(Lecture Notes in Computer Science No. 487, Arndt Bode Ed. Springer-Verlag. ISBN: 0-387-53951-4)

On Automatic Loop Data-Mapping for Distributed-Memory Multiprocessors *

J. Torres, E. Ayguadé, J. Labarta, J. M. Llaberia and M.Valero

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
c/ Sor Eulalia de Anzizu, Mòdul D4. 08034 - Barcelona. SPAIN
e_mail: torres@ac.upc.es

Abstract

In this paper we present a unified approach for compiling programs for Distributed-Memory Multiprocessors (DMM). Parallelization of sequential programs for DMM is much more difficult to achieve than for shared memory systems due to the exclusive local memory of each Virtual Processor (VP). The approach presented distributes computations among VPs of the system and maps data onto their private memories. It tries to obtain maximum parallelism out of DO loops while minimizing interprocessor communication.

The method presented, which is named **Graph Traverse Scheduling** (**GTS**), is considered in this paper for single-nested loops including one or several recurrences. In the parallel code generated, dependences included in a hamiltonian recurrence that involves all the statements of the loop are enforced by the sequential execution of the computation assigned to each VP. Other dependences not included in the hamiltonian recurrence and involving data mapped onto different VPs will need explicit communication and synchronization.

1. INTRODUCTION

Parallelizing compilers exist today for high performance parallel computers in order to efficiently execute sequential programs written in conventional languages such as Fortran and C. These compilers mainly examine DO loops trying to obtain parallel code semantically equivalent to the original sequential one. DO loops offer a great amount of potential parallelism in numerical programs. Such parallelizing compilers perform the restructuring process based on dependence analysis for subscripted variables within the scope of each loop. Such dependences impose an execution order of the statements involved that must be preserved in the parallel code generated.

Most of the work on parallelizing compilers has been done for shared-memory multiprocessors [PGHL89, AlKe87]. On the other hand, few works have been presented in the literature for partitioning DO loops into computations well-suited for DMM systems [RaSa89]. In this case, it is necessary to automatically distribute computations and data among processors trying to maximize the parallelism obtained with a good load-balance over time while minimizing the amount of interprocessor communication required.

Other approaches try to extend a programming language with directives that control the mapping of variables to local memories [CaKe88, KeZi89, Tsen89]. The compiler automatically tries to perform a task partitioning assuming the data partitioning specified by the user. It also inserts all message-passing communication that is required to maintain the semantics of the original sequential program. Optimizing communication by message fussion is vital for obtaining efficient parallel programs [Gern90].

Another approach is to systematically map systolizable problems onto DMM [IbSo89, FLNV91]. The systolic algorithm derived from the specification of the problem is partitioned in order to adapt it to the size of the available DMM. The systolic algorithm is also increased in granularity in order to reduce communication overhead.

* *This work has been supported by the Ministry of Education of Spain (CICYT) in program TIC 299/89 and 392/89*

In this paper we describe **GTS** as a method for partitioning recurrences included in single-nested loops and generating code well-suited for DMM systems. The extension to the multiple-nested loop case can be found in [TALL90] and is not included in this paper for space reasons.

First **GTS** partitions the bounded statement per iteration space in threads in order to obtain maximum parallelism. Each thread consists of a set of points of the space linked by a dependence chain. Each thread will be executed as a task in a VP of the DMM system. The partitioning step assumes the existence of a hamiltonian recurrence in the dependence graph that generates the set of threads. Such hamiltonian recurrence involves all loop statements. If not present, one must be obtained by adding dummy dependences that do not limit the parallelism of the loop. The method proposed obtains a fully independent partition when a single hamiltonian recurrence appears in the dependence graph.

After that, referenced array elements are mapped onto private memories based on the distribution of computations between threads obtained in the previous step. In order to minimize interprocessor communication, all data computed or used in a thread are located in the private memory of the VP that executes this thread. Finally, dependences not included in the hamiltonian recurrence and involving data used on different VPs are satisfied by using the correct data communication and synchronization primitives.

This method was first presented in [ALTB89] and [ALTL90] as an approach for compiling single and multiple-nested loops for shared-memory multiprocessors, respectively. The outline of the remaining sections in this paper is as follows. In section 2 we describe the model of target machine for which **GTS** is presented. In section 3 we review some concepts and definitions on data dependences used along this paper. Section 4 presents **GTS** as a unified approach to task and data partitioning of single-nested loops derived from the original flow-dependence relations of the sequential program. Some comments to the multiple-nested case are given in section 5. The main concluding remarks and future work are given in section 6.

