AUTOMATIC DATA DISTRIBUTION FOR MASSIVELY PARALLEL PROCESSORS

AUTOMATIC DATA DISTRIBUTION FOR MASSIVELY PARALLEL PROCESSORS

Jordi GARCIA

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Barcelona

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor en Informàtica

FEBRUARY, 1997

ABSTRACT

Massively Parallel Processor systems provide the required computational power to solve most large scale High Performance Computing applications. Machines with physically distributed memory allow a cost-effective way to achieve this performance, however, these systems are very difficult to program and tune. In a distributed-memory organization each processor has direct access to its local memory, and indirect access to the remote memories of other processors. But the cost of accessing a local memory location can be more than one order of magnitude faster than accessing a remote memory location. In these systems, the choice of a good data distribution strategy can dramatically improve performance, although different parts of the data distribution problem have been proved to be NP-complete.

The selection of an optimal data placement depends on the program structure, the program's data sizes, the compiler capabilities, and some characteristics of the target machine. In addition, there is often a trade-off between minimizing interprocessor data movement and load balancing on processors. Automatic data distribution tools can assist the programmer in the selection of a good data layout strategy. These use to be source-to-source tools which annotate the original program with data distribution directives. Crucial aspects such as data movement, parallelism, and load balance have to be taken into consideration in a unified way to efficiently solve the data distribution problem.

In this thesis a framework for automatic data distribution is presented, in the context of a parallelizing environment for massive parallel processor (MPP) systems. The applications considered for parallelization are usually regular problems, in which data structures are dense arrays. The data mapping strategy generated is optimal for a given problem size and target MPP architecture, according to our current cost and compilation model.

A single data structure, named Communication-Parallelism Graph (CPG), that holds symbolic information related to data movement and parallelism inherent in the whole program, is the core of our approach. This data structure allows the estimation of the data movement and parallelism effects of any data distribution strategy supported by our model. Assuming that some program characteristics have been obtained by profiling and that some specific target machine features have been provided, the symbolic information included in the CPG can be replaced by constant values expressed in seconds representing data movement time overhead and saving time due to parallelization. The CPG is then used to model a minimal path problem which is solved by a general purpose linear 0-1 integer programming solver. Linear programming techniques guarantees that the solution provided is optimal, and it is highly efficient to solve this kind of problems.

The data mapping capabilities provided by the tool includes alignment of the arrays, one or two-dimensional distribution with `BLOCK` or `CYCLIC` fashion, a set of remapping actions to be performed between phases if profitable, plus the parallelization strategy associated. The effects of control flow statements between phases are taken into account in order to improve the accuracy of the model. The novelty of the approach resides in handling all stages of the data distribution problem, that traditionally have been treated in several independent phases, in a single step, and providing an optimal solution according to our model.

ACKNOWLEDGEMENTS

I am really in debt to Eduard Ayguadé, my thesis advisor, for his guidance, support, advice, and specially for being a friend during my stage as a graduate student. All my gratitude is for him. I would also like to thank Mateo Valero, who caught me to start researching on this never ending field. He is the guilty that I have written this stuff. I wish to thank Jesús Labarta, the leader of the compilers group, for his time and advice.

I feel very proud to belong to the DDTeam. I am indebted to Mercè Gironès for her support in my head-on collision against the ParaScope programming environment. She has been a great work-mate and currently she is a very good friend. Jordi Torres has left a trace in the inter-procedural analysis of DDT, but his personal encouragement has been much more fruitful for me. However, my best gratitude is for Mari Luz Grande, not only for her lots of megaflops in this work, but mainly for her trust in me and her great friendship. Thanks a lot to all the team for this nice work environment.

During my experience optimizing compilers I have met a lot of outstanding people. However I am indebted to Uli Kremer for the fruitful discussions in this topic in several *tapas* bars. I am also grateful to Elena Fernández for her insights in linear programming and for all her time. Professor Ron Perrott has given me a friendly relationship and the chance to work with him in Belfast. A lot more of people has influenced me on this research. It is exciting to know such brilliant researchers from all over the world.

I have been fortunate to work in the Computer Architecture Department, in which I have met a lot of excellent people. José Manuel Martín has been my first room mate, much better as a good friend than as researcher. And Josep Ramón Herrero, my current room mate, has kept a lot of interesting discussions about computer science and about *salsa*. They both have constantly been very patient with me.

I would also like to thank several other colleagues from this department for their company during these five years. A lot of dinners with award holders, assistant professors, and related, have provided an optimal work environment. Thanks to the people that have read a draft of this thesis and still are my friends. I feel privileged to have met this excellent group of researchers and good friends. Also thanks to the splendid technical support provided by the staff from the LCAC and from the CEPBA.

Special thanks go to my friends from Vilanova, that have never understood what I am doing here, but during multiple week-ends have supplied me feedback for these weeks of hard work. Thanks a lot to Meji, Juanki, Josep, Javi, Gabi, Juan Ramón, Panchi, Toni, Carlos, Janine, and more recently, Ignasi. My gratitude is also for Emilio Gutiérrez, my best academic fried, and Oriol Valero, who lent me his home for a summer and shares many lunches with me.

Finally, I want to thank all my family, my brother and his childs, for their love and support, but in special my sister Neus, and my niece and stepdaughter Cristina, that is the nicer child in the world. My deepest gratitude, however, is reserved to my parents, Daniel and Carme, those who have made possible me to be here, and those who have given me all their unconditional love and support during all my life. They have made this thesis possible, and this is dedicated to them.

This work was been partially supported by the Ministry of Education of Spain under contracts TIC 880/92 and TIC 429/95, by CIRIT grant BE94/1-100, the CEPBA (European Center for Parallelism of Barcelona), and by CONVEX Computer Corporation that has supported the development of the DDT tool.

AGRAIMENTS

Estic molt en deute amb l'Eduard Ayguadé, el meu director de tesi, per la seva orientació, suport, bons consells, però especialment per ser un amic durant la meva etapa com a estudiant de doctorat. Tot el meu agraïment és per a ell. També m'agradaria donar les gràcies a Mateo Valero, que em va enganxar per a fer recerca en aquest mon inacabable. Ell és el culpable de que jo hagi escrit això. Desitjaria agrair a Jesús Labarta, el cap del grup de compilació, l'haver-me dedicat part del seu temps i dels seus consells.

Em sento realment orgullós de pertànyer al DDTeam. Estic en deute amb la Mercè Gironès per el seu suport en el meu xoc frontal contra l'entorn de programació ParaScope. Ella ha sigut una gran companya de treball i és una molt bona amiga. En Jordi Torres ha deixat la seva empremta en l'anàlisi interprocedural del DDT, però els seus ànims personals han estat molt més fructífers per a mi. Malgrat això, el més sinzer agraïment és per la Mari Luz Grande, no només per tots els megaflops que ha posat en aquest treball, sinò que principalment per la seva confian ca en mi, i per la seva gran amistat. Moltes gràcies a tot l'equip per aquest agradable entorn de treball.

Durant la meva experiència optimitzant compiladors m'he trobat amb molta gent excepcional. Però em sento endeutat amb l'Uli Kremer per les bones xerrades en aquesta àrea de recerca en múltiples bars de tapes. També estic agrait a la Elena Fernández per la seva ajuda en la programació lineal i per tot el temps que m'ha dedicat. El Professor Ron Perrott m'ha donat una bona amistat i l'oportunitat de treballar amb ell a Belfast. Molta més gent m'ha influenciat en aquesta recerca. És excitant conèixer aquest grup d'investigadors tant brillants d'arreu el món.

He estat afortunat de treballar en el Departament d'Arquitectura de Computadors, en el qual he conegut molta gent excel.lent. En José Manuel Martín ha sigut el meu primer company de despatx, molt millor com a bon amic que com a investigador. Amb en Josep

Ramón Herrero, el meu actual company de despatx, he tingut moltes bones xerrades relacionades amb l'informàtica i amb la salsa. Tots dos han estat constantment molt pacients amb mi.

També m'agradaria agrair molts altres col.legues d'aquest departament per la seva companyia durant aquests cinc anys. Molts sopars amb becaris, professors, i afins, m'han donat un entorn de treball òptim. Gràcies a la gent que s'ha llegit un esborrany d'aquesta tesi i encara són amics meus. Em sento privilegiat d'haver conegut aquest excel.lent grup d'investigadors i bons amics. També vull agrair l'esplèndid suport tècnic proporcionat per l'equip del LCAC i del CEPBA.

Un agraïment especial és per als meus amics de Vilanova, que tot i que mai no han entés el que estic fent per aquí, durant molts caps de setmana m'han estat realimentant per a aquestes setmanes de treball dur. Moltes gràcies al Meji, Juanki, Josep, Javi, Gabi, Juan Ramón, Panchi, Toni, Carlos, Janine, i més recentment a l'Ignasi. El meu agraïment és també per l'Emilio Gutiérrez, el meu millor amic de la Facultat, i per l'Oriol Valero, que em va deixar la seva casa durant un estiu i que comparteix molts dinars amb mi.

Finalment, vull agrair a la meva família, al meu germá i als seus fills, per el seu amor i suport, però en especial a la meva germana Neus, i a la meva nebodeta i fillola Cristina, que és la nena més bonica de tot el mon. El més profund agraïment, però, està reservat als meus pares, Daniel i Carme, que han fet possible que jo estés aquí, i que m'han donat el seu amor i suport incondicional durant tota la meva vida. Ells han fet aquesta tesi possible, i se la dedico a ells.

Aquest treball ha estat subvencionat parcialment per el Ministeri d'Educació Español amb els contractes TIC 880/92 i TIC 429/95, per la beca CIRIT BE94/1-100, per el CEPBA (Centre Europeu de Paral.lelisme de Barcelona), i per CONVEX Computer Corporation que ha col.laborat en el desenvolupament de l'eina DDT.

Al meus pares Daniel i Carme

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Automatic data distribution in the context of a parallelizing environment for massive parallel processors systems is a key topic of current research. In this Chapter we intend to provide the background of our research, covering architectural issues, the data parallel programming model, and assistant tools for parallelization. This introduction is followed by a motivating example that illustrates the importance of data distribution in a real program. Then, an overview of our environment is presented, which includes a description of the main modules of our tool. At the end of this Chapter we provide a Section introducing the main HPF data mapping features that will be used along the following Chapters, and a survey of current HPF compiler implementations.

1.1 BACKGROUND

High performance computing systems provide practical solutions to complex problems, with accuracies not possible some years ago. Their applicability includes fields such as weather modeling, environmental modeling and management, manufacturing design, biomedicine, molecular biology, computational chemistry or computational physics [HPC97]. For instance, weather models allow meteorologists to track severe storms and hurricanes with sufficient accuracy to implement more precise evacuation plans so as to save lives and property. Researchers use numerical simulations to estimate the effect of major storms and floods on the distribution of toxic materials in lakes and streams. Ground water contamination is a serious environmental and economic problem, with detoxification costs for existing sites estimated in the hundreds of billions of dollars. A simulation analysis of the flow about a Delta II rocket, demonstrated an anomaly that had sent a launch vehicle into the wrong orbit. The modeling of biological membranes allows researchers to design drugs that need to pass through membranes effectively. The combination of hardware technology, improvements in parallel languages and compilers, and development of high performance mathematical and scientific libraries, allows very large scale applications in science and engineering to be successfully executed.

To meet the high computational requirements of such applications, high performance computers are needed. Currently high performance computers can be constructed from available hardware components and produce machines that are highly reliable and efficient. *Massively Parallel Processor* systems (MPP) built from off-the-shelf powerful microprocessors are one of the current trends in affordable high performance computing systems. Tipically all processors in these systems work together to obtain higher speed in the program execution. The peak performance provided by these systems is the addition of the individual performance of each processor. However, programs for these machines are much more difficult to design, implement, and debug, than sequential programs. The programmer has to organize the concurrent execution of the program on the parallel machine. There are two main architectural trends towards the design of MPP: *Shared memory multiprocessors* and *Distributed memory multicomputers*.

Shared Memory Multiprocessor (SMM) systems connect several processors that share a single physical memory through a switching network or a shared bus. The memory access

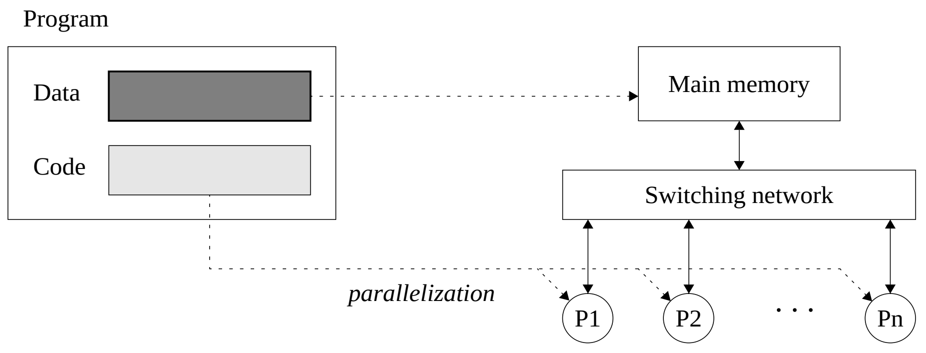


Figure 1.1 Shared memory multiprocessor architecture.

time is uniform for each processor. Each processor computes a part of the same program over a common data set stored on the main memory, as shown in Figure 1.1. With a small number of processors these systems compute efficiently, but in larger numbers the access to the main memory causes bottlenecks. The number of processors connected in a SMM system is heavily limited to the bandwidth of the interconnection network.

Distributed Memory Multicomputer (DMM) systems provide a cost-effective scalability to solve many large scale scientific problems. In DMM systems, each processor has direct access to its local memory, and indirect access to the remote memories of other processors through an interconnection network, as can be seen in Figure 1.2. The key drawback in DMM systems is that communicating data between processors can be more than one order of magnitude higher than the cost of accessing a local memory location. These systems are also called Non-Uniform Memory Access (NUMA), since the memory access time depends on the location of a datum in memory. Reducing the impact of the high memory latency may be achieved by restructuring the data in order to make more accesses local. A good data placement minimizes the remote memory accesses while keeping the parallelism inherent in the program. Unfortunately, the efficiency of the resulting program depends on many complex factors of the target machine, such as the processor topology, synchronization or inter-processor communication overhead.

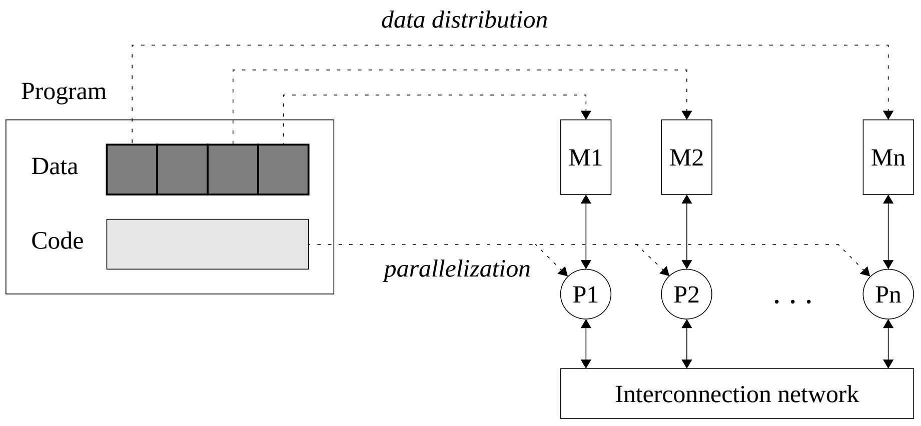


Figure 1.2 Distributed memory multicomputer architecture.

