Automatic Code Distribution

Corinne Ancourt[*]
François Irigoin[†]

Centre de Recherche en Informatique
Ecole Nationale Supérieure des Mines de Paris
35, rue Saint Honoré
77300 FONTAINEBLEAU
FRANCE

Abstract

Distributed memory machines have been around for quite a long time, but, in spite of their excellent performance/price ratio, they are not widely used for scientific applications. The main reason for that setback is that no compiler is available for these applications, usually written in Fortran, and for this machine architecture. We developped in PUMA Esprit Project a prototype compiler, taking as input sequential Fortran code, and generating parallel distributed processes for T9000/C104 based machines. To the best of our knowledge, it is the first of its kind and it was demonstrated at ESPRIT Technical Week 1991. Its main compilation steps (control distribution, data distribution and data movement generation) are described as well as rationale for this approach.

1 Introduction

It is now well known that writing scientific parallel applications for distributed memory machines is not as easy as was expected a few years ago. Large objects like matrices have to be broken into pieces and each piece as to be mapped onto local memories. *Receive* statements have to be added in each process to retrieve data elements stored in different processes and conversely *send* statement have to be introduced in processes which can access data elements required by other processes. Finally all processes have to be synchronized to avoid deadlocks and wrong data exchanges. Writing and debugging distributed processes is much more demanding than declaring a loop parallel in a shared memory machine program.

Our goal is to automatize some of these tedious chores using an *Emulated Shared Memory* scheme and the universal message passing capability provided by the INMOS T9000 processor and C104 hardware router. One half of the processors (`CE0, CE1, CE2 and CE3` - figure 4) perform computations and the other half (`MC0, MC1, MC2 and MC3`) emulate memory banks providing the compiler with a better understood target machine, a multiprocessor with a fast local memory and a slow shared memory. A fast context switching time makes also possible to overlap computations and communications.

Many problems have to be solved to convert code using a unique memory space into parallel distributed code. Objects are mapped in the emulated shared memory, using a fixed scheme, but they cannot be used before they are copied in the memory of one of the compute engines. And, when new objects are computed, they are not available before they are copied back into the emulated shared memory.

For efficiency reason, object elements are not moved each time they are referenced. Some control blocking is performed to decrease scheduling, synchronization and communication overheads. Many parameters have to be taken into account to decide a blocking: dependencies between references, which may make some block-

[*] `<ancourt@cri.ensmp.fr>`
[†] `<irigoin@cri.ensmp.fr>`

ings illegal, context switching time, amount of elementary computations, packet size for messages, emulated memory structure, amount of communication, data re-use,...

It is also necessary to study the reference patterns to decide how many copies of one object have to be made. If two references in the input code conflict, they have to be transformed into references to the same object copy. As a case in point, see the convolution example where six references, `A(I,J)` (twice), `A(I-1,J)`, `A(I+1,J)`, `A(I,J-1)`, `A(I,J+1)`, are remapped to the same copy. Else, if two references are independent like `M(I,J)` and `M(J,I)` in the matrix transpose example, they have to be transformed into two references to two different copies. If this was not recognized, a matrix transpose would require that each processor contains the whole matrix.

```
      DO 100 I = 1, N
        DO 100 J = 1, N
          NA(I,J) = 0.2 * (A(I-1,J)
     &               +A(I,J)+A(I+1,J)
     &               +A(I,J-1)+A(I,J+1)
 100  CONTINUE
```

Figure 1: Convolution code

```
      DO 100 I = 1, N
        DO 100 J = I+1, N
          T = M(I,J)
          M(I,J) = M(J,I)
          M(J,I) = T
 100  CONTINUE
```

Figure 2: Matrix transposition

New control structures have to be generated to enumerate all blocks and all iterations in each block, and data movements are inserted to load and store the local memory without destroying the shared memory consistency. Integer programming techniques had to be developped to statically generate consistency preserving memory updates. This makes the technique scalable since a constant run-time overhead is incurred.

Different approaches currently explored for implementing a shared memory on a distributed memory machine are first surveyed in section 2. We then describe our approach and our motivations. Section 4 describes the compilation phases needed to generate distributed code according to our scheme. We present, using matrix transposition as an example, these different compilation steps, often used in current parallelizers, and show their importance when generating efficient codes for distributed memory machines.