2. TARGET MACHINE AND PROGRAMMING STYLE

The architectural model that we consider in this paper is a fully-connected DMM system. In this model, processors do not have access to a shared memory. Each processor has only access to some amount of local private memory. In a fully-connected model, any processor of the DMM can exchange data with other processors through a direct link of the communication network. Routing switches provide this logically fully-connected network on DMM with a not fully-connected physical topology [iPSC88, Poun90]. In this case, it is assumed that end to end delay is not much larger than point to point delay. If load is high, congestion on the physical links will, of course, increase this delay.

The approach to processor communication is the message-passing model, where processors communicate through explicit messages by using send and receive like primitives. The statement *SEND (dest, var)* sends variable *var* from local memory to processor *dest*. The statement *RECEIVE (src, var)* receives a datum from processor *src* and stores it in variable *var*. We assume that *SEND* operations will complete without blocking while *RECEIVE* operations will block if there is no datum available.

Programming style is that a VP sends a new computed value as soon as possible if it is needed by another VP. The VP that needs this value performs a receive operation on the VP that computes it as late as possible.

3. DEFINITIONS

Restructuring compilers are based on the analysis of dependences among a collection of statements ($\mathbf{S_1}$, $\mathbf{S_2}$, ..., $\mathbf{S_s}$) within the scope of a normalized loop. Dependence relations between statements reflect a given execution order that cannot be modified by the restructuring process.

As a result of the dependence analysis, a directed *dependence graph* **G** (**V**, **E**) is obtained, in which **V** is a set of nodes $\mathbf{V} = \{\mathbf{S_1}, ..., \mathbf{S_s}\}$ representing statements in the loop body, and **E** is a set of arcs $\mathbf{E} = \{\mathbf{d_{ij}} \mid \mathbf{S_i}, \mathbf{S_j} \in \mathbf{V}\}$ representing dependence relations between statements of the loop.

Between each pair of statements $\mathbf{S_i}$ and $\mathbf{S_j}$, where $\mathbf{S_i}$ precedes $\mathbf{S_j}$ in the sequential execution of the loop, some *data dependences* are defined in the literature [KKPL81]. In our model only *flow-dependences* are of concern [ChCh87]. Statement S_j is flow-dependent on statement S_i if S_j uses a variable that S_i can modify. *Anti* and *output dependences* due to the reuse of variables can be removed by assigning different variable names to different versions of data.

Each arc $d_{ij} \in E$ involves two statements of the loop body (source and sink statement) and it has an associated *dependence distance* d_{ij} representing the number of iterations the dependence extends across. In this paper we only consider data dependences with constant dependences known at compile time.

A *chain* C_{ij} is an ordered set of arcs $C_{ij} = \{d_{ik}, d_{kl}, ..., d_{mj}\}$ between two statements S_i and S_j such that each node in the chain is visited only once. Given a chain C_{ij}, we define its *weight* $w(C_{ij})$ as

$$w_{ij} = \sum_{d_{lm} \in C_{ij}} d_{lm}$$

This weight $w(C_{ij})$ represents the number of iterations between any pair of instances of statements S_i and S_j.

A *recurrence* **R** is a cycle or closed chain in the dependence graph. A *Hamiltonian Recurrence* is a recurrence going through all nodes in the dependence graph.

Let $B = \{ R_1, R_2, ..., R_r \}$ be the set of recurrences in a given dependence graph **G**. This graph **G** is an *acyclic dependence graph* when $B=\emptyset$ and it is a *cyclic dependence graph* when $|B| \geq 1$. When at least one recurrence of **B** is hamiltonian, the graph is called *hamiltonian graph*.

The *iteration space* **IS** of a loop is the set of points defined by a vector index $I = \langle i_1, i_2, ..., i_n \rangle$ in a space with dimensionality equal to the depth of the nested-loop structure. Each point represents the execution of an iteration of the loop body. In the scope of this paper we will consider n=1. The *statement per iteration space* **SIS** of a loop is the set of points defined by the cartesian product **IS** × **V**, being **V** the set of nodes of the loop. Each point S_{rI} in this space represents the execution of a given iteration I of a given statement S_r

$$SIS = \{ S_{rI} \mid 1 \leq r \leq s,\ 1 \leq I \leq N \}$$

Dependence relations $d_{ij} \in E$ impose an execution order between any pair of points in **SIS**

$$S_{rI} \text{ and } S_{r\,(I+d_{ij})}$$

4. GRAPH TRAVERSE SCHEDULING

In this section we present **GTS** as a unified approach to task and data partitioning in an automatic restructuring environment for DMM. First **GTS** performs a partitioning of **SIS** in threads so that maximum parallelism is obtained. After that, data mapping is done so that interprocessor communication is minimized.