The first approach to program DMM machines is to use an existing sequential programming language enhanced with message-passing constructs. The programmer has to design the creation, synchronization and communication of parallel processes. This is achieved by distributing data and work over processors, and keeping track of the program execution at a very low level of detail. The references of data owned by other processors are satisfied by inserting appropriate message-passing statements in the code. Managing communication through low-level constructs is time consuming, error prone, and inhibits portability between different DMM machines. The resulting programming style can be compared to assembly programming on a sequential machine.

Data parallel programming languages provide code portability across both shared memory and distributed memory architectures. For this purpose, data parallel programs must be able to be compiled on different target machines, and to achieve reasonably high efficiency on different machines with the same number of processors. These languages allow the programmer to write code using global data references, but require the specification of data distribution over the individual memories of the physical processors. In the late eighties and early nineties, much research has been carried out to design and implement the first data parallel programming and compilation systems. Some early implementations such as Superb, Kali, and Booster are summarized in [CP95]. The main interest in data parallelism was generated by languages such as CM Fortran [TMC91], Fortran D [FHK$^+$90], and Vienna Fortran [CMZ92]. These languages not only produced features to assist the user with data alignment and distribution, but also implemented many of the features. These languages also preceded the effort in producing High Performance Fortran, or HPF [HPF93]. In fact, HPF was heavily influenced by the language designers and implementers involved in these languages.

Data distribution is then used to guide the compiler to generate a Single Program Multiple Data (SPMD) program [Kar87] for execution on the target distributed memory multiprocessor. Each processor executes the same program, but operates on different data. This is implemented by loading the same program image into each processor, and then, each processor allocates and operates on its own local portion of distributed arrays. The compiler has to translate global data references into local and non-local references based on the data mapping specified by the programmer. All non-local references must be satisfied by inserting explicit message-passing statements, usually respecting the owner computes rule [CK88], according to which it is the processor owning a data item that has to perform all computations for that datum.

Much work remains to be carried out in the development of these compilers, however the programmer still has to specify the placement of the data, and understand some features of the target machine. The choice of a good data distribution is important as it determines the amount of remote data accesses and the potential parallelism in the resulting program. The optimal data distribution depends on the program structure, the compiler capabilities, the characteristics of the target machine, and the program's data sizes. The layout of the data may be changed during program execution, this is known as remapping. Due to their influence on the amount of inter-processor communication, the choice of data mapping and remapping have a significant impact on the performance of the parallel program. In addition, there is often a trade-off between minimizing interprocessor data movement and load balancing on processors. Crucial aspects such as data movement, parallelism, and load balance have to be taken into consideration in a unified way to efficiently solve the data distribution problem.

Automatic data distribution tools can assist the programmer in this task. These may be source to source tools, which annotate the original high level program with data distribution directives and executable statements offered by data parallel languages. Automatic data distribution maps arrays onto the physically distributed memory of the processors according to the array access patterns and parallel execution of operations within computationally intensive blocks of code, named *phases*. If there is a single mapping for the whole program, then the solution is said to be *static*. However, in large problems where several computationally intensive phases occur, remapping actions between phases may improve the efficiency of the solution. In this case, the solution is said to be *dynamic*.

There are many other tools to support the development of parallel programs that can help the final success of High Performance Fortran. Run-time systems providing process management and execution control features, such as breakpointing facilities and instrumentation services. Parallel debugging tools providing views of different program levels. Performance analysis and monitoring tools for the identification and tuning of performance problems. Program restructuring tools that free the user from manually transforming the parallel application during the optimization process. Visualization tools and interactive graphic user interfaces that provide more comprehensive information. All these tools should be carefully integrated to provide a friendly and unified environment, useful enough to assist the programmer to write his parallel application codes efficiently.

1.2 MOTIVATING EXAMPLE

In this Section we intend to demonstrate the impacts of the program data sizes and the target architecture parameters on the data mapping selection. For this purpose we use the Alternate Direction Implicit (ADI) integration kernel, which defines a two-dimensional data space of size 256 in each dimension. It has a sequence of initialization loops, followed by an iterative loop that performs the computation. In each iteration of the loop, forward and backward sweeps along rows and columns are done in sequence. The ADI source code can be found in Appendix A.

The computational weight of the program is within the iterative loop, which contains 6 loop nests. In the first three loop nests, data movement takes place across the second dimension of the arrays, and parallelism can be exploited in their first dimension. Therefore a row distribution of the arrays may achieve a good performance. However, in the last three loop nests data movement takes place across the first dimension of the arrays, and the parallelizable loops traverse the second dimension of the arrays. In this case a column distribution of the arrays is advisable. This trade-off suggests several data mapping choices. The optimal choice depends on the program data size and certain target architecture parameters.

In order to illustrate the importance of choosing a good data mapping, we will consider the following five data distributions:

- Sequentialization of all arrays within a single processor. This data distribution does not benefit from the parallel capabilities of the target architecture, but will be considered for comparison purposes. For shorter, this strategy will be named *sequential*.

- Static one-dimensional data distribution of the first dimension of each array. This data mapping will be referenced *row* for shorter.
- Static one-dimensional data distribution of the second dimension of each array. This data mapping will be named *column* for shorter.
- Dynamic one-dimensional data distribution. In this case the first dimension of the arrays will be distributed in the first three loop nests within the iterative loop, and the second dimension of the arrays will be distributed in the last three loop nests within the iterative loop. This data mapping will be referenced *dynamic* for shorter.
- Static two-dimensional data distribution. Both array dimensions will be distributed over a two-dimensional processors grid, whose topology is assumed to be squared, assigning the same number of processors at each array dimension. This data mapping will be named *two_dim* for shorter.

The number of processors considered for this example is 16, and the data movement bandwidth assumed is 10^6 bytes/second. Profiling of the execution time for this code has been performed on a HP PA-RISC workstation. Table 1.1 shows the estimated speed-up of the ADI code with different data sizes. The original ADI code has been modified, declaring arrays of 64×64, 128×128, 256×256, and 512×512 elements.

	N = 64	N = 128	N = 256	N = 512
sequential	1.00	1.00	1.00	1.00
row	1.53	1.69	1.77	1.85
column	1.58	1.72	1.83	1.88
dynamic	2.39	2.48	2.58	2.74
two_dim	3.21	3.57	3.79	3.93

Table 1.1 Speed-up for the ADI code with different data sizes.

Note that with this code, all data mapping strategies improve speed-up when the data sizes are larger. Both the row and the column static distributions tend to reach a speed-up of 2. Speed-up in the dynamic data mapping is better than any static data mapping. However high remapping costs limits the speed-up growth. Finally, the two-dimensional data mapping strategy behaves better than the rest, although speed-up is also limited to 4.

The impact of the bandwidth of the target machine is much higher. We have compared the speed-ups of the column, dynamic, and two-dimensional data mapping strategies, with different bandwidths, ranging from 10^3 bytes/second to 10^9 bytes/second, and assuming 16 processors. Results are shown in the graphic of Figure 1.3. Note that with a low bandwidth, all data mapping strategies tend to a speed-up of zero. With a bandwidth of 10^6 the best data mapping strategy is the two-dimensional, although its speed-up is limited to 4. But with higher bandwidth, the dynamic data mapping strategy grows up to the optimal speed-up of 16.

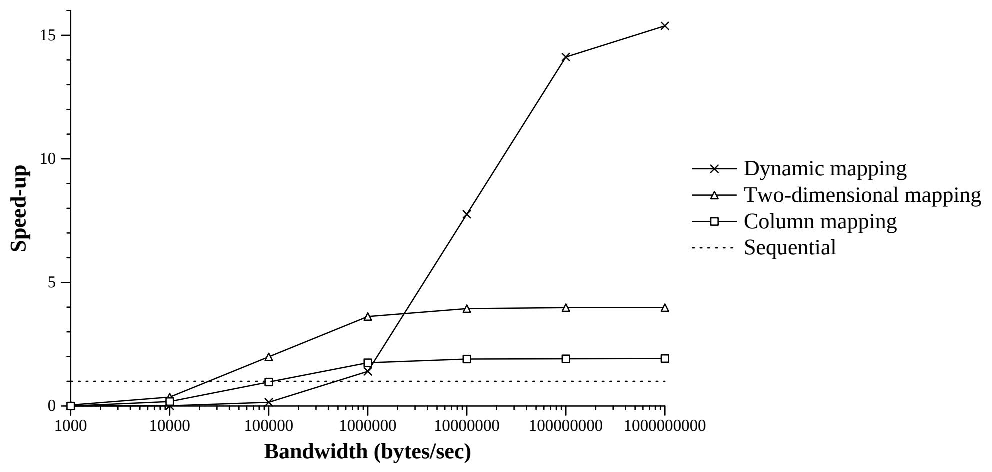


Figure 1.3 Speed-up for the ADI code with different bandwidths.

1.3 OVERVIEW OF OUR TOOL

Our thesis proposes a new framework for an automatic data mapping tool in the context of a parallelizing environment for Massive Parallel Processor (MPP) systems. The applications considered for parallelization are usually regular patterned problems in which data structures are dense arrays. The tool analyzes Fortran 77 programs and determines a data mapping and parallelization strategy for this program. This data mapping is used to annotate the original sequential Fortran program using HPF data mapping and loop parallelization directives. The data mapping strategy generated is optimal for a given problem size and target MPP architecture, according to our current cost and compilation model.

Profiling the sequential execution of the original Fortran 77 program is required in order to obtain some problem specific parameters, such as array sizes, loop bounds and execution time, and probabilities of conditional statements. The main steps performed during the optimization process are outlined below:

- *Program Analysis.* DDT [AGG$^+$97] acts as a preprocessing platform that parses the sequential program, and analyzes the code. The analysis includes reference patterns recognition and optimization, data dependence analysis and phase control flow graph construction.
- *Internal Data Structure Construction.* The data structure defined in our framework is the core of our tool. It contains all the information required to estimate the effects of any possible data mapping and parallelization strategy according to our model.
- *Optimization Problem Modeling.* The main internal data structure is used to model a minimal path problem. Linear 0-1 integer programming techniques are used to find the optimal solution.
- *Data Mapping Strategy Generation.* The output of a general purpose linear programming solver is captured and interpreted to generate the data mapping and parallelization strategy. The solution derived is optimal according to our model.

Profiling information is obtained using the APR Forge Explorer's performance profiler [APR95]. There exists a configuration file which allows the user to specify machine related parameters, such as number of processors and its grid topology, and interconnection network bandwidth. The main components in our environment, shown in Figure 1.4, are described next.

The parsing of the code is performed using the parser module of DDT, another research tool for automatic data distribution implemented by our group. DDT is built on top of ParaScope [KMT90], an interactive parallel programming environment developed at Rice

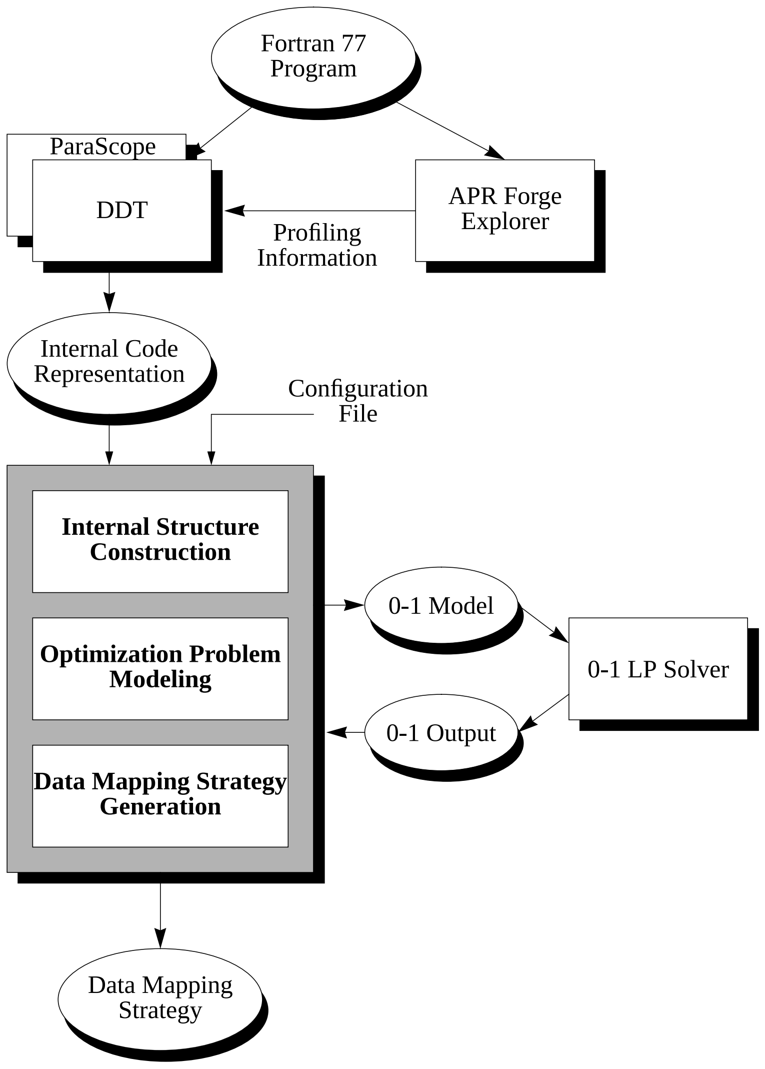


Figure 1.4 Main components in our optimization environment.

University. All reference patterns are obtained from the source code after performing a set of optimizations, such as expression substitution, subscript substitution, and induction variable detection, which improve the quantity and quality of the reference patterns analyzed [AGG+94].

All this information is used by our tool to generate the internal data structure which contains all the information regarding data movement and parallelism inherent in the original Fortran 77 program. This structure is used to model an optimization problem in which the optimal data mapping, according to our model, is obtained. The optimization problem is based on a minimal path problem with a set of additional constraints that ensures the correctness of the solution. The formulation of this problem as a linear 0-1 integer programming problem is written in a data file which contains the set of constraints and the objective function that has to be minimized.

LINGO [LIN94] is a general purpose linear programming solver used to find the optimal solution to the modeled problem. The solver generates an output data file with the value of each 0-1 integer variable. This file is interpreted by our tool to generate the final file containing information about the data mapping and the parallelization strategy suggested. This information can be finally used to annotate the original Fortran 77 source code with HPF data mapping and loop parallelization directives.

The data mapping strategy generated can be static or dynamic, one or two-dimensional, with either `BLOCK` or `CYCLIC` distribution fashion. The tool estimates the effects of control flow statements between phases, although when performing estimations of the cost of different strategies, we assume that the compiler does not perform any loop transformation or communication optimization. Inter-procedural analysis is not supported.

1.4 HIGH PERFORMANCE FORTRAN

The goal of HPF is to define a Fortran language extension to achieve high performance in MPP systems. Data parallel constructions allow data parallel programming (single threaded, global name space, loosely synchronous computation), data mapping features allow top performance in NUMA computers and multi-machine portability, and intrinsic functions and extrinsic procedures allow access to low level programming to tune the code in a specific architecture. In addition to the HPF specification, an official subset is also defined to allow earlier implementations. The more recent projects in this area, such as HPF2 or F90D are concerned with adding support to irregular computations and other less structured work and data distribution strategies.

In this Section we provide an overview of the HPF data mapping features that will be used along the remainder of this Thesis. In addition, a survey of current HPF compilers implementation is presented.