2 A Set of Approaches

Different approaches are currently investigated to provide the user with a shared memory programming model, implemented on a physically distributed memory. The first one is language-based. If no efficient solutions based on high-level languages like STRAND or run-time mechanism like LINDA [5] can be found because of the parallelism granularity, the programmer has to specify the data distribution pattern and the parallelism but he does not have to generate processes nor send/receive pairs between processors. Processes are automatically derived from the data distribution using the so-called *owner rule* and a SPMD model [12],[4],[3]: each processor execute instruction producing a piece of data located in its memory. Send/receive pairs are derived from the data distribution and from the instruction reference patterns. A naive compiler generating guards and send/receive everywhere to respect the owner rule and to fetch external data or send requested data produces very inefficient code. Numerous optimizations must be performed. These might work quite well on regular data structures but no commercial compiler exists. The compiler has also to generate efficient global synchronizations necessary for data parallelism. Finally there is a need for a language standardization and a few American research laboratories have cooperated to define Fortran D, D like distributed. In the mean time, ACPC in Vienna has designed another data distribution dialect, Vienna-Fortran and Digital has launched the HPF Forum.

The second kind of approaches is based on the operating system and some hardware support. A virtual shared memory provides the uniform address space. Some software mechanism maintains the memory consistency while hardware supports efficient page movements between processors. At any given time, each processor owns some pages which he can write and share with other processors copies of pages which they all can read . The page size must be large enough to dwarf the transfer and the consistency maintenance overhead and small enough so as not to kill parallelism by introducing artificial conflicts between data located in the same page. Some preliminary results are not good at all, others are encouraging. But some cooperation between the compilers and the operating systems will be necessary to preserve portability and expectations of naive users.

The third approach fully relies on hardware. Complex cache systems are built with scalable mechanism to insure memory consistency for massively parallel machines with a limited amount of communication. Once again, some compiler involvement would be useful to generate instruction for the caches and to decrease the inter-cache communications.

Different solutions are currently explored for mapping a shared memory model on a distributed memory machine. These solutions differ by the requirements put on each of the following levels (see figure 3):

1. programming language;
2. compiler;
3. run-time and library;
4. operating system;
5. hardware.

Broadly speaking, the emphasis is shifted from the language level to the hardware level when going from left to right as is shown by font sizes. The demand on compilers is more or less centered which is logical since they stand in between. Of course, message-passing languages which do not provide a uniform memory space are ignored.

The left part of the figure is used for *owner rule* based schemes[1]. The owner rule states that each instruction can only update local variables. This immediately implies that:

1. data structure have to be partitioned on the processors;
2. memory consistency is preserved since no data is duplicated;
3. non local read references have to be requested from other processors;
4. each processor has to act as a memory server for the other ones;
5. the control partitioning is derived from the data partitioning.

The data partitioning can be explicit (Explicit Data Partitioning, EDP) if the programming language provides support for it or implicit (Implicit Data Partitioning, IDP) if the compiler uses a fixed scheme. To the best of our knowledge, the fixed scheme has never been suggested and implemented. Explicit partitioning can be either static (SEDP) or dynamic (DEDP). The latter, DEDP, is (supposedly) better for efficiency but requires a better compiler. For the time being, experiments have been reported only for Static Explicit Data Partitioning which require the lowest effort in compiler and hardware technology because a lot is left in the user hands. Automatic data partitioning has not yet been successfully implemented because there is usually no good unique data partitioning for a complete application: arrays are used in different ways at different points of a large program.

The right part of figure 3 is dedicated to control-based partitioning, which assumes that some kind of logical global memory is provided. This means that parallel code can be generated by distributing loop iteration over the elementary processors. Input data are automatically moved on the relevant processor before it starts executing a given iteration and results are sooner or later moved somewhere as input to some other computation. The user does not have to worry about data distribution.

As explained before, data movements can be executed at different architectural levels. Going from right to left, the cache approach is the most demanding one at the hardware level, since cache consistency must be hardwired to provide a realistic shared memory (SM) programming model[2].

[1] To provide more flexibility Fortran D, although it is mainly based on the owner rule, has a feature to let the user specify on which processor iterations should be executed.

