}, m) to be instantiated at runtime on the different nodes: the algorithm to build the scannable polyhedra is applied three times instead of two. The added level allows the I/O nodes ($\mathcal{P}_{\text{io}}$) to scan their processors ($\mathcal{P}$), as the processors are allowed to scan their elements ($\mathcal{E}$).

The resulting scannable polyhedra are shown in Figure 13. They allow to generate the 2 SPMD codes of Figure 14. One is for the I/O nodes, and the other for the computation nodes. The inner loops are skipped, since they are the same as in Figure 10. The I/O nodes scan their attached nodes to collect the needed messages, and the computation nodes communicate with their attached I/O node instead of the host. To com-

```
chpf$ processors P(4,2)
chpf$ io_processors Pio(2)
chpf$ io_distribute P(block,*) onto Pio
```

Figure 15: Suggested syntax for parallel I/O

plete the code generation, a new addressing scheme should be chosen on the I/O nodes, and a parallel file system should be used to respect the distributed data coherency.

This compilation scheme suggests new HPF directives to help compile parallel I/O statements. Indeed, what is needed is a linear link between the I/O nodes and the processing nodes. Moreover every computing node must be given a corresponding I/O node. The easiest way to achieve both linearity and total function is to rely on a distribution-like declaration, that is to distribute processors onto I/O nodes. A possible syntax is shown in Figure 15. It advises the compiler about the distribution of the I/Os onto the I/O nodes of the machine for data-parallel I/O. It is in the spirit of the hint approach that was investigated by the HPF Forum [21], that is to give some information to the compiler. However the record oriented file organization defined by the Fortran standard, which may also be mapped onto I/O nodes, is not directly addressed by this work, but such mappings may also be translated into linear constraints and compiled [4].

3.3 Related work

Other teams investigate the distributed memory multicomputers I/O issues, both on the language and runtime support point of view. Most works focus on the development of runtime libraries to handle parallel I/O in a convenient way for users. The suggested solutions focus more on general issues than on specific techniques to handle the required communications efficiently. They are designed for the SPMD programming model, and the allowed data distribution semantics is reduced with respect to the extended data mapping available in HPF.

In [9], a two-phase access strategy is advocated to handle parallel I/O efficiently. One phase performs the I/O, and the other redistribute the data as expected by the application. The technique presented in this paper would help the compilation of the communications involved by such a redistribution (between the I/O nodes and the computation nodes). This whatever the HPF mapping, as part of a dataparallel program to be compiled to a MIMD architecture.

In [23], the `PETSc/Chameleon` package is presented. It emphasizes portability and parallel I/O abstraction. [18] suggests to decluster explicitely the files, thus defining logical partitions to be dealt with separately. Their approach is investigated in the context of the Vesta file system. An interface is provided to help the user to perform parallel I/O. In [34], the PIOUS system is described. It is in the spirit of the client-server paradigm, and a database-like protocol insures the coherency of the concurrent accesses.

Moreover polyhedron scanning techniques have proven to be efficient and realistic methods for compilers to deal with general code transformations [39, 27] as well as distributed code generation [2, 29, 4]. In [2], a dataflow analysis is used to determine the communication sets. These sets are presented in a linear framework, which includes more parametrization and overlaps. The data mapping onto the processors is a superset of the HPF mapping. The local memory allocation scheme is very simplistic (no address translations) and cyclic distributions are handled through processor virtualization. [29] presents similar techniques in the context of the Pandore project, for commutative loop nests. The mapping semantics is a subset of the HPF mapping semantics. The Pandore local memory allocation is based on a page-like technique, managed by the compiler [30]. Both teh local addressing scheme and the cyclic distributions are integrated in [4] to compile HPF. Moreover equalities are used to improve the scanning loop nests, and temporary allocation issues are discussed.

Conclusion

Experiments were performed on a network of workstations and on a CM5 with the PVM3-based generated code for the host-node model. The performances are as good as what could have been expected on such systems. The control overhead to enumerate the required elements is not too high, especially for simple distributions. For network of workstations, the host should be the file server, otherwise the data must be transfered twice on the network.

We have presented a technique based on polyhedron scanning methods to compile parallel I/O communications for distributed memory multicomputers. This technique for the host-node architecture is implemented within `hpfc`, a prototype HPF compiler developed at CRI. This compiler is part of the PIPS automatic parallelizer [26]. From Fortran 77 code and static HPF mapping directives, it generates portable PVM 3-based code. It implements several optimizations such as message vectorization and overlap analysis on top of a runtime resolution compilation. Future work includes experiments, the implementation of advanced optimizations [4] and tests on real world codes.

Acknowledgements

I am thankfull to Corinne Ancourt, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, Kélita Le Nénaon, Alexis Platonoff and Xavier Redon for their comments and suggestions.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1993.

[3] C. Ancourt. *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales.* PhD thesis, Université Paris VI, Mar. 1991.