GTS is based on the knowledge of the average parallelism of the loop evaluated as described in [ALTL90]. This measure of parallelism can be defined as the average number of active processors executing iterations of the loop. Average parallelism can be evaluated as the quotient between the time to execute the sequential version and the time required to execute the longest path through **SIS**. The longest path is obtained by traversing the most restrictive recurrence of the dependence graph.

The partitioning strategy proposed covers the maximum number of possible dependences within the sequential execution of each VP. The algorithm is considered in sections 4.1 and 4.2 for single-recurrence hamiltonian graphs. If there is no such hamiltonian recurrence, one must be obtained by adding dummy dependences that do not limit the parallelism of the loop. Other flow-dependences involving data mapped on different VPs require explicit interprocessor communication introduced as described in section 4.3. Parallel code generation is briefly considered in section 4.4.

4.1 Thread Partitioning

GTS performs two basic operations: alignment of the loop body through a hamiltonian recurrence and an appropriate assignment of computations to VPs so that communication primitives can be easily introduced when needed.

Given a recurrence **R**, we define a *thread* as the set of points in **SIS** directly dependent through the recurrence. Let *Thread Set* **TS** be the minimum set of threads generated by a hamiltonian recurrence **R** that cover the whole **SIS**.

Each thread in **TS** can be characterized by the point in **SIS** which does not depend on any previous execution and from which the whole thread can be obtained by traversing recurrence **R**. Each arc $\mathbf{d_{ij}}$ in **R** determines those points of **SIS** associated to the sink statement S_j that can be initially executed. The set of initial dependence-free points can be expressed as

$$S_{jx} \mid 1 \le x \le d_{ij} .$$

Figure 1.b shows the threads generated by the hamiltonian recurrence for the loop and dependence graph of figure 1.a. Initial dependence-free points are shown with filled shapes.

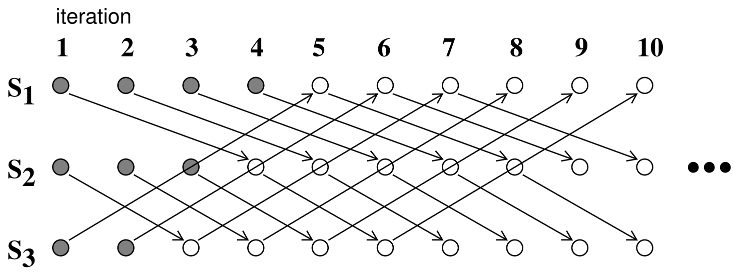
GTS assigns each thread to a different VP of the DMM system. Dependences within the hamiltonian recurrence are embedded in the sequential execution of each thread. Observe that fully independent threads are obtained when dealing with single-recurrence hamiltonian graphs. Figure 1.c shows the assignment of threads to VPs proposed for the previous example. The assignment proposed fulfils that consecutive iterations of a given statement are executed in consecutive VPs. This characteristic will ease interprocessor communication described in section 4.3.

The scheduling of operations can be obtained by traversing the graph backwards and assigning to consecutive VPs all initial dependence-free iterations of each statement. In the assignment proposed, a point S_{jk} of the **SIS** is executed in virtual processor vp given by

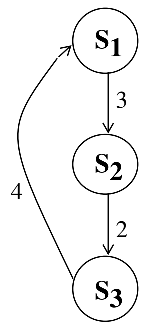
$$vp = (k + w_{j1} - 1) \bmod w_R \qquad (1)$$

denoting by w_{j1} the weight of the chain $\mathbf{C_{j1}}$ through **R** and w_R its weight.

```
    DO i = 1, 98
S1:   A[i + 3] = B[i]
S2:   C[i + 2] = A[i] * D[i - 1]
S3:   B[i + 4] = C[i] / 3
    ENDO
```



(b)



(a)