[2] Experiments, like IBM RP3 project[13][14], were

Unique Logical Memory Space						
Owner rule (Data Partitioning)			Shared Memory (Control Partitioning)			
Explicit Data Partitioning		Implicit Data Partitioning	Run-Time Emulation		Operating System	Cache
Dynamic	Static		Specific	Generic		
DEDP	SEDP	IDP	CESM	GESM	VSM	SM
languages	languages	language	language	language	language	language
compiler	compiler	compiler	compiler	compiler	compiler	compiler
hardware	hardware	hardware	hardware	hardware	hardware	hardware

Figure 3: A Set of Approaches

Consistency can also be maintained at the operating system level if the cache lines are large enough to be called pages and be part of a virtual memory. Conflicts are then assumed to be infrequent enough to let software handle them. Such schemes are known as *Virtual Shared Memory* (VSM) systems. They are based on well-known hardware for page translation and a very fast network to exchange pages between processors. Work in this area was initiated by K. Li [11] and a company, KSR, is marketing such a machine. Efficiency is not always sure because the full consistency protocols are quite heavy and because artificial conflicts appear between independent words in the same page. This situation can be improved by involving the user or the compiler to allocate objects in different kinds of pages for which different kinds of consistency are insured, and to distribute the code among processors so as to decrease page conflicts.

Finally, consistency can be maintained by a piece of software emulating a shared memory (Emulated Shared Memory, ESM). This can be done dynamically, somehow as in LINDA, at the expense of efficiency. Each memory request is checked and a global memory request is issued. This can sometime be done statically. The program is analyzed and references to global objects are put together to decrease global memory request overhead. The emulated shared memory server can be generic and accept *fetch* requests for any address and any length (GESM) or be compiled from the application so as to avoid a multiple send/receive protocol to serve emulated memory requests. The specific emulated memory server (also called Compiled Emulated Shared Memory, CESM) executes the control part of the application and *guesses* which shared memory services are needed. Requirements for the hardware and the input language are quite low, but the final efficiency will very much depend on the compiler ability to group memory requests so as to lower overheads and to cache data locally so as to decrease the number of multiple transfers. Scalability is insured because consistency issues are analyzed statically by the compiler.

These different solutions can be compared by taking into account the following criteria. What kind of applications are well adapted to the solution? How convenient is the programming model? How easy is it for the user to improve efficiency? Is he still in control or is the physical machine model invisible? What are the problems for the compiler? What are the constraints for the hardware? These criteria are qualitative and it is difficult to evaluate them for each solution without experimenting. As of today, they do not let us define *the* best solution.

Our approach stands in the midst of this solution spectrum. We show, using simple programs, what the trade-offs are and how our compiling techniques are combined to generate efficient codes for distributed memory machines.

also designed to explore software cache consistency. The compiler has to explicitly invalidate cache lines if an inconsistency might occur. Generating such cache instruction has proved to be inefficient for general programs. A combination of specific hardware and some compiler directives is usually advocated[8]

3 Our Approach

Our approach emulates a shared memory using compiled memory servers (CESM model). It was developped for the T9000 transputer[3] which does not provide support for virtual memory nor for multiprocessor cache consistency. However its virtual channels and its companion intelligent router chip, the C104, provide some support for a global memory emulation, if not for global synchronization. Data and process mappings are thus less of a problem than on T800 based machines or on Intel first-generation hypercubes.

The basic idea is to abandon the owner rule and to replace it by an emulated shared memory at the price of an increase in inter-processor communication load. The potential advantages are:

- implicit data distribution based on control partitioning which means there is no need for static or dynamic explicit data partitioning;
- efficient handling of more general constructs than with the owner rule;
- memory servers do not have to be run on the same processors as computations; this increases the communication bandwidth and decreases the number of context switches on computational processors;
- easier load balancing because any process can be started on any processor;
- parallelism granularity can be easily tuned because it is not implicitly linked to a data distribution; this also may improve load balancing.

Communications and computations should be generated to overlap as much as possible.

The second idea is to use application-specific memory servers which know what the requests are going to be. There is no need for explicit memory requests and the useful global bandwidth is increased because no addresses nor message lengths are transmitted. Care should be taken not to generate static process mapping or some load unbalance may be lost again.

[3] Also known as the H1 project.

We assume that efficient point-to-point communication is available and that the interconnection topology can be ignored thanks to hardware routers.

The main characteristics of our approach are the following ones.

Data distribution is implicit. The objects are uniformly distributed on the emulated shared memory and there is no need for static or dynamic explicit data partitioning. Their distribution is independent from their use.