1.4.1 Data Mapping with HPF

The HPF directives are structured comments that suggest implementation strategies to the compiler. They may affect the efficiency of the program, although they do not change the program semantics. The program should generate the same results whether the directives are processed or not.

HPF includes data alignment and distribution directives that allow the user to specify how the compiler allocates data objects across the processors memory. The HPF compiler interprets these annotations and manages the data placement, minimizing data movement while retaining parallelism.

There is a three-level mapping of data objects to the physical memory. Data objects, typically array elements, are first *aligned* relative to one another. Alignment is performed across dimensions (inter-dimensional alignment) and within dimensions (intra-dimensional alignment). This group of aligned arrays is then *distributed* on an abstract grid of processors, according to a given pattern. Finally there is an optional mapping step from abstract to physical processors.

The `ALIGN` directive specifies relationships among array dimensions. Arrays are usually aligned onto a common array declared as the template. The `TEMPLATE` directive declares the name, size and dimensionality of a template. Arrays aligned to the same template are automatically aligned with each other. Inter-dimensional alignment is concerned with permuting the array dimensions with respect to the template. Intra-dimensional alignment allows, for each aligned dimension, shifting, striding, and reversing the ordering. For instance, in the following code:

```
        REAL A(8, 8), B(8, 8)
!HPF$   TEMPLATE T(8, 8)
!HPF$   ALIGN A(I, J) WITH T(I, J)
```

```
!HPF$  ALIGN B(I, J) WITH T(J-1, I+2)
```

the array A is exactly mapped onto the template, whereas the array B is transposed with respect to the template. In addition, the first dimension of B is shifted by $+2$ with respect to the template, and the second dimension of B is shifted by -1. This is shown in Figure 1.5.

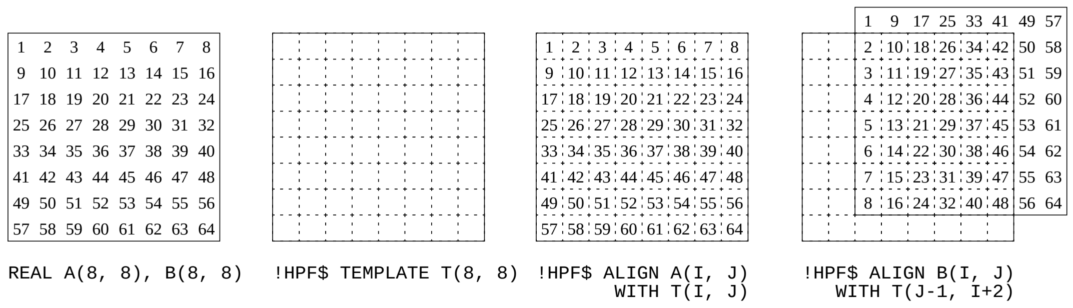

REAL A(8, 8), B(8, 8)
!HPF$ TEMPLATE T(8, 8)
!HPF$ ALIGN A(I, J) WITH T(I, J)
!HPF$ ALIGN B(I, J) WITH T(J-1, I+2)

Figure 1.5 HPF template and alignment directive examples.

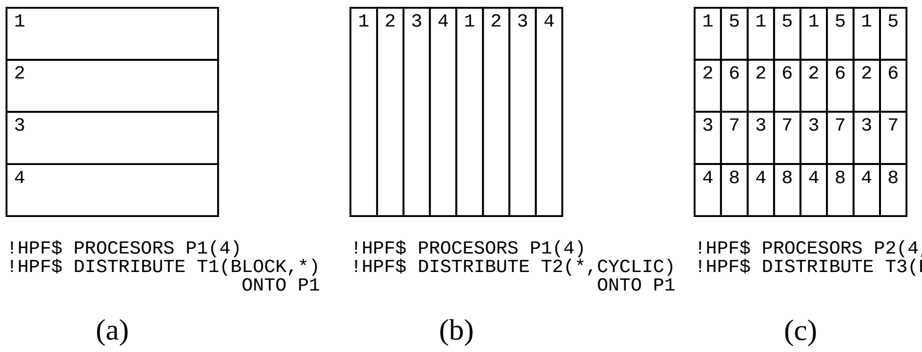
The `DISTRIBUTE` directive specifies a mapping of a group of arrays onto an abstract processors grid according to a given distribution fashion, either `BLOCK` or `CYCLIC`. A `BLOCK` distribution splits an array dimension by slicing it uniformly into blocks of contiguous elements. The `CYCLIC` distribution splits an array dimension in such a way that successive array elements are assigned to successive processors in a round-robin fashion. The `CYCLIC(m)` distribution, also known as *block_cyclic*, assigns blocks of m consecutive elements to each processor in a round-robin fashion. By definition, `CYCLIC(1)` means the same than `CYCLIC`. The symbol `*` is used to specify the dimensions that are not distributed. The group of arrays to distribute is usually specified by means of the template. In addition, the processor arrangement may be specified with the `PROCESSORS` directive, which declares its rank and the number of processors in each dimension. For instance, in the following code:

```
!HPF$  PROCESSORS P1(4)
!HPF$  TEMPLATE T1(8, 8), T2(8, 8)
!HPF$  ALIGN ···
!HPF$  DISTRIBUTE T1(BLOCK, *)  ONTO P1
!HPF$  DISTRIBUTE T2(*, CYCLIC) ONTO P1
```

the processors arrangement $P1$ is one-dimensional with 4 processors. The first dimension of template $T1$ is distributed in a blocked fashion, and the second dimension is sequentialized. All arrays aligned to template $T1$ are distributed the same way. Similarly, the second dimension of template $T2$ is distributed in a cyclic fashion, as shown in Figure 1.6.a and 1.6.b. Alternatively, in the next code:

```
!HPF$  PROCESSORS P2(4, 2)
!HPF$  TEMPLATE T3(8, 8)
!HPF$  ALIGN ···
!HPF$  DISTRIBUTE T3(BLOCK, CYCLIC) ONTO P2
```

the processors arrangement $P2$ is two-dimensional, with 4 processors assigned to the first dimension, and 2 processors assigned to the second one. In addition, the first dimension of template $T3$ is distributed in a blocked fashion across 4 processors, and the second dimension is distributed cyclically across 2 processors, as shown in Figure 1.6.c. Note that the first array dimension is distributed by chunks, and the second array dimension is distributed to successive processors in a round-robin fashion.



```
!HPF$ PROCESORS P1(4)
!HPF$ DISTRIBUTE T1(BLOCK,*)
                     ONTO P1
```

```
!HPF$ PROCESORS P1(4)
!HPF$ DISTRIBUTE T2(*,CYCLIC)
                     ONTO P1
```

```
!HPF$ PROCESORS P2(4,2)
!HPF$ DISTRIBUTE T3(BLOCK,CYCLIC)
                         ONTO P2
```

Figure 1.6 HPF processors and distribute directive examples.

An object can be remapped by *realigning* or *redistributing* it. In this case communication is required to move the data, in order to reflect the new mapping. Realignment is performed using the `REALIGN` directive, which is considered executable. Redistributing an object causes all objects aligned with it also to be redistributed to maintain the alignment relationships. This is performed using the `REDISTRIBUTE` directive, which is considered executable as well. Realignment and redistribution directives may appear only in the execution part of a program at any time, provided the objects have been declared *dynamic*, using the `DYNAMIC` directive.

For instance, in the following code fragment:

```
       REAL A(8, 8), B(8, 8)
!HPF$  PROCESSORS P(4)
!HPF$  TEMPLATE T(8, 8)
!HPF$  DYNAMIC A, B, T
!HPF$  ALIGN A(i, j) WITH T(i, j)
!HPF$  ALIGN B(i, j) WITH T(i, j)
!HPF$  DISTRIBUTE T(BLOCK, *) ONTO P
       ··· code ···
!HPF$  REDISTRIBUTE T(*, CYCLIC)
       ··· code ···
!HPF$  REALIGN A(i, j) WITH T(j, i)
```

both arrays A and B have been declared dynamic, as well as the template T. The redistribution directive causes all arrays aligned to the template to be dynamically redistributed, and the realignment directive causes array A to be realigned, transposing it with respect to the template. Remapping can require a lot of communication effort at run time, therefore the programmer must take care when using these directives.

There are many other HPF features not included in this Section, either related to data mapping or data parallelism. However they are not relevant to our work on automatic data mapping. A full HPF language description can be found in [HPF93].

1.4.2 Compilers for HPF

The HPF standards became publicly available by Spring of 1993. In the following years substantial progress has been made in finding solutions to many problems of implementing the language and data features defined in HPF. In addition to the HPF specification, an official subset has also been provided to allow earlier implementations. A wide range of systems are currently being implemented by various research groups and commercial companies.

Commercial HPF compilers have appeared for a broad range of architectures, from workstation clusters to massive parallel processor systems. However, many research groups

Compiler Name	Research Institution	Support
ADAPTOR	GMD - SCAI	F+
Annai Environment	CSCS - NEC	O+
The D System	Rice University	S+
HPFC	Ecole de Mines de Paris	S+
PANDORE	IRISA	S
PARADIGM	University of Illinois at Urbana-Champaign	S
shpf	University of Southampton	O+
vfcs	University of Vienna	S+

Table 1.2 List of currently available research compilers for HPF. F: full language definition; O: official subset; S: some features; +: additional features.

have targeted their efforts toward a more effective compilation of HPF programs, including in some cases features not included in the official definition of the language. Table 1.2 shows a list of some of the available compiler prototypes for HPF developed at research institutions or universities. HPF compilers presented in Table 1.3 are commercial. Both tables show whether the compiler handles the full definition of the language (F), the official subset defined for it (O), or just some features (S). They also show if new features (+)

Compiler Name	Company Name	Support
EXPERT HPF	Associated Computer Experts	O
Fortran 90 HPF	Digital Equipment Corporation	F+
HPF Mapper	N.A. Software	O
Paragon HPF	Intel Corporation	F-
pghpf	Portland Group Inc.	F-
TM/HPF	Thinking Machines Corporation	O+
VAST-HPF	Pacific Sierra Research Corporation	F-
xHPF	Applied Parallel Research	S
XL HPF	IBM Corporation	S

Table 1.3 List of currently available commercial compilers for HPF. F: full language definition; O: official subset; S: some features; +: additional features.

have been included in the standard definition. A more detailed study of these compilers can be found in [PAGT97] [1].

In summary, only one implementation is available for the full language, but it also includes extra features, namely, DEC Fortran 90 HPF. Some other implementations have omitted some features of the official HPF standard, like GMD-SCAI's Adaptor, Intel Paragon HPF, The Portland Group pghpf, or VAST-HPF. The same is true for the official subset with the Annai Tool Environment, Southampton's shpf, ACE Expert HPF, and Thinking Machines TM/HPF. The other implementations still have to implement some features for the official set or subset.

[1] This information is also available and updated at http://www.ac.upc.es/HPFSurvey/

2 RELATED WORK

In this Chapter we present the work related to automatic data distribution that we think that is the most representative on this topic. Some of this work has influenced on our research. In addition, an overview of previous research developed by our group is also given. From this experience, we started from the scratch with a novel approach towards finding an optimal data distribution and parallelization strategy. The main aspects of our work are described and compared to the related work, and the most important contributions of this Thesis are finally enumerated.

2.1 RELATED WORK

There is a large number of researchers that have addressed the problem of automatic data distribution in the context of regular applications. The main differences between the proposed methods is the kind of structure selected to represent the problem, the performance estimation model adopted, and the techniques applied to find a solution. Most of them split the static problem into two main independent steps: *alignment* and *distribution*. The alignment step tries to relate the dimensions of different arrays, according to the access patterns between them. A good alignment will minimize the overall overhead of inter-processor data movement. The distribution step decides which of the aligned dimensions are distributed, and the number of processors assigned to each of them. A good distribution maximizes the potential parallelism of the application, and offers the possibility of further reducing data movement by serializing. The main features of some systems are described below.

Li and Chen

Li and Chen [LC90, LC91], have been involved in the Crystal language and compiler project at Yale University. Crystal is a high level functional language, compiled for execution on a massively parallel distributed-memory machine. They address the generic problem of automatic data layout, although they concentrate on the problem of finding a static inter-dimensional alignment within a single phase. In addition, they have developed a methodology for communication synthesis in generating parallel programs for DMM systems.

On compilation, a Crystal program is first decomposed into code blocks, named phases, each of which is treated as a unit in the layout problem. The alignment and distribution problems are independently solved for each phase, and the results of different phases are finally merged.

They solve the alignment problem in two separate steps: the inter-dimensional alignment step, in which permutation and embedding are considered; and the intra-dimensional alignment step, in which shift and reflection is considered. The inter-alignment problem is modeled as a graph problem. The graph is called the Component Affinity Graph (CAG), and it is constructed from an analysis of the reference patterns in the source program. The nodes in the graph represent array dimensions, and are grouped by columns, where each column represents the components of the same array. An edge represents a preference for alignment between the connected dimensions. They assign weights to the edges of the CAG to reflect the strength of the preference, however their metric is reduced to weights of ε or 1. All edges between a pair of nodes are replaced by a single edge whose weight is the sum of the original weights.

For instance, consider the following code fragment:

```
do i = 2, N
        do j = 1, N
                A(i, j) = B(i-1, j) + 1
                B(i, j) = A(i, j) + 2
                C(j, i) = B(i, j) + 3
        enddo
enddo
```

with a loop nest, in which three reference patterns are identified:

$$A(i, j) \leftarrow B(i-1, j)$$
$$B(i, j) \leftarrow A(i, j)$$
$$C(j, i) \leftarrow B(i, j)$$

The reference patterns determine affinity relations between array dimensions. The generated CAG is illustrated in Figure 2.1. This consists of three columns with two nodes each. According to the affinity relations in the reference patterns, six edges are included, but only four of them are different. The graph contains global information about alignment preferences.

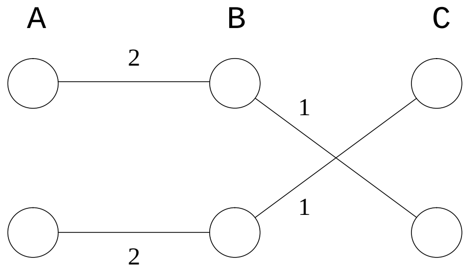


Figure 2.1 Example of a Component Affinity Graph.

The alignment problem is formulated as an optimization problem where edges in the CAG may carry different weights. They proved the alignment problem to be NP-complete in [LC90], so they present a heuristic algorithm to solve it. The algorithm is an enhanced greedy algorithm where a single array is chosen at each step for aligning with the target domain, and there is no back-tracking.

For the distribution step they propose to match the aligned reference patterns to a predefined set of data movement routines [LC91]. Each routine has an architecture-dependent cost parameterized in terms of number of processors involved in the data motion and the amount of data moved. The cost function for all the patterns is minimized by generating all possible distributions and selecting the most effective. They did not solve the dynamic problem, although their CAG has been used by many other researchers as the main data structure to solve the alignment problem.

Wholey

Wholey [Who92] at Carnegie-Mellon University developed a compiler for the ALEXI high level language. Communication and parallelism are expressed by using a small set of primitives, that manipulate elements of arrays in parallel. The ALEXI system breaks the data mapping problem into two parts, alignment and layout, as they assume the alignment problem to be architecture independent.

For the alignment step, he adopted the Preference Graph defined by [KLS90] in the framework of SIMD architectures, which allows the addition of parallelism related information while solving the alignment problem. However, its solution is based on heuristic techniques.