vp_0	vp_1	vp_2	vp_3	vp_4	vp_5	vp_6	vp_7	vp_8
S_{11}	S_{12}	S_{13}	S_{14}	S_{31}	S_{32}	S_{21}	S_{22}	S_{23}
S_{24}	S_{25}	S_{26}	S_{27}	S_{15}	S_{16}	S_{33}	S_{34}	S_{35}
S_{36}	S_{37}	S_{38}	S_{39}	S_{28}	S_{29}	S_{17}	S_{18}	S_{19}
$S_{1\ 10}$	$S_{1\ 11}$	$S_{1\ 12}$	$S_{1\ 13}$	$S_{3\ 10}$	$S_{3\ 11}$	$S_{2\ 10}$	$S_{2\ 11}$	$S_{2\ 12}$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$S_{1\ 91}$	$S_{1\ 92}$	$S_{1\ 93}$	$S_{1\ 94}$	$S_{3\ 91}$	$S_{3\ 92}$	$S_{2\ 91}$	$S_{2\ 92}$	$S_{2\ 93}$
$S_{2\ 94}$	$S_{2\ 95}$	$S_{2\ 96}$	$S_{2\ 97}$	$S_{1\ 95}$	$S_{1\ 96}$	$S_{3\ 93}$	$S_{3\ 94}$	$S_{3\ 95}$
$S_{3\ 96}$	$S_{3\ 97}$	$S_{3\ 98}$		$S_{2\ 98}$		$S_{1\ 97}$	$S_{1\ 98}$	

(c)

*Figure 1: (a) Example of a single-recurrence hamiltonian dependence graph. (b) Statement per Iteration Space and threads. (c) Thread partitioning and assignment obtained by **GTS**.*

4.2 Data Partitioning

Data partitioning is done by **GTS** so that the amount of communication between VPs is minimized. It considers the thread partitioning obtained as described in the previous section.

The basic idea is that each VP stores in its private local memory all array elements referenced by the assigned thread. In the case of single-recurrence hamiltonian graphs, these elements are computed and used in the same thread. In the case of multiple-recurrence graphs, a given vector element can be computed and used in threads assigned to different VPs. In this case, data is stored in the local memory of the VP that computes them. Communication primitives ensure the use of the correct value by the VP that uses them.

For each vector V in the original program, we generate a vector LV in each VP. Next we present the function that maps a given element of vector V onto the local vector LV of a VP:

* If V is a vector whose elements are computed in statement S_j, a given element V [f(i)] is stored in local memory of virtual processor

$$vp = (f(i) - a + w_{j1} - 1) \bmod w_R ,$$

assuming linear indexing functions $f(i) = i + a$ for the vector variable V.

The position of this element in the local vector LV is given by

$$(f(i) - a + w_{j1} - 1) / w_R$$

* If W is a vector whose elements are not computed within the loop but used in a assignment statement like

$$S_j:\ V[f(i)] = \ \dots \ W[g(i)]$$

a given element W[g(i)] is stored in the same local memory as vector element V[f(i)]. Assuming linear indexing functions $f(i) = i + a$ and $g(i) = i + b$ for both vector variables, element W[g(i)] is stored in the local memory of virtual processor

$$vp = ((g(i) - a - b) + w_{j1} - 1) \bmod w_R$$

The position of this element in the local vector LW is given by

$$(g(i) - a - b + w_{j1} - 1) / w_R$$

As a result, there will be as many copies of a vector variable W as different uses in the loop.

With this distribution of data, only those elements of a vector used in the thread are stored in the local memory of the VP that executes it. For each vector V of size N, the size of the local vector LV is $\lceil N / w_R \rceil + 1$.

Figures 2.a through 2.c show the mapping of vector variables computed in each statement of the single-nested loop of figure 1.a. Only those shaded elements must be initially loaded in local memories. Other elements are computed during the execution of threads. Figure 2.d shows the mapping of variable D which is used but never computed within the loop. In this case all vector elements must be initially loaded in local memories.