Task building is based on control partitioning. The dependence graph between program instructions is used to build parallel tasks. Loop transformations like *tiling transformation* are used on nested loops in order to define blocs of loop iterations that can be computed in parallel.

Block sizes are tuned to reduce scheduling and communication startup overheads to less than x % of the execution time and, when it is possible, to make sure that the communication time is lower than the computation time. Execution time and communication time are statically estimated.

The granularity of parallelism can be easily tuned because it is not implicitly linked to a data distribution but only to the control graph of the program; this may improve load balancing too.

Each block is then seen as a logical independent task. Each task is made of three parts: a prologue to read the input data from the emulated shared memory, a computational part and a final part to store the results.

Usually, with massive parallelism, many such tasks have to be assigned to each processor. Obtaining good load balancing should easier than with the *owner rule scheme* because any process can be started on any processor and neither communications nor synchronizations are needed during the computations. However a fixed mapping scheme was implemented to start with.

Communications and computations should be generated to overlap as much as possible. Emulated shared memory latency was to be hidden by pipelining many computational processes on each processor. However it seems that link bandwidth is dominant. Thus a pipeline of depth 3 was found to be sufficient for initial experiments. Each processor should support at the same time three logical task. Task $n-1$ is storing results while task n is computing and task $n+1$ is fetching its inputs.

The memory servers executes the same control code as the computational processes. They send inputs when they become necessary and receive results as soon as they are available. They use direct links to each processor and communicate in parallel with all of them. In the same way, memory request from computational engines are served in parallel by each memory server.

Figure 4 describes this implementation for a network of eight T9000. The four upper transputers (MC 0 to 3) emulate the shared memory, while the other four ones (CE 0 to 3) perform the computations. The virtual channels of T9000 associated to the C104 intelligent router allow to move data in parallel between each memory emulator and each computational processor.

4 Compilation Steps

To automatically generate distributed code the following compiling phases are necessary:

- program analysis and variable declarations;
- generation of parallel tasks (*tiling*);
- generation of data movements.

4.1 Program analysis and variable declarations

The matrix transposition code figure 2 is a naive example that is small enough to graphically understand how the program is transformed. It also shows a limit for *owner rule* based schemes. It is difficult to map iterations of the loop body to a processor owning all updated array elements. `M(I,J)` and `M(J,I)` are most of the time on different processors. The overlap technique defined by Gerndt in [6] would find that the whole matrix `M` itself is the overlap between the two references.

It is also impossible to distribute the loop because `T` is a *private* variable to each iteration (its initial values are set up in each loop body iteration and its final values are not used subsequently). It would have to be expanded and use as much space as the full matrix `M`.

It is important to detect these private variables because they can be allocated in the local memories of the computational processes. They do not have to be fetched from the emulated shared memory as input. They do not have to be stored as results. Scalar variables can be identified as private by PIPS[10]. The dependence graph, more especially the "use-def chains", is used for identifying *private* variables. They are not allocated in the emulated shared memory. They are only declared in the computational part of the code.

Standard dependences between references must be analyzed to find out scheduling constraints between loop iterations. If these constraints define a total order, the loops have to be executed sequentially and it is not possible to perform any kind of partitioning interesting to decrease execution time. Some partioning may still be needed however to meet the local memory size constraint.

"Input dependencies", linking two *uses* of a variable, have also to be taken into account. They let us know how many local copies have to be associated to each variable. In the transposition matrix example the two references `M(I,J)` and `M(J,I)` are independent. Two copies of `M`, characterizing two distinct memory allocations called `L_M_0_0` and `L_M_1_0` see figure 5.2, must be declared. One part of the whole matrix `M` must also be declared in the *Emulated Shared Memory* program. It is called `ES_M` and corresponds to the intersection of `M` and one memory bank.

Note that some of these techniques are used in parallelizers and vectorizers and just re-used to compile applications for distributed memory machines.

4.2 Tiling

Task generation is based on control partitioning. The dependence graph is used to define sets of tasks which can be computed in parallel. Loop transformations, namely *tiling transformations*, are used to define blocks of loop iterations, called *tile*. The iteration space is tiled according to [9].