The execution time is estimated using a cost estimate function which is constructed from the source program in a straightforward way. Each primitive has an associated cost function which computes the running time given the descriptions of the mapping of its arguments. When the problem size is not known at compile time, default values are used instead of profiling information. The estimated time of a program is then computed as the addition of the estimated times of each individual array primitive.

In order to solve the layout step, he uses a hill climbing search method as an approximation. Initially all data is assigned to a single processor, and this cost is computed. Then the number of available processors is doubled and the new cost is computed. This process is iteratively performed until all available processors have been distributed, or until the total cost is not further reduced. Multi-dimensional distributions are supported, but only for communication purposes. Nested parallelism is not addressed, nor dynamic data layouts.

Gupta and Banerjee

Gupta and Banerjee [GB93] at the University of Illinois at Urbana-Champaign, implemented the PARADIGM compiler on top of the Parafrase-2 system, which is used to provide internal information about the program. They developed a methodology for automatic data partitioning given a sequential or shared-memory parallel program, generating SPMD programs with explicit communication.

They split the automatic data distribution process in four passes. The align pass maps each array dimension to a processor-mesh dimension. The block-cyclic pass determines for each array dimension, whether it should be distributed in a blocked or cyclic manner. The block-size pass determines the block size for each dimension distributed in a cyclic manner. And the num-procs pass determines the number of processors assigned in each of the processor-mesh dimensions, assuming that the maximal number of distributed dimensions is two.

They estimate computational costs from the estimation of the computation of a single instance of each statement, and the count of the number of times that statement is executed. They also consider synchronization delays. For the communication costs they perform an accurate analysis of data sizes, number of processors involved in the communication, where the communication would be placed, and number of iterations that the communication takes place.

During the alignment step, they use the CAG defined in [LC90], although their metric to weight edges is more accurate. They assume a default distribution, and for each edge, they try to estimate the extra communication cost incurred if those dimensions are not aligned. Unfortunately, it is not possible to find a single alignment configuration under which the communication costs may be compared to determine this *extra* communication cost. However, this proposal is more accurate than the previous one, although they use the same heuristic algorithm to solve the alignment problem.

For each aligned array dimension, they decide if the distribution fashion has to be block or cyclic. This is performed by estimating the penalty incurred in execution time if the distribution is block, and if the distribution is cyclic, and the compiler chooses the one with the higher value. If the distribution selected is cyclic, they obtain a block size for the corresponding dimension.

Finally, they decide the candidate dimensions to distribute, assuming again a default number of processors to be assigned in each dimension and estimating the total data movement plus computation cost. When more than one dimension is candidate to be distributed, they choose two of them and decide the number of processors to be assigned at each dimension by generating all possible combinations.

The final data partitioning generated is static, i.e. dynamic remapping is not supported. A solution for the dynamic approach has actually been proposed by Palermo and Banerjee in [PB95] using [GB93] to find the static mapping for each phase in the program. They propose a divide-and-conquer approach in which the program is recursively decomposed into a hierarchy of candidate phases. Then, estimating the costs of remapping between these candidate phases, a shortest path algorithm determines the final sequence of phases with the lowest cost.

Kremer and Kennedy

Kremer and Kennedy [KK95, Kre95] at Rice University, designed a framework to be used inside a data layout assistant tool for Fortran 77 that generates Fortran D or HPF. A prototype based on their framework has been implemented as part of the D system [ACG$^+$94].

Their framework for automatic data layout consists of four steps. First they partition the program into program segments named phases. Then a set of candidate layouts are built for each phase. In the third step, each candidate data layout and their possible data remapping are evaluated in terms of estimated execution time. And finally, based on these estimations, the optimal data layout strategy is selected.

As usual, they split the static data layout problem for a single phase in two stages: alignment and distribution. The first stage is based on the CAG, but they formulate the inter-alignment problem as a linear 0-1 integer programming problem, which provides an exact solution for their model. Instead of selecting the best alignment, a promising set of alignment candidates is selected, and the decision of selecting the final alignment is postponed until all distribution candidates are known.

Distribution analysis is performed after the alignment analysis. An exhaustive distribution search is performed, considering `BLOCK` and `CYCLIC` distribution. Once the distribution candidates have been determined, the cross product of alignment candidates and distribution candidates defines the candidate data layout search spaces for each phase. The user can limit the number of candidate data layout selected for each phase.

Each candidate data layout and each possible remapping action between candidate data layouts are evaluated in terms of expected execution time. Their approach to estimate performance is based on training sets [BFKK91]. A training set is a collection of kernel routines that measure the cost of several communication and computation patterns. The statements in the source code are matched to a previously defined training set, and the measured cost is used as the execution estimation.

As a result of the performance estimation step, estimated execution times for all candidate data layouts and possible remapping between layouts are available. The data layout search space is modeled with the Data Layout Graph, which has one node for each candidate data layout, and edges represent possible remapping between phases. Again, the problem is modeled as a minimal path linear 0-1 integer programming problem suitable to be solved by a state-of-the-art general purpose integer programming solver [BKK94b]. Kremer proved in [Kre93] that this exploration is NP-complete.

Anderson and Lam

Anderson and Lam [AL93] have implemented their algorithms in the SUIF compiler at Stanford University. They propose a methodology to for automatically decompose both data and computation onto a virtual processor array. Their main interest is to find the *shape* of the decompositions, and do not worry about load balancing, block size of cyclic distributions, or determining the number of physical processors to lay out in each dimension. In addition, as the target compiler used is SUIF, they can generate data decompositions that can not be specified by means of HPF directives.

Initially, the compiler apply unimodular transformations in order to leave loop nests in a canonical form (fully permutable), and positions parallel loops at the outermost level. They have developed a mathematical framework for expressing and calculating data and computation decompositions, and propose an iterative algorithm that looks for communication-free static decompositions for a single loop nest.

If communication-free decompositions can not be found, they propose to remove the constraints that force the sequential execution of a loop. This simplification requires to introduce nearest-neighbor communication, and will result in pipelined execution. The resulting decomposition is solved as a communication-free decomposition. Their communication cost estimation model is reduced to inexpensive and expensive data movements.

Finally, they describe the Communication Graph to model dynamic decompositions, and a heuristic algorithm to find the final dynamic decompositions. The algorithm tries to join loop nodes in order to eliminate possible remapping costs. Two nodes are merged if the performance of both nodes with the same decomposition is higher than the performance of each individual decomposition plus the remapping cost. They proved in [AL93] that the dynamic data distribution problem in the presence of control flow between loop nests is NP-hard.

Schreiber et al.

Schreiber et al. [CGS93, CGSS94] at the Research Institute for Advanced Computer Science and at Xerox Park, in the framework of the Excalibur project, propose a methodology for compiling array-oriented languages, such as Fortran 90.

They have developed a data flow representation called the Alignment Distribution Graph (ADG), consisting of ports, nodes and edges. Ports represent array objects, nodes represent program operations and edges connect definitions of array objects to their usage in these operations. In [CGS93] they solved the alignment problem using a dynamic programming approach, trying to minimize data movement costs. In [CGSS94] they use a divide-and-conquer approach to the dynamic mapping problem. They initially assign a static mapping valid for all the nodes and then recursively divide them into regions which are assigned different mappings. Two regions are merged when the cost of the dynamic mapping is worse than the static mapping considering computation, data movement, and remapping costs.

They also propose techniques in [SSGC95, SSP$^+$95] to reduce the graph complexity transforming the ADG into the Constraint Graph, and further simplifications by graph contraction techniques.

Other Proposals

There are several other researchers that in the last few years have developed some other new data distribution approaches. A summary of some of them is given as follows.

The FCS system [CP93] considers the problem in the framework of a data distribution tool for Fortran90 source codes. A phase is basically a DO-loop containing array-syntax assignment statements or `WHERE` masks in its body. Instead of looking for the optimal solution, they use a tree-exhaustive algorithm with some heuristics to prune the search space. A Conflict Table storing the conflicts between the mappings of the arrays from one phase to the other is the basis of the remapping algorithm. From this information, a tree showing all the different alternatives of remapping is built. The aim is to determine the path in the tree with the lowest cost. The full remapping tree can easily grow to intractable proportions.

In [CFZ93], a prototype to be included into their VFCS system is described, with the CAG as a basis for the alignment step. In the distribution step they propose as heuristic a bottom-up pass over the call graph to select a set of considered distributions for the arrays, and a top-down pass to select the final array distributions.

[DHR93] describe a parallelizing process, in which a methodology for data distribution is proposed. An exhaustive exploration tree of all possibilities is generated to solve the dynamic problem, and limits redistribution at the outermost loop level in order to reduce the complexity. Their cost model is based only in the data movement, and propose branch-and-bound to further reduce complexity.

[Fea93] considers the static distribution of data and computation among a two-dimensional processors geometry, in a DMM. The source code is restricted to assignments and `DO` loops. He constructs the Dataflow Graph that allows the representation of communication patterns in the source program. He concerns in minimizing communication, and seeks parallelism by scheduling statements. The algorithm used is a greedy algorithm based on the gauss-Jordan elimination.

[XN94] solve the alignment and distribution problems. During the alignment step they define the Alignment Graph, and propose a greedy algorithm as a heuristic to compute the graph. In the distribution step trapezoid distributions are considered as a universal distribution pattern to maximize processor load balance and minimize neighboring data movement.

[NDG95] as part of their EPPP system, describe a technique to decompose computation and data for distributed memory machines. The program is divided into collections of loop nests (Clustering Algorithm), and for each nest, decomposition and data locality constraints are formulated as a system of homogeneous linear equations (Locality Algorithm). Both algorithms are polynomial in complexity. They exploit several types of parallelism.

[Lee95] extends the CAG for two-dimensional alignments but uses the same heuristic to compute the graph. He proposes a dynamic programming algorithm that heuristically determines the dynamic data distribution for the full program. Some optimizations to hide data movement latency by data pipelining are described.

[KP96] define a weighted graph to solve the dynamic data distribution problem, taking into account several loop transformations, such as loop interchange and distribution. The graph includes communication and parallelism information. They formulate a minimal path algorithm, and perform an exhaustive search with pruning strategies to find the solution.

Some other authors [RS91, BKK$^+$94a] present methods to obtain communication-free data distributions using matrix notation, and solving the problem with linear algebra.

When communication-free partitioning of the arrays is not possible, they propose different problem formulations to minimize communication costs.

2.2 PREVIOUS WORK IN OUR GROUP

Some of the work referenced previously has influenced our research, however we obtained a lot of experience with the implementation of our first automatic data distribution tool (DDT and later PDDT), that derives inter-procedural dynamic multi-dimensional data distributions. Following there is a description the previous work developed by our group.

DDT

In 1992 our group started a research project to evaluate the quality of some methods which automatically derive data partitions from sequential code. The name of the project is *Automatic Data Distribution for the Convex MPP*, and was supported by Convex Computer Corporation and Convex Supercomputer S.A.E. The objective of the research project was the implementation and evaluation of two existing techniques for automatic data distribution: one based on the CAG by Li and Chen [LC90] and the other based on the communication-free approach by Huang and Sadayappan [HS91]. The platform chosen for the development of the tool was the ParaScope [KMT90] parallel programming environment developed at Rice University. In a first phase the group performed an exhaustive study and selection of application codes. Some results of this study have been published in [ALG$^+$93, ALG$^+$94, ALG$^+$95]. In a second phase we implemented both methods and obtained the first preliminary results. The approach by Li and Chen was improved, and its results have been published in [AGG$^+$94]. The tool, called Data Distribution Tool (DDT), was enhanced in order to support a wider range of distributions, redistributions, and to perform inter-procedural analysis [AGG$^+$95].

DDT analyzes Fortran77 programs and annotates them with directives and executable statements of HPF. The decisions are done so that the amount of remote accesses is reduced as much as possible, while maximizing the parallelism achieved. DDT is targeted to generic Non-Uniform Memory Architectures (NUMA) with local and remote memory accesses. Each processor has its own memory hierarchy and can access the memories in other processors through the interconnection network. Data movement costs are estimated as the number of remote accesses multiplied by the remote access time. Given a parallelization strategy, computation costs are estimated from a profile of the sequential execution on a workstation based on the same processor and with the same memory hierarchy than the parallel machine (which is a common fact in most of the hardware vendor product lines).

Profiling the sequential execution of the original Fortran 77 program is required in order to obtain some problem specific parameters, such as array sizes, the number of iterations for the loops and their execution time, and the probabilities of the different branches in conditional statements. There exists a configuration file that allows the user to specify some machine specific parameters (such as number of processors, overhead of parallel thread creation and local and remote memory access costs), and to restrict the kind of solutions explored by DDT (such as number of distributed dimensions and loops to parallelize, static or dynamic solutions or number of candidate mappings for phases and procedures). All cost estimations in DDT are performed numerically assuming the above mentioned problem and machine specific parameters.

The main steps of the unified data distribution and loop parallelization process performed by DDT are outlined below:

- *Program analysis.* ParaScope acts as a preprocessing platform that parses the sequential program into an intermediate representation and analyzes the code to generate flow, dependence and call graphs. DDT performs some optimizations in the analysis, such as expression substitution, subscript substitution, and induction variable detection, in order to obtain more information from the source program.
- *Detection of phases* or computationally intensive portions of code, which mainly correspond to nested loops and calls to procedures. A phase control flow graph is built, where nodes represent phases or control flow statements, such as loops surrounding phases or conditional statements, and edges connect nodes when there is a flow of control between them. Redistribution actions can only happen between these phases.

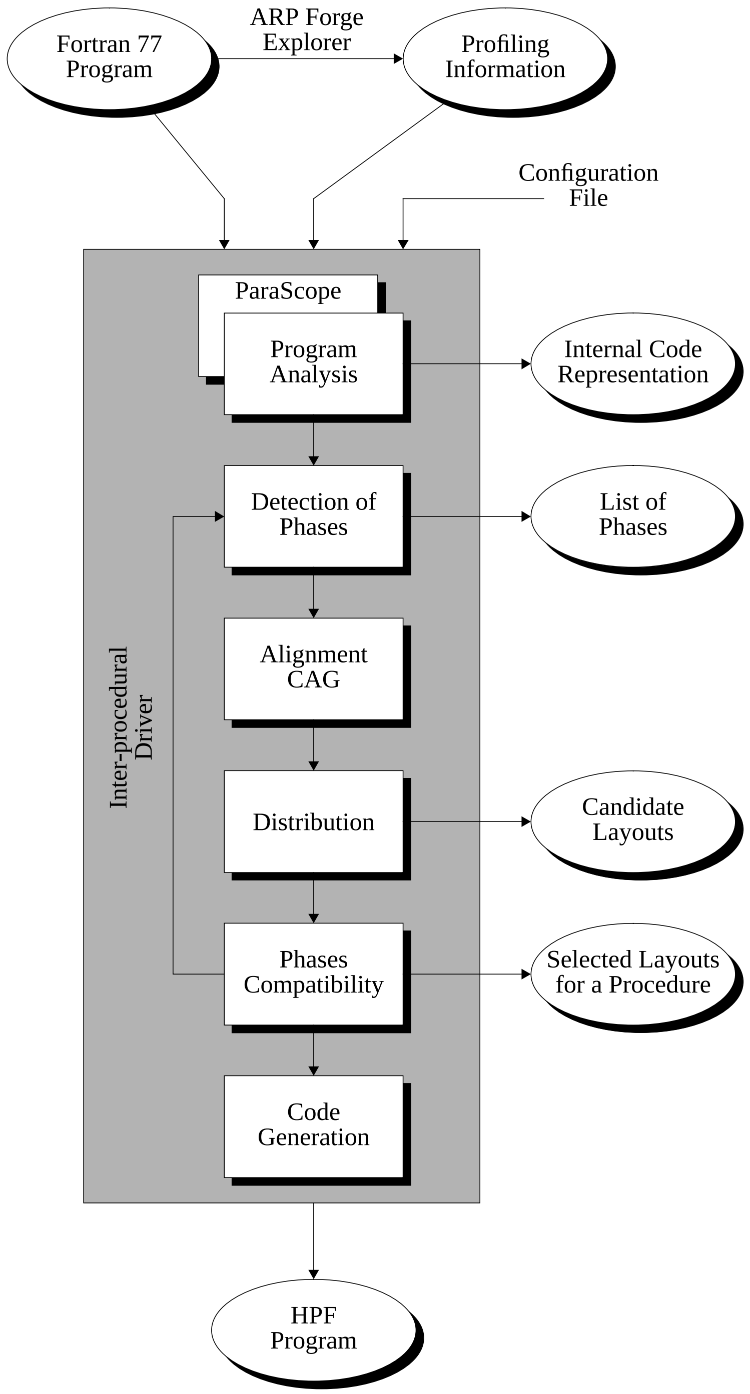


Figure 2.2 Main components of DDT.