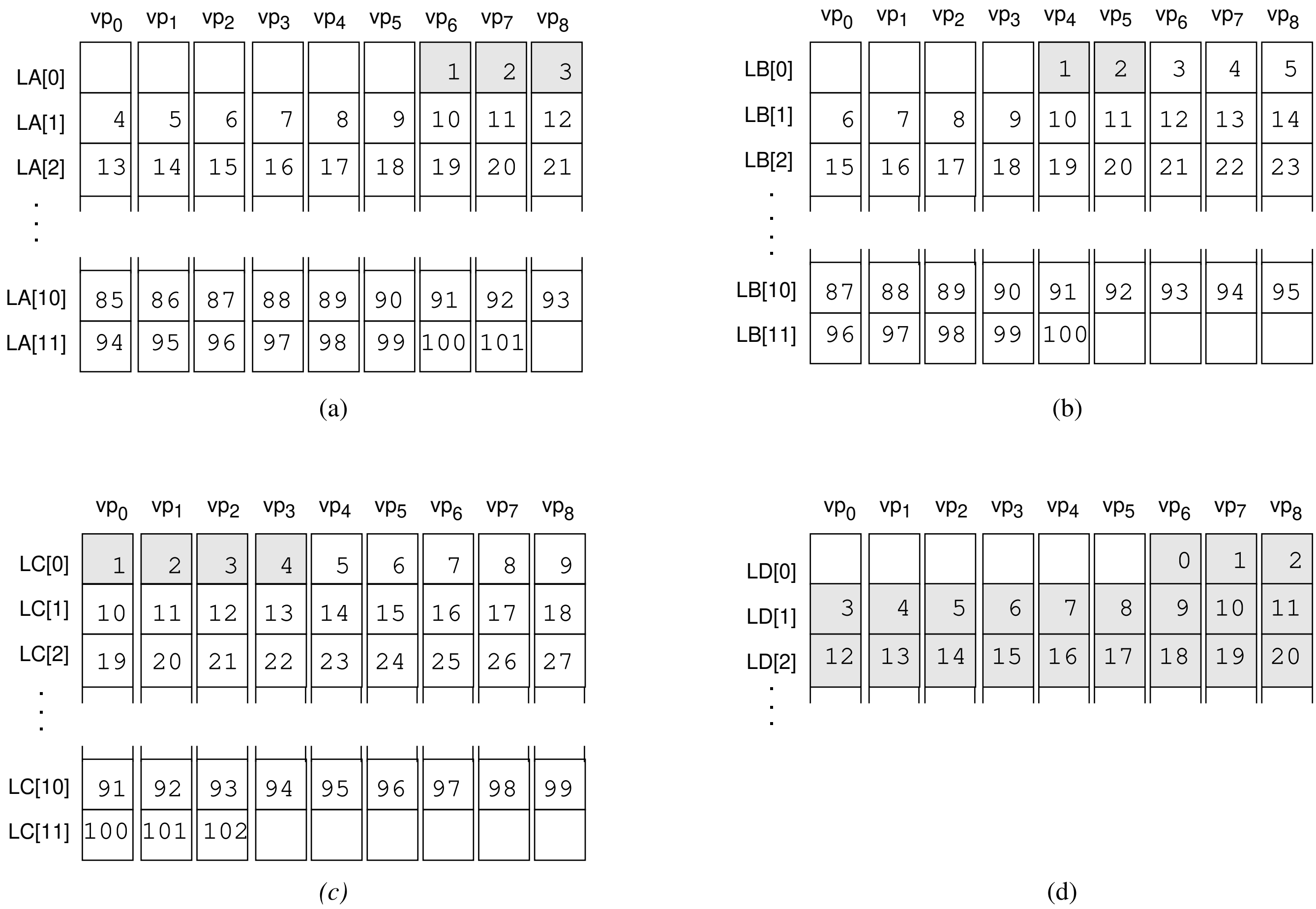


Figure 2: Mapping onto virtual processors of the elements of vector variables A, B, C and D of example 1.

4.3 Data Communication

In the case of a hamiltonian graph with more than one recurrence, the scheduling is performed by applying the same procedure described previously to a hamiltonian recurrence $\mathbf{R_{sch}}$ of the loop. Once $\mathbf{R_{sch}}$ has been obtained, dependences not included in $\mathbf{R_{sch}}$ and involving data mapped on different VPs require the use of communication primitives. On the other hand, dependences included in $\mathbf{R_{sch}}$ are embedded in the sequential execution of each thread.

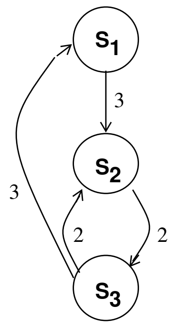
Explicit communication must be introduced for any arc $\mathbf{d_{ij}} \notin \mathbf{R_{sch}}$ in the graph going from node S_i to node S_j. Detailed proofs of expressions given in this section can be found in [TALL90].

For each arc $\mathbf{d_{ij}} \notin \mathbf{R_{sch}}$, a *send* operation to virtual processor *vp'* must be executed in virtual processor *vp* after the source statement S_i. A *receive* operation from virtual processor *vp* must be executed in virtual processor *vp'* before the sink statement S_j. Due to the thread assignment proposed in the previous section, the relationship between both *vp* and *vp'* can be expressed as follows:

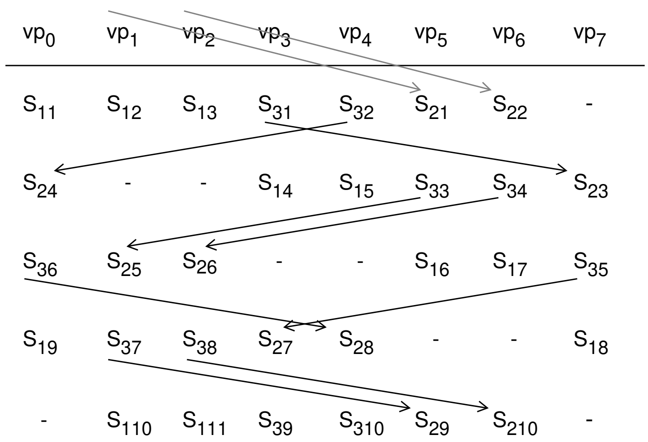
$$vp' = (vp + d_{ij} - w_{ij}) \bmod w_R \qquad \text{and} \qquad vp = (vp' - d_{ij} + w_{ij}) \bmod w_R \tag{2}$$