The matrix transpose is fully parallel and the dependence graph does not impose any constraints on the tiling. Rectangular blocks, parallel to the initial loop directions, are chosen

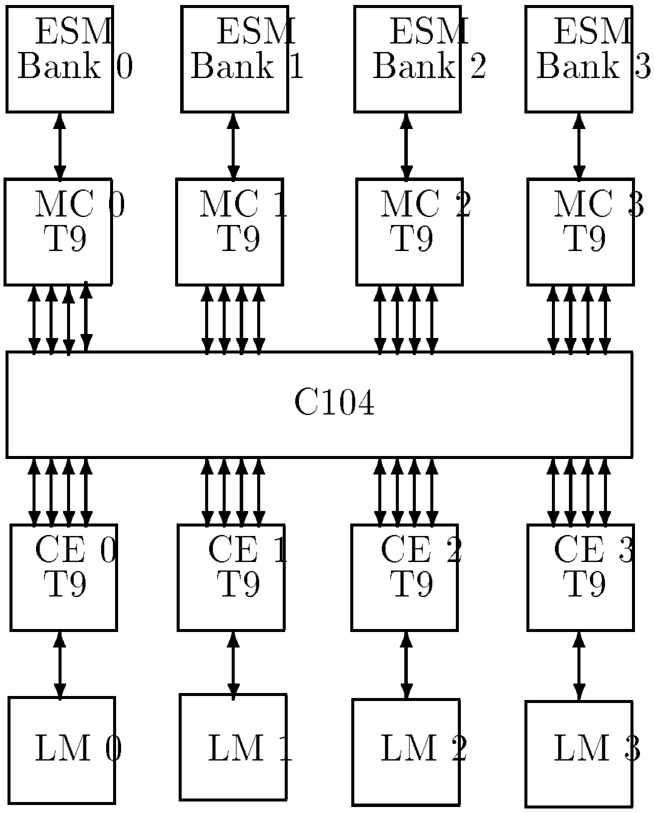

Figure 4: A Four-Processor T9-Based Implementation

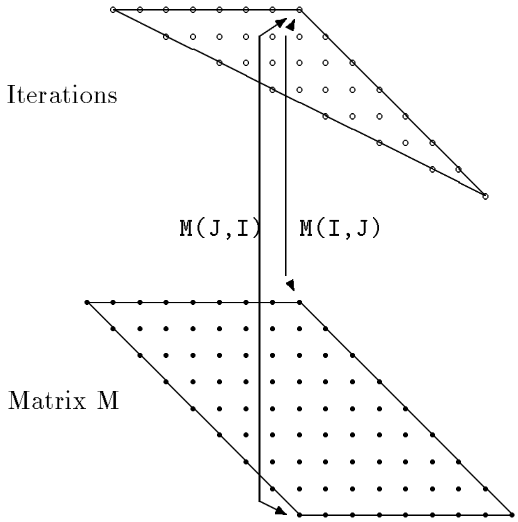

Figure 5: Access pattern for matrix transposition

by default (see figure 6). Their size has to be tuned according to the machine model (local memory size, overheads,...).

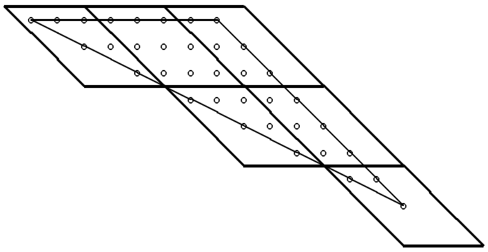

Figure 6: control distribution

4.3 Data Movements

Once the partitionning of iterations in tasks is decided, data movements between emulated shared memory banks and local memories can be computed.

Each *computational processor* only performs computations on local variables. The set of data *used* by a tile must be copied into their local memories before their execution begin. Then the set of locations *defined* during the execution must be copied back from the local memory into the shared memory. Figure 7 represents these two sets for the transposition example with the tiling described in the previous section. Figure 8 represents the location on banks of these two accessed regions. To simplify the picture the number of banks is 3, the number of elements per bank line is 2 and the matrix size is 9, but realistic figures would be much larger.