- *Alignment* for each previously detected phase. This selection is performed based on an analysis of the reference patterns and data dependences within the scope of phases.

The inter-dimensional alignment is based on the Component Affinity Graph defined in [LC90], but improved in order to include parallelization information. The heuristic algorithm used has also been improved in order to obtain better alignments. Intra-dimensional alignment is also performed.

- *Distribution* of the set of aligned arrays. For the distribution step, an exhaustive exploration is performed. In this step, the data movement and parallelism costs are estimated. Each solution represents a particular distribution of the elements of the arrays and a parallelization strategy for the loops in the nest. The kind of solutions currently handled by DDT are `BLOCK` or `CYCLIC` multidimensional distributions. After this step, a set of candidate solutions has been generated for each phase.

- *Analysis of compatibility* among phases, and selection of solutions for each of them. This selection is done by analyzing the cost of phases in terms of data movement and computation time, and its benefits in the cost of successive phases. A recursive algorithm (that includes a cost function to control the pruning of the search space) explores the combination of the alternative solutions for each phase.

- *Code generation*. Generation of HPF directives to specify the data distribution strategies and parallelization directives that determines loops that are run sequentially or in parallel, according to the data distribution strategy. These directives are inserted to the Fortran 77 source program, generating a new HPF source file.

The process described above is done under control of the inter-procedural analysis module; this module builds the call graph for the entire program and records information about call sites and actual arguments. Once built, a bottom-up pass over the call graph decides the order in which procedures are analyzed, analyzes them and records information into the DDT inter-procedural database.

The native HPF compiler for the target machine is used to translate the annotated Fortran77 code generated by DDT into an efficient code, taking care of all the aspects related to scalar optimizations, further locality exploitation and proper storage of the arrays. All this work is summarized in [AGG$^+$97].

For instance, an outline of the HPF program generated by DDT as a result of analyzing the ADI program is shown in Figure 2.3. Statements in the loop bodies have been intentionally omitted for simplicity. The HPF directives inserted to the original source code can be seen in bold font.

```
   program adi
      double precision x(256,256)
      double precision a(256,256), b(256,256)
CHPF$    TEMPLATE TARGET(256,256)
CHPF$    ALIGN a(i,j) WITH TARGET(i,j)
CHPF$    ALIGN b(i,j) WITH TARGET(i,j)
CHPF$    ALIGN x(i,j) WITH TARGET(i,j)
CHPF$    DYNAMIC, DISTRIBUTE TARGET(BLOCK,*)
      do 1 i = 1,  256
         ...
1     continue
      do 2 j = 2, 255
         do 2 i = 1,  256
            ...
2     continue
      do 3 i = 1,  256
         ...
3     continue
      do 10 iter = 1, 10
C ADI forward & backward sweeps along rows
CHPF$    REDISTRIBUTE TARGET(BLOCK,*)
         do 4 j = 2,  256
            do 4 i = 1,  256
               ...
4        continue
         do 5 i = 1,  256
            ...
5        continue
         do 6 j = 255 , 1, -1
            do 6 i = 1,  256
               ...
6        continue
C ADI forward & backward sweeps along columns
CHPF$    REDISTRIBUTE TARGET(*,BLOCK)
         do 7 j = 1,  256
            do 7 i = 2,  256
               ...
7        continue
         do 8 j = 1,  256
            ...
8        continue
         do 9 j = 1,  256
            do 9 i = 255 , 1, -1
               ...
9        continue
10    continue
   end
```

Figure 2.3 Source code for ADI with HPF data mapping and remapping directives generated by DDT.

PDDT

In 1995 the project was extended with the name *Integrating Data and Work Distribution in the Exemplar Systems*. The goal is to apply the techniques developed for distributed memory multiprocessors to cache-coherent shared memory parallel systems. In these systems, cache miss penalties can be significantly large and false sharing, invalidations and excessive data replication can have negative effects in performance. In some cases, these effects can easily offset any gain due to parallel execution.

The motivation of this change is that the technology developed for distributed memory compilers can be useful for shared memory architectures in which each processor has access to a high-capacity private cache. In these systems, the cache behaves as an attraction local memory that stores data referenced by the processor. Trying to minimize true and false sharing reduces data motion through the interconnection network. The techniques we have developed represent the application of the owner computes rule, frequently used in distributed-memory systems, to shared-memory machines.

The name of this new tool is Parallelization and Data Distribution Tool (PDDT), which accepts programs written in Fortran 77, and generates code for either distributed network caches like the Globally Shared Memory Convex SPP systems [CON94], or private caches in bus based symmetric multiprocessor systems like the Power Challenge SGI systems [Sil96]. This work has been published in [AGGL96].

Most current shared-memory compilers choose a loop in each nest for parallelization, and it is interchanged as far out as data dependence analysis allows. Inner loops are strip-mined and blocked to exploit all possible data reuse in the processor cache. Iterations in each parallel loop are distributed across the parallel threads according to a fixed scheme. Some compilers also ensure that each major data structure in the program is aligned on a cache line boundary and make the contiguous dimension of an array (i.e., the first dimension in Fortran) an integer multiple of a cache line. This is useful to avoid false sharing of cache lines so that each processor works with complete cache lines.

In a parallel loop, a chunk of iterations is assigned to each processor. The execution of this chunk will bring any remote data to its cache. Notice that data remapping is implicitly done by the caching mechanism itself. We propose to parallelize loops taking into account the data that is stored in the private cache of each processor, either because it has been previously computed or fetched in other loops, or that needs to be stored in the cache because it will be useful in the following loops. PDDT keeps track of the array sections that are accessed during the execution of the different computational phases in an application in order to decide, with a global view, the parallelization strategy for each loop. This is done by analyzing the reference patterns inside computational phases and predicting the cache behavior that different parallelizations would imply. The generation of code for the target shared-memory programming models makes intensive use of well known techniques, such as loop tiling and loop limit adaptation to partition the iteration space, loop interchange to reduce the overhead of parallel thread creation and improve spatial locality, and parallel synchronized execution of dependent loops to minimize execution time.

In cache-coherent shared-memory systems, false sharing might also introduce additional data motion. Since data is transferred in cache lines, for instance 128 bytes long in SGI Power Challenge multiprocessors, different processors may share the same cache line and never access to the same data items. Every time a processor writes a data item in the line, other copies of the same line are invalidated. When another processor re-uses a data item (col-located on the same cache line), the item may no longer be in its cache due to the access by the other processor. Therefore, spatial locality may be lost and additional data movement may happen. PDDT also addresses the problem of minimizing false sharing by synchronizing the access to cache lines shared by different processors in parallel loops. In addition to that, PDDT also pads the contiguous dimension of arrays to make it multiple of cache line size and aligns major data structures to cache line boundaries. Other techniques oriented to the optimization of code for uniprocessor cache performance are left to the native compiler of the target parallel machine.

PDDT transforms the original source code in order to provoke the program to behave as assumed by PDDT. For instance, the transformed code for phases 4 and 7 of program ADI generated by PDDT can be seen in Figure 2.4. Note that phase 4 has been parallelized with pipelining to minimize false sharing. Phase 7 has been parallelized for chunk affinity and pipelined to minimize the sequentialization due to data dependences. The number of processors assumed is 8, and the chunk size is 4 in both phases.

PDDT is a research tool in the sense that it is flexible to specify machine dependent characteristics and to specify different compilation options and strategies. In addition to automatic parallelization, PDDT is also a performance prediction tool that may help the user in the task of writing parallel code for the target machine; it accepts directives in the source program which narrows the search space of solutions and provides the user with information about the behavior of the program.

```
    do 10 iter = 1, MAXITER
       ...
       do jj = 1, NUM, 2
          token(jj) = NUM/2
          token(jj+1) = 0
       enddo
C$PAR PARALLEL DO LOCAL(i, j, jj, my$p, lb$i, ub$i, next$p)
       do my$p = 0, 7
          lb$i = max((my$p * NUM / 8) + 1, 1)
          ub$i = min((my$p + 1) * (NUM / 8), NUM)
          next$p = 1
          do jj = 2, NUM, 4
444          if (next$p .gt. token(my$p+1)) goto 444
             do 4 j = jj, min(jj + 3, NUM)
                do 4 i = lb$i, ub$i
                   ...
4            continue
             token(my$p + 1) = token(my$p + 1) + 1
             next$p = next$p + 1
          enddo
       enddo
       ...
       do jj = 1, NUM
          token(jj) = 0
       enddo
C$PAR PARALLEL DO LOCAL(i, j, jj, my$p, lb$i, ub$i)
       do my$p = 0, 7
          lb$i = max((my$p * NUM / 8) + 1, 2)
          ub$i = min((my$p + 1) * (NUM / 8), NUM)
          do jj = 1, NUM, 4
777          if (token(jj) .ne. my$p) goto 777
             do 7 j = jj, min(jj + 3, NUM)
                do 7 i = lb$i, ub$i
                   ...
7            continue
             token(jj) = token(jj) + 1
          enddo
       enddo
       ...
10  continue
```

Figure 2.4 Transformed code for phases 4 and 7 of ADI generated by PDDT.

2.3 CONTRIBUTION OF THIS THESIS

Most automatic static data distribution approaches [LC90, Gup92, KK95, SSGC95, AGG$^+$94] decompose the data mapping process into two main steps: alignment and distribution. The alignment step finds appropriate relative alignments between all arrays in a block of code; for each array: (i) it decides the dimensions that will be combined with the dimensions of another array called the template (inter-dimensional alignment) and, (ii) for each aligned dimension, it decides the relative offset between the array and the template dimensions (intra-dimensional alignment). In [LC90] the authors proof that the inter-dimensional alignment problem is NP-complete. Once the alignment has been performed, the distribution step decides which dimension or dimensions of the template are distributed, the number of processors assigned to each of them, and the distribution fashion, i.e. `BLOCK` or `CYCLIC`. Alignment tries to minimize interprocessor communication and distribution tries to maximize the potential parallelism of the code, balance the computational load, and further reduce communication by serializing. However, these two steps are inter-related and should be important to examine the opportunities for parallelism when considering the alignment of the arrays.

In large programs where different computationally intensive phases occur, remapping actions between phases can increase the efficiency of the solution. In this case, a reasonably good solution is assumed independently for each phase, and remapping actions ensure that each phase executes with its solution. Note that the data mapping selection for one phase affects the decision of how to map the data and parallelize the loops in the next phase. Since it may be advantageous to select suboptimal mappings for some phases most dynamic data mapping methods proposed [CP93, BKK94b, SSP^{+}95, PB95, AGG^{+}95] consider a set of suboptimal mappings for each phase in order to perform the global analysis. However, this approach is a simplification, as long as only a subset of possible mappings are taken into account. In [Kre93] Kremer demonstrates that the optimal selection of a single mapping for each phase between a set of candidate mappings, is again NP-complete. And in [AL93] Anderson and Lam show that their model of dynamic data mapping problem in the presence of control flow statements between phases is NP-hard.

Our work has focussed on developing a novel approach towards automatically finding the best static or dynamic, `BLOCK` or `CYCLIC`, one-dimensional or two-dimensional data mapping strategy for a program, given the number of available processors and the problem size. The applications considered are regular problems, in which data structures are dense arrays. The solution generated takes into account the effects of control flow statements between phases, and includes the alignment and distribution of all the arrays at each phase, a set of remapping actions between phases if profitable, and the loop parallelization strategy induced by the data mapping.

The main contributions of our work can be summarized as follows:

- We have defined a new data structure named Communication-Parallelism Graph (CPG), that contains all the information required to consider all phases of the data mapping problem. This single data structure holds all data movement and parallelism information inherent in each phase of the program, plus additional information between phases denoting the data movement cost occurred if the distribution of one array in one phase is different than the distribution of the same array in another phase.

- The CPG allows us to easily estimate the performance effects of any selected mapping. This may be useful as a training tool, in which the user can obtain comprehensive information about the application, or identify performance problems.
- Due to the nature of the CPG, it can be extended in a straightforward way in order to support new features, such as sequentialization or replication of arrays in a phase, or some communication related optimizations. These features have to be tightly related to the capabilities of the target HPF compiler that will be used.
- The CPG can be modeled as a minimal path problem with a set of additional constraints that ensures the correctness of the solution, in which the parallel execution time of the program is the objective function to minimize. This is computed in a single step, which allows the alignment, distribution, and remapping problems for the whole program to be solved together. The solution found will be optimal globally instead of good solutions individually for each phase.
- Linear 0-1 integer programming techniques are used to find the solution to the minimal path problem. This technology guarantees that the solution provided is optimal, therefore avoids the use of heuristics while computing the solution. The idea of using linear programming techniques is based on the experience of Kremer et al. [BKK94b], although our problem structure and formulation is different than their model.

3 ONE-DIMENSIONAL DISTRIBUTION

In this Chapter we describe our framework to automatically generate one-dimensional data mappings. A valid data mapping strategy distributes at most one dimension of each array over the one-dimensional processors topology, either with a `BLOCK` or `CYCLIC` fashion. The distribution derived might be dynamic, and it takes into account the effects of control flow statements between computationally intensive code blocks. The number of processors p of the target architecture is assumed to be known at compilation time.

3.1 THE COMMUNICATION-PARALLELISM GRAPH

In our framework, we intend to define a single data structure able to represent the effects of any data mapping strategy allowed in our model. The name of this data structure is the Communication-Parallelism Graph (CPG), and it is the core of our approach. The CPG is a directed graph that contains all the information related to data movement and parallelism of the program under analysis. It is created from the analysis of all assignment statements within loops that reference, at least, one array.

The idea of the CPG is to have as many nodes as distributable array dimensions in the source program, therefore, each node represents one dimension of one array. The data movement and parallelism related information connect nodes when the distribution of such node has an impact on the performance. In order to estimate the effects of a given data mapping strategy, the corresponding array dimension for each array has to be selected.

All data movement and parallelism related information contained within the selected set of nodes, is considered for the parallel performance estimation.