Figure 3.b shows the thread partitioning obtained by **GTS** for the dependence graph of figure 3.a. Arrows represent data communication that must be introduced due to the flow-dependence d_{32}. Observe that virtual processors vp_5 and vp_6 need some elements of vector C not computed in virtual processors vp_1 and vp_2 respectively due to the actual bounded iteration space. These elements must be initially sent in order to allow the execution of threads that use them.

```
      DO i = 1, 99
S1:     A[i + 3] = A[i - 5] + C[i]
S2:     B[i + 2] = A[i] - C[i + 1] * 2
S3:     C[i + 3] = B[i] / 3
      ENDO
```



(a)



(b)

*Figure 3: (a) Multiple-recurrence hamiltonian graph. (b) Thread partitioning and communication introduced by **GTS**.*

Any dependence arc $\mathbf{d_{ij}} \notin \mathbf{R_{sch}}$ allows the execution of the first d_j iterations of the sink statement S_j. In order to allow their execution, those elements of the vector variable that causes the dependence must be initially sent by the appropriate VPs. Taking into account where the free iterations of the sink statement are executed (1), each virtual processor *vp* will do the following number of initial sends

$$\lfloor d_{ij} / w_R \rfloor + 1 \qquad \text{if} \quad (vp - w_{i1} + d_{ij}) \bmod w_R < d_{ij} \bmod w_R$$

$$\lfloor d_{ij} / w_R \rfloor \qquad \text{if} \quad (vp - w_{i1} + d_{ij}) \bmod w_R \geq d_{ij} \bmod w_R$$

to virtual processors *vp'* given by (2).

Observe that if we consider a new dependence relation d_{11} in the graph of figure 3.a with an associated distance d_{11}=8, this dependence relation will not need explicit inter-processor communication because the element of vector A used in a given iteration of a thread has been computed in the previous iteration of the same thread.

4.4 Code generation

Finally, parallel code must be generated so that each processor of the DMM system executes a given thread and establishes communication with the appropriate processors. In the parallel code generated, all VP execute the same program on the variables allocated in their local address space. Explicit communication is automatically inserted to provide access to non-local data.

Figure 4 shows a possible version of the parallel code generated by **GTS** for the sequential loop of figure 3. In this case, it can be decomposed in three parts: *prolog, core and epilog.*

```
DOACROSS j= 0,7
 Δ=⌊(93-j)/8⌋
-------------------------------------------------------
  IF (j>=1 && j<=2)
      send ((j-4) mod 8, LC[0])
  ENDIF
  IF (j>=5)
      receive((j+4) mod 8, X)
      LB[0] = LA[0] - X * 2
 ENDIF                                          prolog part
 IF (j>=3)
      LC[0] = LB[0] / 3
      send ((j-4) mod 8, LC[0])
 ENDIF
-------------------------------------------------------
 DO i = 1,1 + Δ
      LA[i] = LA[i-1] + LC[i-1]
      receive ((j+4) mod 8, X )
      LB[i] = LA[i] - X * 2
      LC[i] = LB[i] / 3                         core part
      send ((j-4) mod 8, LC[i])
 ENDDO
-------------------------------------------------------
 IF ((1+Δ) * 8 + j < 99)
      LA[1+Δ] = LA[Δ] + LC[Δ]
 ENDIF
 IF ((1+Δ)*8 + j + 3) < 99)
      receive((j+4) mod 8, X)                   epilog part
      LB[1+Δ] = LA[1+Δ] - X * 2
 ENDIF
ENDOACROSS
-------------------------------------------------------
```