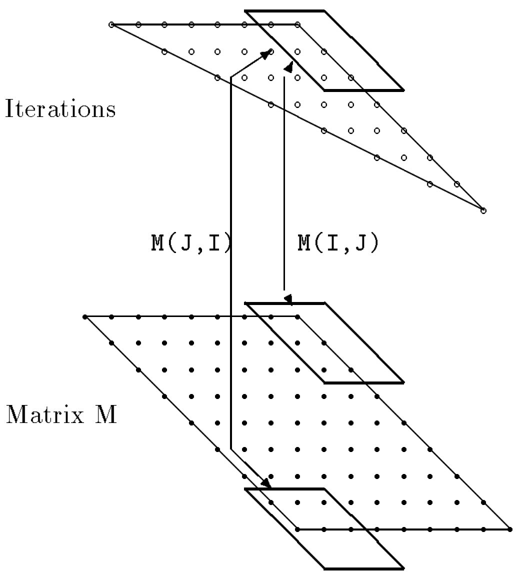

Figure 7: Def and used regions for a tile

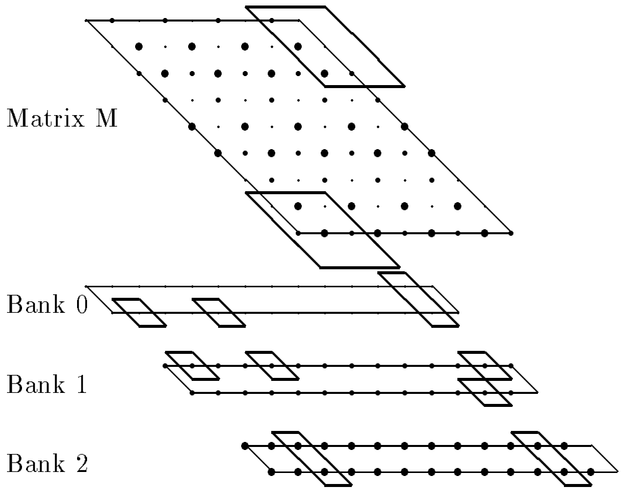

Figure 8: Tile access pattern

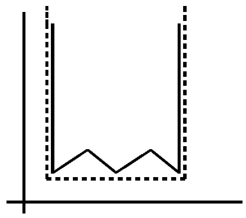

Figure 9: Convex approximation of a *use* set

To generate these movements the sets of locations *used* and *defined* by each task must be characterized. Characterization of use and def locations is hard when arrays are referenced. As a matter of fact, it is the set of all array elements referenced by a set of loops contained in a parallel task that must be characterized. This set is equivalent to a set of integer points and is the major cause of problems. Because these sets are not necessarily convex (see figure 8), data movement codes contain complex loop bound expressions. They would not be easy to write for a programmer.

Algorithms to solve this problem are described in [1]. The availability of many library routines manipulating linear systems, matrix and polyhedra, that had been previously developped in the PIPS project, let us implement them quickly.

Copy-in code

For transfers from shared memory to local memories of *used* data, it is not necessary to copy the exact set of *used* data. We can copy a larger set because these data will only be *used* by local parallel tasks. Thus, no conflicts between different copies can arise.

If the set of used array elements is not a convex polyhedron, a slightly larger set will be transferred: the smallest convex polyhedron including the set of actually used data. In case of several *use* array dependencies only one copy code is generated for all array references and each array element is only transferred once.

Copy-back code
For transfers from local memory to shared memory of *modified* data, it is not correct to copy back one element that has not been modified by the local task. This element might have been modified by another task, and the newer value could be overwritten by the old one in the shared memory. Thus, precise copy back code must be generated. For each *def* array reference one copy back code is generated, unless references are equal up to a translation.

When the data set to transfer is not a convex set, integer divisions are inserted in bound expressions for loops of transfer code in order to convey the non-convexity. A linear test in the inner loop is sometimes needed too. To avoid this test, which increases the control overhead, data movements are sometimes distributed: one transfer code is generated for each convex set contained in the modified data set. Thus data movements remain minimal and each element is copied only once.

5 Generated code

5.1 General structure

The generated code is composed of two subroutines:

1. one subroutine, called `COMPUTE(PROC_ID)`, contains the computational part of the code and receives a (logical) processor number as parameter;

2. one subroutine, called `BANK(BANK_ID)`, contains the shared memory emulator part of the code and receives a (logical) bank number as parameter.

The general structure of these two subroutines are very close since each send (receive) must be met by a corresponding receive (send). Like the input program they are sequences of nested loop. The outermost loop nest defines which tile is executed. Each tile body is made of two or three sections:

1. a sequence of nested loops used to load the different local copies from the emulated shared memory; this sequence may be empty when there is no input, as is the case with a simple array initialization;

2. a sequence of nested loops to execute all iterations of the current tile; the loop body is a copy of the initial loop body but references to user variables are replaced by references to local variables; this section is empty in the memory emulator code;

3. a sequence of nested loops used to store the different local copies in the emulated shared memory; this sequence might be empty if there were no output but this is very unlikely!