In the following Sections the creation of the CPG is described, initially allowing just `BLOCK` distributions. The analysis is then improved in order to consider the effects of control-flow statements in the program, and the CPG is enhanced to support the generation of `CYCLIC` distributions. And finally, the interpretation of the information contained in the CPG is fully detailed. As a short working example along this Chapter to illustrate the creation of the CPG, the sample Fortran 77 code in Figure 3.1 will be used.

```
do i = 2, N
    do j = 1, N
        A(i, j) = B(i-1, j) + 1
        B(i, j) = A(i, j) + 2
        C(j, i) = B(i, j) + 3
    enddo
enddo
do i = 1, N
    do j = 1, i
        C(i, j) = D(i, j) + 1
        E(j, i) = D(i, j) + 2
    enddo
enddo
```

Figure 3.1 Sample code used to illustrate the components of the Communication-Parallelism Graph.

3.1.1 Nodes in the CPG

The first step in our framework is to decompose the program into computationally intensive code blocks named *phases*. Each phase will have its own data mapping strategy, and realignment or redistribution actions might be performed only between phases. The definition of phase is a topic of current research, and can lead to different solutions. In our approach we have adopted the following definition of phase made by [KK95]:

A phase is a loop nest such that for each induction variable occurring in a subscript position of an array reference in the loop body, the phase contains the surrounding loop that defines the induction variable. This operational definition does not allow the overlapping or nesting of phases.

Nodes in the CPG are organized in columns. There is a column in the CPG associated to each array in a phase. If one array is used in several phases, there will be a column for each phase in which this array appears. Each column contains as many nodes as the maximal dimensionality d of all arrays in the program. If the array has dimensionality $d' < d$, then the column is padded with $d - d'$ additional nodes.

Each column in the CPG represents all possible mappings of the array associated to it. At this point, only one-dimensional `BLOCK` distributions are allowed, therefore each node in a column represents one of the distribution choices for the array. Each node in a column will be mapped into one of the dimensions of the common array called template with dimensionality d. Additional nodes added for the arrays with dimensionality lower than the template are included to allow an embedding of the array on the template. Note that if the same array appears in two different phases, the CPG will have two different columns for this array, allowing the array to have a different mapping at each phase.

For instance, consider the sample code in Figure 3.1. The phases analysis detects that there are only two phases, each one surrounded by a loop i. Three arrays are used within each phase, say arrays A, B, and C in the first phase, and arrays C, D, and E in the second one. The maximal dimensionality d of the arrays is 2, so the CPG will have 6 columns with 2 nodes each, as can be seen in Figure 3.2.

Nodes are the basis of the CPG, over which the data movement and parallelism information is added in terms of *data movement edges* and *parallelism hyperedges.*

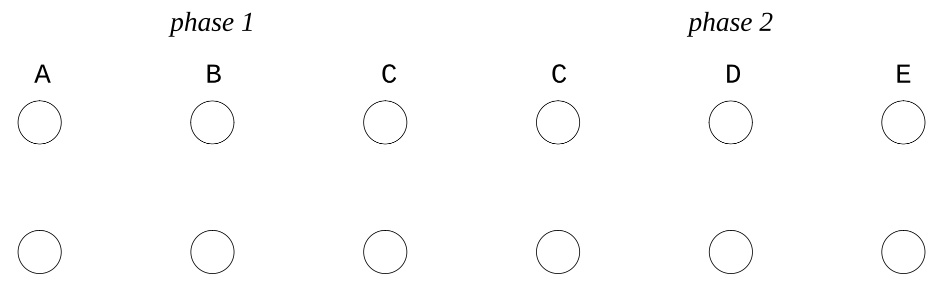


Figure 3.2 Nodes in the CPG.

3.1.2 Data Movement Edges

Data movement information is represented by means of edges that reflect possible alignment choices between pairs of arrays. An edge connecting two nodes represents the effects, in terms of data movement, of aligning and distributing these two nodes. This information is captured from the analysis of *reference patterns* within phases, and a data flow analysis between phases that derives the *realignment patterns*.

Reference Patterns

For each phase P_i in the program, the data movement information is obtained by performing an analysis of reference patterns between pairs of arrays within the scope of P_i. The meaning of reference patterns is defined in [LC90], and represents a collection of dependences between arrays in both sides of an assignment statement. When the same array is used in both sides, the reference pattern is called a self-reference pattern. For instance, in the first phase of the sample code of Figure 3.1, there are the following three reference patterns:

$$A(i,j) \leftarrow B(i-1,j)$$
$$B(i,j) \leftarrow A(i,j)$$
$$C(j,i) \leftarrow B(i,j)$$

For each reference pattern between two different arrays, $d \times d$ directed edges are added connecting each node associated to the right-hand side array of the assignment to each node associated to the left-hand side array of the assignment. These edges represent the behavior of all alignment alternatives between dimensions of these two arrays. The weight that is assigned to each edge is a symbolic expression that represents the data movement cost to be carried out if these two dimensions are aligned with respect to each other and distributed. And the edge direction indicates the direction of the data flow. When a self-reference pattern is found, d self edges are added in the column associated to the referenced array, one in each node. As in the previous case, the weight assigned to each edge is a symbolic expression that represents the data movement cost to be carried out if this dimension is distributed. Several edges between a pair of nodes are replaced by a single edge with weight equal to the sum of the original ones. In Section 3.3.1 the data movement cost model assumed in our framework is fully described.

For instance, the first reference pattern of the sample code of Figure 3.1 connects all nodes in column B to all nodes in column A. The edge that connects the first dimension of array B to the first dimension of array A (written $A[1] \leftarrow B[1]$) is labeled with a symbolic expression representing a *One_to_One* data movement primitive, as this is the expected data movement performed if these two dimensions are aligned and distributed. The edge $A[1] \leftarrow B[2]$ is labeled with a symbolic expression representing a $Many\_to\_Many$ data movement primitive, as this means that if these two dimensions are aligned and distributed, a data movement involving all processors will be performed while executing the loop. Similar analyses can be applied to all other edges in the reference pattern. The resulting CPG filled with data movement information is shown in Figure 3.3. Dotted edges represent $Local\_Memory\_Access$, which implies that, if the corresponding two dimensions are aligned and distributed, the required memory accesses to execute this statement are local.

Note that the CPG when filled only with data movement information, is different than the *Component Affinity Graph* (CAG) used by other authors [LC90, Gup92, KK95]. The meaning of the edges in the CAG is a preference for alignment, this is, how good is the alignment of two dimensions. Their weight should reflect benefits in terms of data movement if the corresponding dimensions are aligned. However, it is not possible to identify a unique alignment configuration for the whole CAG, under which the data movement cost may be estimated and compared to determine these benefits. In our framework,

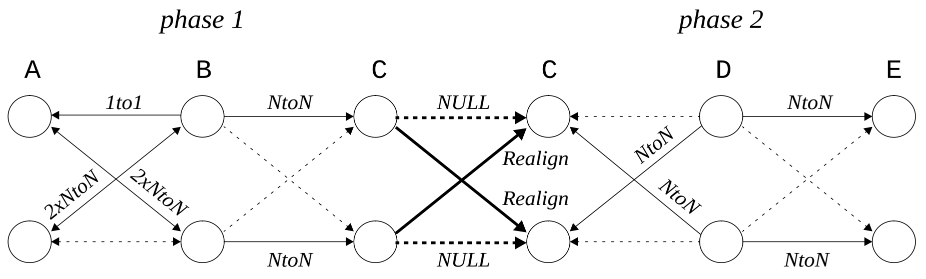


Figure 3.3 CPG with data movement information.

the meaning of an edge is just the opposite, this is, how expensive is, in terms of data movement, to align and distribute the corresponding dimensions. This cost is computed in seconds, therefore it is independent of any other alignment strategy.

Realignment Patterns

Realignment information is included in terms of data movement edges, and allows realignment actions of an array in a sequence of phases. Note that arrays may only be realigned between phases, therefore the data flow analysis detects if an array A in a phase P_i is used in a later phase, say P_j. In this case, $d \times d$ edges are added connecting each node of the column associated to array A at phase P_i to each one associated to array A at phase P_j. The weight of each edge is a symbolic expression representing the data movement cost incurred if the corresponding dimensions are aligned and distributed. When the array keeps the same alignment across these phases, the weight of the edges connecting the same dimension of the array between phases P_i and P_j is null. Otherwise the weight is the cost of a *realignment* data movement primitive.

The addition of the realignment information in terms of data movement edges, allows our model to easily combine both types of data movement information within the same data structure. Section 3.1.4 describes how to analyze realignment in a sequence of phases with presence of control flow constructions between phases, such as iterative loops or conditional statements.

For instance, the data flow analysis of the sample code of Figure 3.1 detects that array C is used in both phases, so a new set of edges allowing different realignment for this array is added, as shown with thick edges in Figure 3.3. The expression associated to the edges that connect the same dimension of array C in both phases is $NULL$. The other edges have a *Realignment* data movement primitive associated to them.

3.1.3 Parallelism Hyperedges

Parallelism information is obtained after performing a data-dependence analysis [KMT91]. All loops that can be executed in parallel are detected and marked as candidate parallel loops. In distributed memory machines a loop can be fully parallelized if it does not carry any data flow dependence [Tse93]. According to the owner computes rule [CK88], the processor that owns a datum is the one that performs all computations to update it. Therefore if a loop has to be parallelized, the array dimensions subscripted by the loop control variable of all arrays accessed in the left hand side of each assignment statement inside the loop, have to be distributed.

The parallelism related information is added in the CPG in terms of hyperedges. A hyperedge is a generalization of an edge as it can connect more than two nodes. Each candidate parallel loop has a hyperedge in the CPG, that links all the array dimensions that have to be aligned and distributed for the loop to be parallelized. In other words, our tool searches for each assignment statement inside a candidate parallel loop, if an array in the left hand side is subscripted by the loop control variable of the loop, then the node associated to the corresponding array dimension is linked to the hyperedge. For the candidate loop to be parallelized, all nodes connected by its hyperedge have to be aligned and distributed. The weight of a hyperedge is a symbolic expression representing the computation cost saved when the corresponding loop is parallelized. Therefore this weight may be considered a benefit, as opposite to a cost. In Section 3.3.2 the computation cost model assumed in our framework is fully described.

For instance, the data dependence analysis in the sample code of Figure 3.1 reveals that the j loop in the first phase, and the i and j loops in the second one could be parallelized, so three hyperedges are added to the CPG. The first one is associated to the first j loop, therefore it links the second dimension of arrays A and B, and the first dimension of C in the first phase, and it is weighted with a symbolic expression function of the execution time saved if the first j loop is finally parallelized. The second hyperedge corresponds to the i loop in the second phase, and it links the first dimension of array C and the second dimension of array E. Similarly, the weight associated to this hyperedge is a symbolic function of the computation time saved if the associated loop is parallelized. And finally the third hyperedge, associated to the j loop in the second phase, links the second dimension of array C and the first dimension of array E. In Figure 3.4 the CPG filled only with parallelism information is shown.

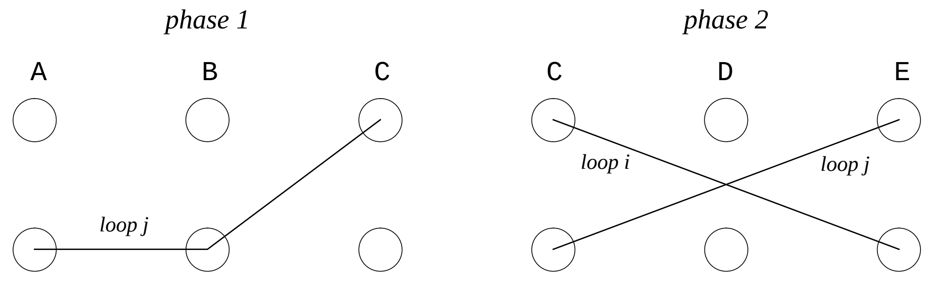


Figure 3.4 CPG with parallelism information.

3.1.4 Control Flow Analysis

Control flow statements between phases are taken into account when performing the data flow analysis. Entry or exit points, conditionals or iterative statements can modify the execution flow of a program, and can provoke a sequencing of the phases in the program different than the lexicographic order. The *Phase Control Flow Graph* (PCFG) records information about the sequencing of phases: nodes are phases, and edges link nodes when there is a flow of control between them. From the information in the PCFG, the different sequences of phases that might appear during the execution of the program are generated. These sequences determine pairs of phases between which remapping actions may be necessary.

Conditional Statements

Conditional statements generate alternative phase sequences that are executed depending on the condition evaluated in the statement. The probability of taking each alternative branch is obtained by profiling, and it multiplies the costs of phases and realignment edges in the branch.

```
phase1
if ( condition ) then
        phase2

        phase3
else
        phase4
endif
phase5
```

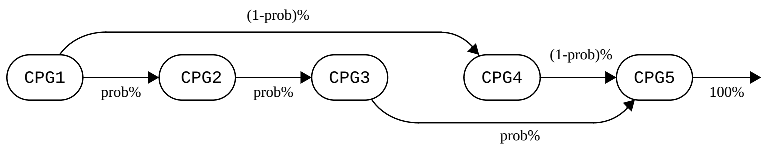


Figure 3.5 Sample code with a conditional statement, and its phase control flow analysis.

For instance, consider the code fragment in Figure 3.5 which contains a conditional statement surrounding several phases. According to the phase control flow graph, if the condition evaluates $TRUE$, $phases2$ and $phase3$ will be executed after $phase1$. Alternatively if the condition evaluates $FALSE$, $phase4$ will be executed after $phase1$. Assume that the probability of evaluating $TRUE$ is $prob$, therefore the probability of evaluating $FALSE$ is $1-prob$. There are two possible paths in this code: one is $1-2-3-5$ with probability $prob$ and the other is $1-4-5$ with probability $1-prob$. The data flow analysis will add realignment information between arrays used in two consecutive phases according to the paths generated. The costs of the remapping patterns are weighted up with the probability of the corresponding path, as well as the data movement and parallelism costs of the phases within the path.

Note that an initial phase has to be added to guarantee the array mapping coherency for the arrays used in both $phase2$ and $phase4$, but not in $phase1$. The initial phase contains all arrays used in the program, and ensures, for each array, that there is a single initial mapping.

Iterative Loops

Iterative loops or loops whose loop control variables (or induction variables generated by them) are not used to subscript arrays, provoke a sequence of phases different than lexicographic: after executing the last phase inside the loop, the first phase inside it is executed again. Remapping actions might be necessary between phases that are not in the lexicographic order. The number of iterations of this outer loop, say N, is used to compute the number of times that each phase is executed and to weight up its cost according to this number. The cost of all remapping edges between phases is multiplied by N except for backward remapping edges that are multiplied by $N - 1$.

When an outer loop is found in the source code, the analyzer generates a sequence of phases that try to represent what happens during the actual execution. The body of the outer loop is duplicated. Phases in the first sequence are assumed to be executed once, while phases in the second one are assumed to be executed $N - 1$ times. This technique can be applied to any random nest of loops.

```
phase1
do it = 1, 40
       phase2

       phase3
enddo
phase4
```

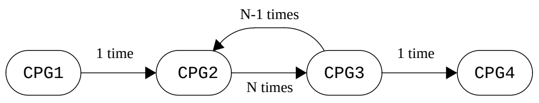


Figure 3.6 Sample code with an iterative loop statement, and its phase control flow analysis.

For instance, the code fragment in Figure 3.6 contains an iterative statement. Assuming that the outer loop iterates 40 times, remapping between *phase*2 and *phase*3 has to be accounted for 40 times, while backwards remapping between *phase*3 and *phase*2 has to be accounted for 39 times. In addition, each phase is assumed to be executed 40 times.

Nesting Control Flow Statements

Both control flow statements could be randomly nested. The effects of several nested conditionals is that a conditional tree with several branches is built, and different probabilities are assigned to each branch. When several loops are nested, the same process described above proceeds from the innermost loop upwards.