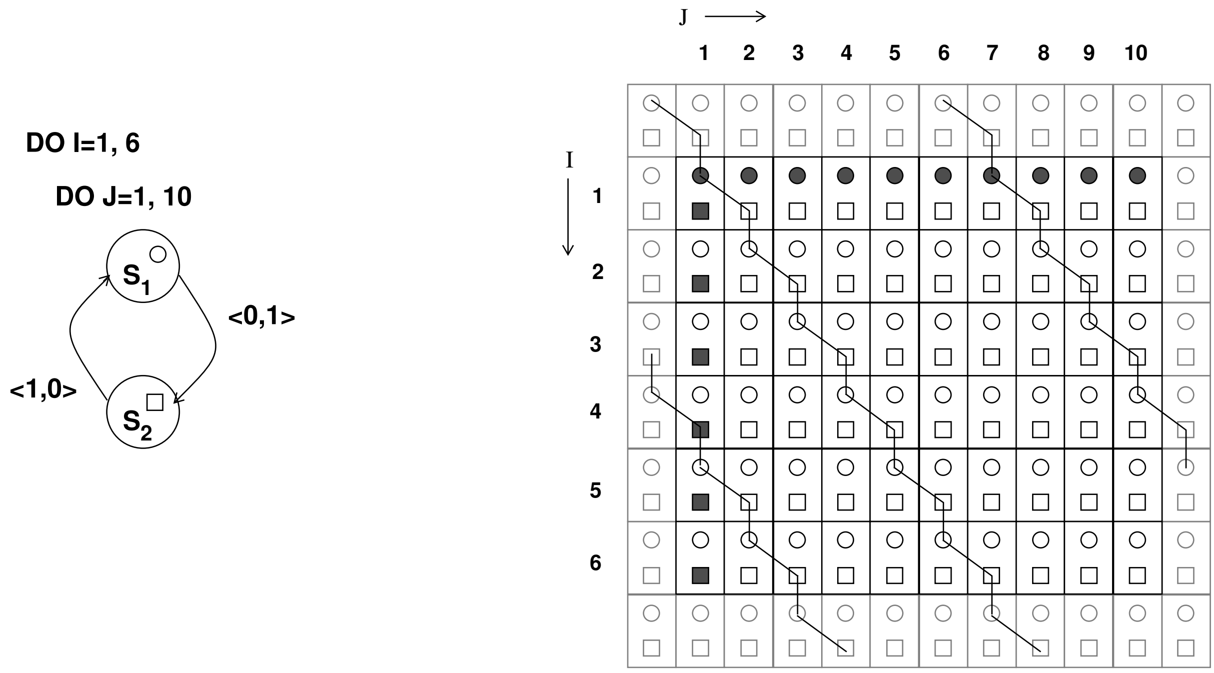
Figure 4: Parallel code generated for the example of figure 3.

In the prolog part, each processor executes the initial send primitives and some initial iterations of statements extracted from the inner sequential DO loop. This has been done in order to use the same code for all the processors. The epilog part executes the final part of the thread that can not be executed in a complete iteration of the core part. The general code structure is described in [TALL90].

5. SOME CONSIDERATIONS TO THE MULTIPLE-NESTED CASE

In this section we briefly outline some aspects on the extension of **GTS** to the multiple-nested loop case. Figure 5 shows a dependence graph and **SIS** for a possible double-nested loop. In this case we distinguish between *unbounded and bounded statement per iteration spaces.* The bounded **SIS** is the finite subset of the unbounded one determined by the actual loop iteration limits. Points outside the bounded **SIS** are drawn in dashed lines in figure 5.b. In this case, a thread is considered as the set of points of the bounded **SIS** linked by a dependence chain in the unbounded space. Figure 5.b shows some of the threads in the **SIS** generated by the hamiltonian recurrence of figure 5.a

The length of the threads generated is not constant so the load assigned to processors of the DMM will not be balanced if as many VPs as threads are allocated. A good load balancing can be obtained if we statically group threads without overcoming the execution time of the largest thread before grouping. It is important to guarantee that the grouping of threads is deadlock-free when dependences of the graph require explicit synchronization. In the example of figure 5, the two shortest threads can be executed in the same VP without overcoming the execution time of the longest thread generated before grouping.



(a) *(b)*

*Figure 5: (a) dependence graph for a double nested-loop and (b) associated **SIS** and threads generated.*

As in the single-nested loop case, each matrix V is distributed among local memories of VPs and stored in one-dimesional local vectors LV. Each VP stores only those elements computed by the thread assigned and those used by it but not computed in the loop.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have extended a previous work on loop parallelization for shared-memory machines to the distributed memory ones. In this case a unified approach to data and task partitioning has been considered in order to achieve maximum parallelism and minimum interprocessor communication.

Next we briefly comment some open questions left by the work presented in this paper. The partitioning method presented assumes the existence of a hamiltonian recurrence in the dependence graph. This is not the common case, so the problem must be taken into consideration [Aygu89]. A hamiltonian recurrence can be obtained by adding a set of dummy dependences **E'** such that the parallelism of the loop is not limited by the new set of recurrences that appear in the dependence graph. Many sets of dummy arcs **E'** can be used to obtain a hamiltonian recurrence. In this case, we will choose that solution which minimizes the number of threads generated and the cardinality of set **E'**, in order to reduce the amount of synchronization required. Fast heuristics for obtaining good **E'** sets should be looked into if the size of the problem becomes large enough.