5.2 Matrix transposition example

The transposition algorithm is fully parallel. In our approach each tile can be computed in parallel without synchronizations during computation. Communications are reduced to data movements of exact used and modified array elements. Data movements accessing contiguous elements in all cases where it is possible, this allows for an easy and efficient implementation of the two transfer routines called `WP65_send` and `WP65_receive`. Basically they only have to initialize DMA transfers.

Compared to our approach, the data partitionning approach, with classical data distribution, would generated codes with either less parallelism (all the transposition having to be computed on the same processor) or a lot of extra-communications (the matrix `M` having to be distributed on each processor).

The *Virtual Shared Memory* approach parallelism is based on control partitionning. Because data dependencies are computed on *unit* memory allocation and not on *page* memory allocation, synchronizations must be added during computation in order to preserve the memory coherency. Unless parallel tasks access exactly one (or a multiple of) page(s) of data, the useful parallelism will be smaller than in our approach because pages will be exchanged between processors all the time. Some 2-D blocking would prove necessary. The hardware approach based on consistent cache implies the same constraints.

Because of its control partitionning approach, our scheme creates parallel tasks, independent of any data distribution and containing no synchronizations. Data movements are optimized in order to transfer only the sets of data used or

Excerpt of the computation code for matrix transposition

```
                SUBROUTINE COMPUTE(PROC_ID)
                INTEGER PROC_ID,BANK_ID,L,I_0,L_I,J_0,L_J,L_I_1,L_I_2
                REAL*8 L_M_0_0(0:24,0:24),L_M_1_0(0:24,0:24)

C               DISTRIBUTED CODE FOR TRANSP

C               TO SCAN THE TILE SET FOR COMPUTATIONAL PROCESSOR
                DO 99975 I_0 = PROC_ID, 3, 4
                   DO 99976 J_0 = I_0, 3
                      DOALL 99994 BANK_ID = 0, 3
                         DO 99995 L_J = MAX(0, 25*I_0-25*J_0), MIN(24, -25*J_0+
           &                98)
                            DO 99996 L = (-1*BANK_ID+I_0+100*J_0+4)/4, MIN((396
           &                -BANK_ID+I_0)/4, (100-BANK_ID+I_0+100*J_0)/4)

                               L_I_1 = MAX(0, -2500*J_0-100*L_J-25*I_0-100+25*
           &                   BANK_ID+100*L)
                               L_I_2 = MIN(24, 25*J_0+L_J-25*I_0, -2500*J_0-100
           &                   *L_J-25*I_0-76+25*BANK_ID+100*L)
                               CALL _RECEIVE(BANK_ID, L_M_0_0(L_I_1,L_J),
           &                   L_I_2-L_I_1+1)

99996                       CONTINUE
99995                    CONTINUE
99994                 CONTINUE
                            ...

C               TO SCAN EACH ITERATION OF THE CURRENT TILE
                      DO 99987 L_I = 0, 24
                         DO 99988 L_J = MAX(0, -25*J_0+L_I+25*I_0), MIN(24, -25
           &             *J_0+98)
                            T = L_M_0_0(L_I,L_J)
                            L_M_0_0(L_I,L_J) = L_M_1_0(L_J,L_I)
                            L_M_1_0(L_J,L_I) = T
200                         CONTINUE
99988                    CONTINUE
99987                 CONTINUE
                         ...
99976              CONTINUE
99975           CONTINUE
                RETURN
                END
```

Excerpt of the shared memory emulator for matrix transposition

```
      SUBROUTINE BANK(BANK_ID)
      INTEGER PROC_ID,BANK_ID,L,O,I_0,L_I,J_0,L_J,O_1,O_2
      REAL*8 ES_M(0:24,0:99)
C     BANK DISTRIBUTED CODE FOR TRANSP
C     TO SCAN THE TILE SET FOR BANK
      DO 99973 I_0 = 0, 3
         PROC_ID = MOD(I_0, 4)
         DO 99974 J_0 = I_0, 3
            DO 99997 L_J = MAX(0, -25*J_0+25*I_0), MIN(24, 98-25*J_0)
               DO 99998 L = (4+I_0-BANK_ID+100*J_0)/4, MIN((396+I_0-
     &         BANK_ID)/4, (100+I_0-BANK_ID+100*J_0)/4)
                  O_1 = MAX(100*L_J+2500*J_0+25*I_0-100*L-25*BANK_ID+
     &            100, 0)
                  O_2 = MIN(100*L_J+2500*J_0+25*I_0-100*L-25*BANK_ID+
     &            124, 101*L_J+2525*J_0-100*L-25*BANK_ID+100, 24)
                  CALL _SEND(PROC_ID, ES_M(O_1,L), O_2-O_1+1)
99998          CONTINUE
99997       CONTINUE
            ...
99974    CONTINUE
99973 CONTINUE
      RETURN
      END
```