For instance, consider the sample code from Figure 3.7. It consists of a conditional statement between *phase*1 and *phase*6. If the condition evaluates $TRUE$, then two nested loops will be executed. Otherwise, another conditional statement will be executed. Assume that the probability of evaluating $TRUE$ in the first conditional is *prob*1 and the probability in the second conditional is *prob*2. Assume also that the first loop is executed $N1$ times, and the second loop is executed $N2$ times. The analysis indicates that *phase*2, *phase*3, and *phase*4 will be executed with probability *prob*1; *phase*5 with probability $(1 - prob1) \times prob2$; and the above phases will be skipped with probability

```
phase1
if ( cond1 ) then
       do it1 = 1, N1
              do it2 = 1, N2
                     phase2

                     phase3
              enddo
              phase4
       enddo
else
       if ( cond2 ) then
              phase5
       endif
endif
phase6
```

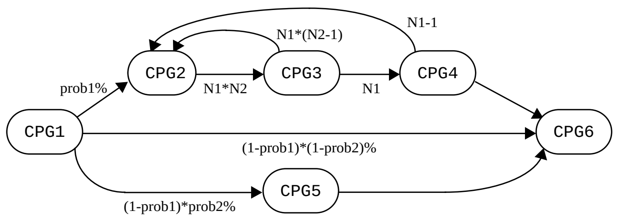


Figure 3.7 Sample code with conditional statements and iterative loops and its control flow analysis.

$(1 - prob1) \times (1 - prob2)$. We can also see that $phase3$ will be executed $N1 \times N2$ times after $phase2$, that $phase2$ will be executed $N1 \times N2 - N1$ times after $phase3$, and so forth, as seen in Figure 3.7.

Note that it is not possible to predict the behavior of a conditional statement inside an iterative loop, as any of both branches may be randomly taken during different iterations of the loop. In our framework the control flow analysis is simplified, and we assume that the evaluation of the conditional holds across all iterations of the loop. The analysis performed generates a conditional tree with two independent branches, one for each case in the conditional.

3.1.5 `CYCLIC` Distributions

Some applications perform computations over a triangular iteration space. According to the owner computes rule, a triangular loop parallelized assuming a `BLOCK` distribution lead to a poor load balancing, and therefore loss of performance. In Figure 3.8.a there is an example of a triangular loop partitioned by blocks. The amount of data elements to be computed by processor 1 is, in this case, seven times the amount of elements to be computed by processor 4. Alternatively, a `CYCLIC` distribution balances the computational load in triangular loops. As can be seen in Figure 3.8.b, all processors have to compute a similar amount of data.

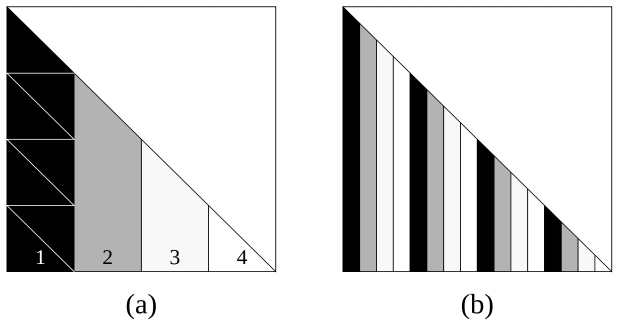


Figure 3.8 Effects of (a) `BLOCK` and (b) `CYCLIC` distributions in triangular loops.

However, if neighbor communication patterns appear in the loop, a `CYCLIC` distribution incurs in excessive data movement, as all processors have to send the whole array to its neighbor one. This is illustrated in Figure 3.9.b. In this case, `BLOCK` distributions reduces the amount of data to be moved. This may result in a trade-off between `BLOCK` and `CYCLIC` distributions to be considered during the decision of the data distribution.

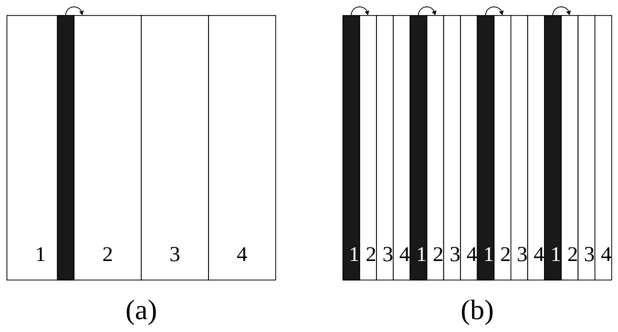


Figure 3.9 Effects of (a) `BLOCK` and (b) `CYCLIC` distributions in neighbor communications.

In our framework by default, if the code does not contain any triangular loop, the distribution assumed is `BLOCK`. Alternatively, if the code contains triangular loops and it does not contain any nearest neighbor like communication, then the `CYCLIC` distribution is assumed. In case of conflict, the CPG is duplicated in order to represent both alternatives. In the first copy of the CPG, named CPG_{BLOCK}, all costs are estimated assuming a `BLOCK` distribution; in the second one, named CPG_{CYCLIC}, all costs are estimated assuming a `CYCLIC` distribution. Now, the distribution choices for each array in each phase include both the `BLOCK` and `CYCLIC` distributions. In addition to the realignment information between phases in a single copy of the CPG, another set of redistribution edges connecting arrays in phases of different copies of the CPG has to be included, in order to allow arrays to change their distribution fashion between phases. The weight of this set of edges represents the cost of changing the distribution fashion, i.e. a redistribution. The model does not allow both `BLOCK` and `CYCLIC` distributions to be mixed in a single phase for those arrays related with data movement edges or parallelism hyperedges.

For instance, consider again the sample code in Figure 3.1. There is a nearest neighbor communication in the first statement of the first phase, and the second phase generates a triangular iteration space. In this case, the CPG is duplicated as shown in Figure 3.10. Note that although the information is apparently duplicated, the costs of CPG_{BLOCK} have to be computed assuming a `BLOCK` distribution, and the costs of CPG_{CYCLIC} have to be computed assuming a `CYCLIC` distribution. In addition, there is a set of edges connecting array C in the first phase of CPG_{BLOCK}, to array C in CPG_{CYCLIC}, and vice versa. Their weight is a symbolic expression representing the cost of changing from `BLOCK` to `CYCLIC` distribution, i.e. a redistribution.

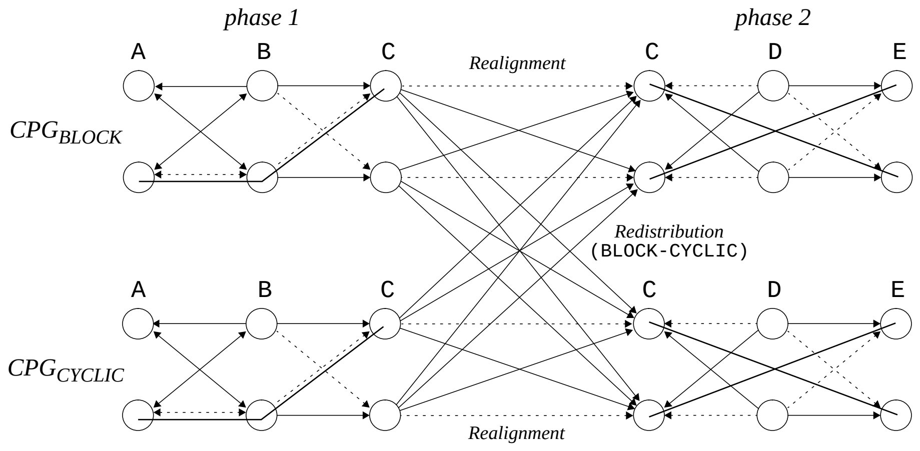


Figure 3.10 CPG with `BLOCK` and `CYCLIC` information.

3.2 ALIGNMENT AND DISTRIBUTION IN THE CPG

The CPG contains all the information required to estimate the performance effects of the program for different mappings, in terms of data movement and parallel loops inside each phase, and realignment and redistribution actions between phases. A *valid mapping* strategy has to include a node for each column, which determines the array dimensions distributed for each array in each phase. The estimated data movement cost for the selected mapping strategy is determined with the set of edges that remain inside the selected set of nodes. The estimated computation saving time due to parallelization for the selected mapping strategy is determined with the set of hyperedges whose nodes are all included within the selected set.

For instance, to estimate the behavior of the sample code of Figure 3.1 when distributing the first dimension of each array in each phase, all edges and hyperedges connecting nodes corresponding to the first dimension of the arrays have to be considered, as can be seen in Figure 3.11. The HPF data mapping directives that generate this mapping strategy are the following:

```
!HPF$  PROCESSORS PROC_GRID(P)
!HPF$  TEMPLATE T(N, N)
!HPF$  ALIGN A(i, j) WITH T(i, j)
!HPF$  ALIGN B(i, j) WITH T(i, j)
!HPF$  ALIGN C(i, j) WITH T(i, j)
!HPF$  ALIGN D(i, j) WITH T(i, j)
!HPF$  ALIGN E(i, j) WITH T(i, j)
!HPF$  DISTRIBUTE T(BLOCK, *) ONTO PROC_GRID
```

where N is the matrix sizes, and P is the number of available processors in the one-dimensional processors grid.

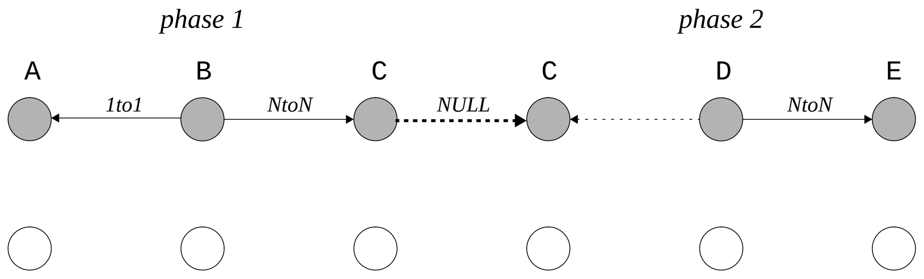


Figure 3.11 Legal mapping in the CPG for the sample code.

The total data movement cost for this mapping is summarized with the addition of the costs of the edges belonging to this set. Similarly, the computation time saved due to parallel loop execution is summarized with the addition of the weights of the hyperedges that belong to this set. According to this example, no candidate parallel loops can be parallelized. In addition, there is a *One_to_One* data movement primitive flowing from array B to array A, and a $Many\_to\_Many$ data movement primitive flowing from array B to array C in the first phase. In the second phase there is another $Many\_to\_Many$ data movement primitive from array D to array E. This mapping strategy leads to a poor performance, as there is no parallelism, and some data has to be moved.

A good solution for the sample code in Figure 3.1 is to align and distribute the second dimension of arrays A and B, and the first dimension of array C in the first phase. In the second phase the first dimension of arrays C and D, and the second dimension of array E have to be aligned and distributed. This mapping strategy is illustrated in Figure 3.12, and the HPF data mapping directives are specified below:

```
!HPF$  PROCESSORS PROC_GRID(P)
!HPF$  TEMPLATE T(N, N)
!HPF$  ALIGN A(i, j) WITH T(j, i)
!HPF$  ALIGN B(i, j) WITH T(j, i)
!HPF$  ALIGN C(i, j) WITH T(i, j)
!HPF$  ALIGN D(i, j) WITH T(i, j)
!HPF$  ALIGN E(i, j) WITH T(j, i)
!HPF$  DISTRIBUTE T(BLOCK, *) ONTO PROC_GRID
```

In this case the solution is static, there is no data movement during the execution of the phases, and loops j in the first phase and i in the second one can be parallelized. Note that the parallelism achieved is maximal, assuming one-dimensional distributions.

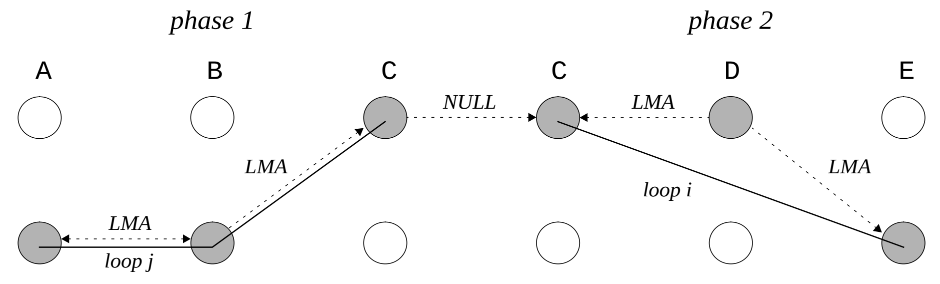


Figure 3.12 Optimal mapping in the CPG for the sample code.

If the CPG has been duplicated in order to model both BLOCK and CYCLIC distributions, the definition of a valid mapping strategy stands. Note that in this case a node for a column includes both the BLOCK and CYCLIC CPG copies. Similarly, the performance effects in terms of data movement and loop parallelization results in the set of edges and hyperedges that remain within the selected set of nodes. For instance, assuming the code in Figure 3.1, the effects of aligning and distributing the first dimension of all arrays in the first phase with a BLOCK distribution, and aligning and distributing the second dimension of all arrays in the second phase with a CYCLIC distribution is illustrated in Figure 3.13, and the HPF data mapping directives specifying this mapping strategy are the following:

```
!HPF$  PROCESSORS PROC_GRID(P)
!HPF$  TEMPLATE T1(N, N), T2(N, N), T3(N, N)
!HPF$  DYNAMIC T2
!HPF$  ALIGN A(i, j) WITH T1(i, j)
!HPF$  ALIGN B(i, j) WITH T1(i, j)
!HPF$  ALIGN C(i, j) WITH T2(i, j)
!HPF$  DISTRIBUTE T1(BLOCK, *) ONTO PROC_GRID
!HPF$  DISTRIBUTE T2(BLOCK, *) ONTO PROC_GRID
!HPF$  ALIGN D(i, j) WITH T3(i, j)
!HPF$  ALIGN E(i, j) WITH T3(i, j)
!HPF$  DISTRIBUTE T3(*, CYCLIC) ONTO PROC_GRID
       ··· 1st phase ···
!HPF$  REDISTRIBUTE T2(*, CYCLIC) ONTO PROC_GRID
       ··· 2nd phase ···
```

Note that the initial distribution of arrays A, B, and C is `BLOCK` and the initial distribution of arrays D, and E is `CYCLIC`. If we use template $T1$ to align array C, the redistribution directive will redistribute arrays A, B, and C. However only array C has to be redistributed, therefore we use a different template $T2$ for this purpose.

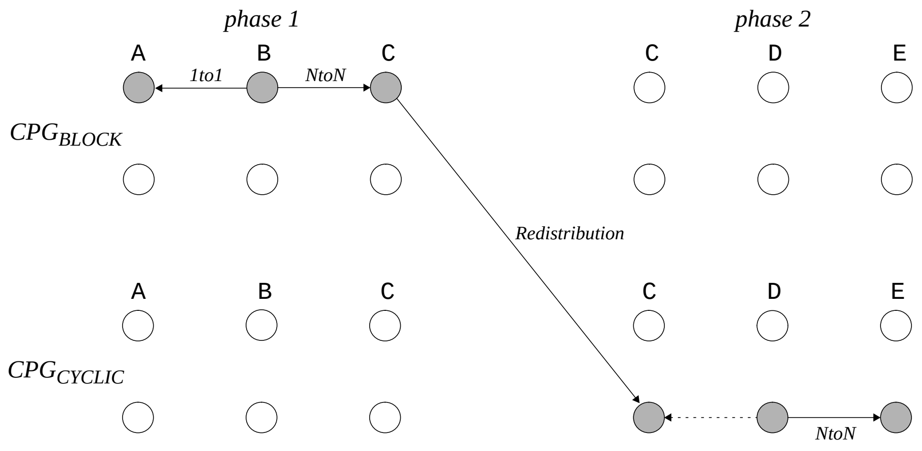


Figure 3.13 Legal mapping for the sample code with the CPG supporting both `BLOCK` and `CYCLIC` distributions.