Dummy-arcs addition is a technique that can be also used to modify the number of processors for which parallel code is generated. In this case we avoid the overhead due to dynamic scheduling of processors in systems with less processors than parallelism of the loop.

In the case of a sequence of loops, the data must be reorganized between the end of one loop and the begin of the next [GaJG88]. It will be interesting to minimize data reorganization by using a similar dummy-arc addition technique by considering the whole loop sequence.

In the case of non fully-connected DMM, it would be interesting to reduce data routing through processing elements by performing a mapping of virtual to real processors, taking into consideration the physical interconnection topology.

REFERENCES

[AlKe87] J.R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form", ACM Transactions on Programming Languages and Systems, Vol. 9, No. 4, October 1987.

[ALTB89] E. Ayguadé, J. Labarta, J. Torres and P. Borensztejn, "GTS: Parallelization and Vectorization of Tight Recurrences", Proc. of the Supercomputing'89, Reno-Nevada, November 1989.

[ALTL90] E. Ayguadé, J. Labarta, J. Torres, J.M. Llaberia and M. Valero, "Parallelism Evaluation and Partitioning of Nested Loops for Shared-Memory Multiprocessors", Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine-California, August 1990.

[Aygu89] E. Ayguadé, "Automatic Parallelization of Recurrences in Numerical Sequential Programs", Ph.D. Thesis, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Oct. 1989 (in spanish).

[ChCh87] Z. Chen and C-C. Chang, "Iteration-Level Parallel Execution of DO Loops with a Reduced Set of Dependence Relations", Journal of Parallel and Distributed Computing, No. 4, 1987.

[FLNV91] A. Fernandez, J.M. Llaberia, J.J. Navarro and M. Valero-Garcia, "Interleaving Partitions of Systolic Algorithms for Programming Distributed Memory Multiprocessors", Proceedings of the 2nd European Distributed Memory Computers Conference, Springer-Verlag (in this volume), 1991.

[CaKe88] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors", The Journal of Supercomputing, No. 2, October 1988.

[GaJG88] K. Gallivan, W. Jalby and D. Gannon, "On the problem of Optimizing Data Transfers for Complex Memory Systems", Proceedings of the 1988 ACM International Conference on Supercomputing, St. Malo-France, 1988.

[Gern90] H.M. Gerndt, "Automatic Parallelization for Distributed Memory Multiprocessing Systems", Ph.D. dissertation, University of Bonn, Technical Report Series ACPC/TR 90-1, Austrian Center for Parallel Computation, 1990.

[IbSo89] O.H.Ibarra and S.M.Sohn, "On Mapping Systolic Algorithms onto the Hypercube", Proceedings of the 1989 International Conference on Parallel Processing, Vol. I, August 1989.

[iPSC88] iPSC/2, Intel Corporation, 1988. Order Number 280110-001.

[KeZi89] K. Kennedy and H.P. Zima, "Virtual Shared Memory for Distributed-Memory Machines", Proceedings of the 4th Hypercube Conference, Monterey-California, 1989.

[KKPL81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe, "Dependence Graphs and Compiler Optimiza-tions", Proc. of the 8th ACM Symposium on Principles of Programming Languages Williamsburg, January 1981.

[PGHL89] C.D. Polychronopoulos, M. Girkar, M. R. Haghighat, C.L. Lee, B. Leung, D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", Proceedings of the 1989 International Conference on Parallel Processing, Vol. II, August 1989.

[Poun90] D. Pountain, "Virtual Channels: The Next Generation of Transputers", BYTE, April 1990.

[RaSa89] J. Ramanujam and P. Sadayappan, "A Methodology for Parallelizing Programs for Multicomputers and Complex Memory Multiprocessors", Proceedings of the Supercomputing'89, Reno-Nevada, November 1989.

[TALL90] J. Torres, E. Ayguadé, J. Labarta, J.M. Llaberia and M. Valero, "A Technique for Data and Task Partitioning of Nested Loops for Distributed-Memory Parallel Computers", Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, UPC/DAC Research Report RR-90/13, June 1990.

[Tsen89] Ping-Sheng Tseng, "A Parallelizing Compiler for Distributed Memory Parallel Computers", Ph.D. Thesis, Carnegie Mellon University, CMU-CS-89-148, May 1989.