modified during the computation. Extra data are copied only in rare cases (non-convex set of used array elements).

6 Conclusion

The differents compilation phases presented in this paper were developped in C and are a part of the PIPS prototype[10]. PIPS is an automatic parallelizer developped to investigate interprocedural issues in automatic parallelization. It provided a very good support for the implementation of this new project.

The compilation scheme presented here was constrained by and optimized for T9000 and C104 based machines. The emphasis was put on:

- minimization of scheduling, synchronization and communication costs by tiling;
- data locality increase by tiling;
- data movement optimization (optimized copy allocation, vectorized transfers);
- overlap of communication and synchronizations (latency hiding by excess parallelism).

Generated codes were executed on networks of transputers and workstations. They were shown to be correct. The optimization problems encountered in data movement code generation are similar to optimization problems that will have to be coped with when using other implicit distribution schemes like virtual shared memory and cache. These problems are harder than those encountered with explicit distribution scheme. Algorithms described in [2] were found to reach their complexity limits in term of inequality number and coefficient magnitude for smaller problems than expected.

Further work is needed to assess the efficiency of the technique presented here. Experiments should be carried out on different platforms, including Transputer networks. It is also necessary to extend the input language and to support interprocedurality, nested parallelism, conditional statements, indirection, input-output, etc... Finally additional optimizations, like array privatization, and analyses, like static space estimation, will have to be researched.

References

[1] C. Ancourt, *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales*, Thse de doctorat, Unversité Paris 6, 1990

[2] C. Ancourt, F. Irigoin, *Scanning Polyhedra with DO loops*, Third ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'91), Williamsburg, 1991

[3] F. Andre, J-L. Pazat, H. Thomas, Pandore: a system to manage data distribution, *Int. Conf. on Supercomputing*, pp.380-388, June 1990

[4] D. Callahan, K. Kennedy, Compiling programs for distributed memory multiprocessors, *Journal of Supercomputing*, Vol.2, pp.151-169, 1988

[5] D. Gelernter, S. Ahuja, N. Carriero, Linda and Friends, *IEEE Computer*, Vol. 19, pp.26-34, august 1986.

[6] H.M. Gerndt, *Automatic parallelization for distributed-memory multiprocessing systems*, PhD, Bonn University, 1989

[7] E. Granston, S. Turner, A. Veidenbaum, *Design and analysis of a scalable, shared-memory system with support for burst traffic*, Tech. Report CSRD, 1991

[8] E. Granston, A. Veidenbaum, *An integrated hardware-software solution for effective amangement of local storage in high-performance systems*, ICPP'20, 1991

[9] F. Irigoin, R. Triolet, *Supernode Partitioning*, ACM Symposium on Principles of Programming Languages, San-Diego, 1988

[10] F. Irigoin, P. Jouvelot, R. Triolet, *Semantical Interprocedural Parallelization: An Overview of the PIPS Project*, ACM International Conference on Supercomputing (ICS'91), Cologne, 1991

[11] K. Li, P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM transactions on Computer Systems,* Vol.7, No.4, Nov. 89,pp321-359

[12] C. Koelbel, P. Mehrotra, Supporting Shared Data Structures On Distributed Memory Architectures, *Tech. report CSD-TR-915, Dep. of Computer Science, Purdue University*, 1990

[13] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *International Conference on Parallel Processing*, 1985, pp. 764-771.

[14] G. F. Pfister, An Introduction to the RP3, in: J. J. Dongarra, ed., *Experimental Computing Architectures* (North-Holland, Amsterdam, 1987).

[15] L. Zhou, *Enhanced Static Evaluation of Fortran Programs in the PIPS Environment*, Tech. Report E/160/CRI, Ecole des Mines de Paris, 1991