[4] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A Linear Algebra Framework for Static HPF Code Distribution. In *Workshop on Compilers for Parallel Computers, Delft*, Dec. 1993. Also available as TR EMP A/250/CRI.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Symposium on Principles and Practice of Parallel Programming*, Apr. 1991.

[6] B. Apvrille. Calcul de régions de tableaux exactes. In *Rencontres Francophones du Parallélisme*, pages 65–68, June 1994.

[7] B. Apvrille-Creusillet. Régions exactes et privatisation de tableaux. Master's thesis, Université Paris VI, Sept. 1994.

[8] F. Bodin, L. Kervella, and T. Priol. Fortran-S: A Fortran Interface for Shared Virtual Memory Architectures. In *Supercomputing*, Nov. 1993.

[9] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Supercomputing*, pages 452–461, Nov. 1993.

[10] R. R. Bordawekar, A. N. Choudhary, and J. M. del Rosario. An experimental performance evaluation of touchstone delta concurrent file system. In *ACM International Conference on Supercomputing*, pages 367–376, July 1993.

[11] T. Brandes. Efficient data parallel programming without explicit message passing for distributed memory multiprocessors. Internal Report AHR-92 4, High Performance Computing Center, German National Research Institute for Computer Science, Sept. 1992.

[12] T. Brandes. Evaluation of high performance fortran on some real applications. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 417–422, Apr. 1994.

[13] Z. Chamski. Fast and efficient generation of loop bounds. Research Report 2095, INRIA, Oct. 1993.

[14] M. Chen and J. Cowie. Prototyping Fortran-90 compilers for Massively Parallel Machines. *ACM SIGPLAN Notices*, pages 94–105, June 1992.

[15] F. Coelho. Étude de la Compilation du *high performance fortran.* Master's thesis, Université Paris VI, Sept. 1993. Rapport de DEA Systèmes Informatiques. TR EMP E/178/CRI.

[16] F. Coelho. Experiments with HPF compilation for a network of workstations. In *High-Performance Computing and Networking, Springer-Verlag LNCS 797*, pages 423–428, Apr. 1994. Also available as TR EMP A/257/CRI.

[17] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from Systems of Affine Constraints. LIP RR93 15, ENS-Lyon, May 1993.

[18] P. F. Corbett, D. G. Feitelson, J.-P. Prost, and S. Johnson Baylor. Parallel Access to Files in the Vesta File System. In *Supercomputing*, pages 472–481, Nov. 1993.

[19] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

[20] S. A. Fineberg. Implementing the NHT-1 Application I/O Benchmark. *ACM SIGARCH Computer Architecture Newsletter*, 21(5):23–30, Dec. 1993.

[21] H. P. F. Forum. *High Performance Fortran Journal of Development.* Rice University, Houston, Texas, May 1993.

[22] H. P. F. Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, May 1993. *Version 1.0.*

[23] N. Galbreath, W. Gropp, and D. Levine. Application-Driven Parallel I/O. In *Supercomputing*, pages 462–471, Nov. 1993.

[24] A. Geist, A. Beguelin, J. Dongarra, J. Weicheng, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.

[25] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD Distributed-Memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[26] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing*, June 1991.

[27] W. Kelly and W. Pugh. A framework for unifying reordering transformations. UMIACS-TR-93 134, Institute for Advanced Computer Studies, University of Maryland, Apr. 1993.

[28] M. Le Fur. Scanning Parametrized Polyhedron using Fourier-Motzkin Elimination. Publication interne 858, IRISA, Sept. 1994.

[29] M. Le Fur, J.-L. Pazat, and F. André. Commutative loop nests distribution. In *Workshop on Compilers for Parallel Computers, Delft*, pages 345–350, Dec. 1993. extended version in IRISA TR 757, Sept. 93.

[30] H. Le Verge, V. Van Dongen, and D. K. Wilde. Loop nest synthesis unsing the polyhedral library. Publication interne 830, IRISA, May 1994.

[31] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, Sept. 1986.

[32] Z. Lin and S. Zhou. Parallelizing I/O Intensive Applications for a Workstation Cluster: a Case Study. *ACM SIGARCH Computer Architecture Newsletter*, 21(5):15–22, Dec. 1993.

[33] Y. Mahéo and J.-L. Pazat. Distributed array management for HPF compilers. Publication interne 787, IRISA, Dec. 1993.

[34] S. A. Moyer and V. S. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Scalable High Performance Computing Conference*, pages 71–78, 1994.

[35] B. K. Pasquale and G. C. Polyzos. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Supercomputing*, pages 388–397, Nov. 1993.

[36] R. Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures.* PhD thesis, Université Paris VI, 1984.

[37] R. Triolet, P. Feautrier, and F. Irigoin. Direct parallelization of call statements. In *Proceedings of the ACM Symposium on Compiler Construction*, 1986.

[38] C.-W. Tseng. *An Optimising Fortran D Compiler for MIMD Distributed Memory Machines.* PhD thesis, Rice University, Houston, Texas, Jan. 1993.

[39] M. J. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[40] H. Zima and B. M. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, Feb. 1993.