3.3 COST MODEL IN THE CPG

The predicted computation time for the selected mapping is estimated as the sequential execution time, plus the effects of distributing data and parallelizing loops. In order to decide the best data mapping strategy, a weight should have to be assigned to each edge and hyperedge of the CPG that allows the comparison of the different kinds of information contained in the CPG. The *data movement cost* estimates the communication time in seconds spent in moving data across the processors memories, and the *computation cost* estimates the computation time in seconds saved due to the parallel execution of parallel loops. Both types of information are estimated in seconds, therefore the optimal solution has to minimize the summation of the weights of the edges minus the summation of the weights of the hyperedges. In the following Sections the cost model assumed in our framework is described.

3.3.1 Data Movement Cost

The data movement cost of a reference pattern depends on the number of bytes to interchange with remote memory accesses and on some machine specific characteristics, such as the number of processors, communication latency and bandwidth. In our current model, the memory latency is considered null.

For each reference pattern, $d \times d$ edges are added in the CPG, each with its corresponding weight. In order to estimate its cost, each edge of a reference pattern is matched with a set of predefined data movement routines [LC91]. The routines considered by our framework are listed in Table 3.1, together with the reference pattern that matches them. Note that the information in this Table is machine dependent and should be tailored to the specific target architecture.

In this Table, p is each dimension of the left-hand side (lhs) array, q is each dimension of the right-hand side (rhs) array, and i_p and j_q are the lhs and rhs subscripts of the reference pattern in the $p-th$ and $q-th$ dimension respectively. Function $const(expr)$ returns true if $expr$ contains a constant expression.

Pattern	**Primitive**
$i_p \simeq j_q$	*Local_Memory_Access*
$const(i_p - j_q)$	*One_to_One*
$const(j_q)$	*One_to_Many*
$const(i_p)$	*Many_to_One*
$i_p \neq j_q$	*Many_to_Many*

Table 3.1 Matching between reference patterns and communication primitive.

For instance, assume the first reference pattern in the first phase of the sample code of Figure 3.1:

$$A(i, j) \leftarrow B(i - 1, j)$$

The first component of the left-hand side subscript of the reference pattern is i, and the first component of the right-hand side subscript of the reference pattern is $i-1$. The type of the pattern is a $const(i - (i - 1))$ function, therefore a *One_to_One* data movement primitive is assigned to the edge $A[1] \leftarrow B[1]$. Similarly, the second component of the left-hand side subscript of the reference pattern is j. The pattern matches with $j \neq i$, therefore the data movement function associated to $A[2] \leftarrow B[1]$ is a *Many_to_Many* primitive. The same *Many_to_Many* data movement primitive is associated to the edge $A[1] \leftarrow B[2]$, and a *Local_Memory_Access* is associated to the edge $A[2] \leftarrow B[2]$.

The matching between data movement routines and reference patterns is performed to obtain an estimation of the remote accesses overhead. This cost is dependent on the block size B of the data to transfer. Table 3.2 shows the estimated block size for `BLOCK` distributions, for each data movement routine assumed in our model. Note that according to the owner-computes rule, the data to be moved is the data that has to be read, i.e. the data on the right-hand side.

In this Table, q is the dimension of the right-hand side array under consideration. N_q is the array size at dimension q, and P_q is the number of processors assigned to that dimension if finally the dimension is distributed. B_{other} is the product of the block size of the other dimensions of the right-hand side array that are traversed by a loop control variable, this is, the number of elements moved in another dimension different to q.

Primitive	**Block Size**
Local_Memory_Access	0
One_to_One	$1 \times B_{other}$
One_to_Many	$1 \times B_{other}$
Many_to_One	$(N_q/P_q) \times B_{other}$
Many_to_Many	$(N_q/P_q) \times B_{other}$

Table 3.2 Estimated block size B for each data movement primitive for `BLOCK` distributions.

For instance, consider again the first reference pattern of first phase of the sample code of Figure 3.1. Assume that the size of matrix B is $N_1 \times N_2$. The block size computed for the *One_to_One* data movement primitive assuming a `BLOCK` distribution is $1 \times B_{other}$. The block size of the other dimensions traversed by a loop control variable (j in this case) is N_2, therefore the block size computation for this edge is:

$$B \;=\; 1 \times N_2 \;=\; N_2 \; elements$$

If the distribution is `CYCLIC` then the block size computation is the same for all data movement primitives but the *One_to_One*, in which case the block size is equivalent to a *Many_to_Many* data movement primitive.

The final data movement cost is computed, expressed in seconds, as follows:

$$pattern\_cost \;=\; \frac{block\_size\ (elem) \;\times\; element\_size\ (byte/elem)}{bandwidth\ (byte/sec)} \quad (secs)$$

3.3.2 Computation Cost

The computation cost of a parallelizable loop depends on its sequential execution time and on some machine specific characteristics, such as the number of processors, and the parallel thread latency. In our current model, the parallel thread creation time is assumed to be null.

The weight assigned to each hyperedge reflects the computation time saved when the corresponding parallelizable loop is effectively parallelized. When the parallelizable loop is rectangular, and for both `BLOCK` and `CYCLIC` distributions, the saved computation time is illustrated with the shaded fragment of the matrix in Figure 3.14.a and 3.14.b, and it is computed, expressed in seconds, as:

$$paral\_loop\_saving \;=\; ((P-1)/P) \;\times\; sequential\_loop\_time$$

where P is the number of processors assigned to that dimension, in this case, the total number of available processors.

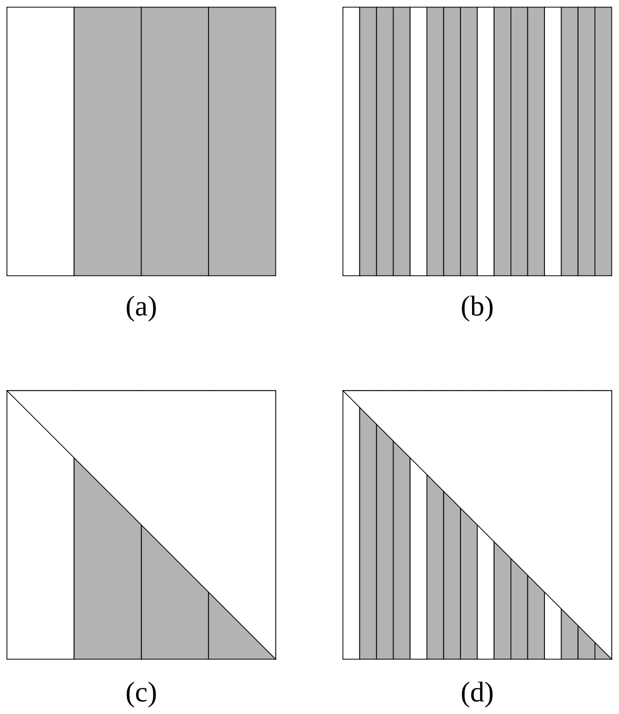


Figure 3.14 Amount of computation saved for different loop structures and distribution fashion.

Alternatively, if the parallelizable loop is triangular, there is a performance difference between `BLOCK` and `CYCLIC` distributions. If the distribution is assumed to be `BLOCK`, then the saved computation time is illustrated with the shaded fragment of the matrix in Figure 3.14.c, and it is computed as:

$$paral\_loop\_saving \;=\; ((P-1)/P) \;\times\; ((P-1)/P) \;\times\; sequential\_loop\_time$$

If the distribution is assumed to be CYCLIC, then the load is balanced and therefore the saved computation time is the same than in rectangular loops. This is illustrated with the shaded fragment of the matrix in Figure 3.14.d.

3.4 MODELING THE PROBLEM

Once the CPG has been built, it contains all required information regarding data movement and parallelism of the program. All weights in the CPG are expressed in seconds assuming P processors. The optimal mapping for the problem is a set of nodes, including one node for each column, such that the summation of weights of the corresponding edges minus the summation of weights of the corresponding hyperedges is minimal.

Linear programming (LP) provides a set of techniques that study those optimization problems in which both the objective function and constraints are linear functions. Optimization means to maximize or minimize a function with usually many variables, subject to a set of inequality and equality constraints [NW88]. A linear pure integer programming problem is a LP in which variables are subject to integrality restrictions. In addition, in many models, the integer variables are used to represent binary choices, and therefore are constrained to equal 0 or 1. In this case the model is named linear 0-1 integer programming problem.

In our framework, we translate the whole data mapping problem into a single minimal path problem with a set of additional constraints that guarantees the correctness of the solution. Our problem is not purely a minimal cost path problem, as long as multiple additional restrictions are added to the problem. Therefore this is modeled as a linear 0-1 integer programming problem, in which a 0-1 integer variable is associated to each node, edge and hyperedge. The final value for each binary variable indicates if the corresponding node, edge or hyperedge belongs to the optimal solution. The objective function that has to be minimized is the estimated execution time of the parallelized version of the original sequential program.

We have implemented two different models. The *node based model* is more intuitive but its computation using general purpose techniques is not very effective. We include its description for simplicity, although the current implementation is based on the *edge based model*, which computes much faster than the first model.

Some considerations about the CPG have to be stated before going into details with the linear 0-1 integer programming model.

- Edges in the CPG can be considered undirected. The direction of the edges in useful to understand the data flow of the reference patterns. Once their cost has been translated into units of time, its direction is not relevant any more.
- All pairs of edges connecting the same two nodes can be replaced by a single edge with weight equal to the addition of the weights of the original ones.
- There is a path between any pair of columns in the CPG. If a set of columns is not connected, then this set can be analyzed independently and assigned a different template.

In both models, a valid solution has to be specified by means of constraints and 0-1 integer variables. The objective function to minimize is *time*, the execution time of the parallel version of the original program, which is estimated as follows:

$$time \ = \ Sequential\_Time \ - \ Paral\_Saving \ + \ Comm\_Cost$$

that is, the sequential execution time, minus the time saved due to the parallel execution of loops that have been selected for parallelization, plus the overhead due to remote data accesses. Note that this is only one estimation. Actually there are some other factors that may affect this time, such as overhead for parallel thread creation, communication latency, and some others. However we assume that this costs are not significant in the overall problem.

3.4.1 Node Based Model

In this model, nodes are used to specify the correctness of the solution.

> *A valid solution is a solution that contains one and only one node for each column. The optimal solution is a valid solution which minimizes the cost of the edges that remain inside the solution minus the cost of the hyperedges that remain inside the solution.*

As it is usual in this kind of problems, one 0-1 integer variable is introduced for each node, edge and hyperedge in the CPG. Let $ncols$ be the total number of columns in the CPG, and let X_P denote the set of variables associated to nodes in column P, for each $P \in \{1..ncols\}$. Each set X_P contains d elements, named $X_P[i]$ for each $i \in \{1..d\}$. Its value is one if the node is selected in the solution, and zero otherwise. Let Y_{PQ} denote the set of variables associated to edges connecting nodes in column P to nodes in column Q. Each set Y_{PQ} contains $d \times d$ elements. Let $Y_{PQ}[i, j]$ be the variable associated to the edge connecting node i in column P to node j in column Q. Its value is one if the corresponding edge belongs to the path, and zero otherwise. Note that, as the graph is undirected, $Y_{PQ}[i, j]$ is the equivalent to $Y_{QP}[j, i]$. Finally, if an index is assigned to each hyperedge, Z_k will denote the 0-1 integer variable associated to the $k{-}th$ hyperedge. Similarly, its value will be one if all the nodes it links belong to the path, and zero otherwise.

To ensure the correctness of the solution, three sets of constraints have to be specified:

- $NodeConstraints$. This set of constraints guarantee the correctness of the solution in terms of nodes that have to be selected.
- $EdgeConstraints$. This set of constraints ensure that, if the nodes connected by an edge are selected, the edge has to be selected.
- $HyperConstraints$. This set of constraints ensure that, if the nodes connected by a hyperedge are selected, the hyperedge has to be selected.

The set of node constraints ensures the validity of the solution, that is, one node has to be selected in each column. In terms of 0-1 variables and their values, it can be stated that for each column P, the summation of all nodes i in the corresponding set of nodes

X_P must be equal to 1.

$$\sum_{i=1}^{d} X_P[i] \;=\; 1; \qquad \forall P \;\in\; \{1..ncols\}$$

The set of edge constraints ensures that edges connecting selected nodes are taken into account in the solution. In terms of 0-1 variables and their values, it can be stated, for each edge $Y_{PQ}[i,j]$ connecting node $X_P[i]$ to node $X_Q[j]$, that:

$$\begin{aligned} X_P[i] \;&\geq\; Y_{PQ}[i,j] \\ X_Q[j] \;&\geq\; Y_{PQ}[i,j] \\ X_P[i] \;+\; X_Q[j] \;&\leq\; Y_{PQ}[i,j] \;+\; 1 \end{aligned}$$

If node $X_P[i]$ or node $X_Q[j]$ equals zero, the first two constraints force the edge $Y_{PQ}[i,j]$ to be zero. The only case in which $Y_{PQ}[i,j]$ could not be zero is when both nodes equal one. In this case, the third constraint forces the edge to be equal to one. Note that the first two constraints are not obligatory, because the preferred edge value is zero as long as the objective function has to be minimized.

The set of hyperedge constraints ensure that hyperedges connecting selected nodes are considered in the solution. Assume that hyperedge Z_k connects h nodes $X_{P^1}[i^1]\cdots X_{P^h}[i^h]$ in the CPG. The hyperedge belongs to the selection if all nodes linked by it have been selected. Similarly to the set of edge constraints, in terms of 0-1 variables and their values, it can be stated, for each hyperedge Z_k, that:

$$\begin{aligned} &X_{P^1}[i^1] \;\geq\; Z_k \\ &\cdots \\ &X_{P^h}[i^h] \;\geq\; Z_k \\ &X_{P^1}[i^1] \;+\; \cdots \;+\; X_{P^h}[i^h] \;\leq\; Z_k \;+\; (h-1) \end{aligned}$$

If any node $X_{P^1}[i^1]\cdots X_{P^h}[i^h]$ equals zero, the first h constraints force the hyperedge Z_k to be zero. Alternatively, when all nodes are one, the last constraint forces the hyperedge to be one. Again note that the last constraint is not obligatory, because the preferred hyperedge value in a minimization problem is one.

Finally, the objective function has to be specified. Only edges and hyperedges have a cost associated to them. Let m be the total number of edges in the CPG. The summation of the cost of all selected edges can be expressed as the scalar product in the space $\Re^m$ of Y by C_e:

$$cost\_of\_edges \; = \; Y \; . \; C_e$$

where Y represents the vector of all 0-1 integer variables associated to edges, and C_e represents their respective weights. Similarly, let n be the nudger of hyperedges in the CPG. The summation of the cost of all selected hyperedges can be expressed as the scalar product in the space $\Re^n$ of Z by C_h:

$$cost\_of\_hyper \; = \; Z \; . \; C_h$$

where Z represents the vector of all 0-1 integer variables associated to hyperedges, and C_h represents their respective weights. The objective function has to be minimized, and can be expressed as:

$$min\_cost \; = \; cost\_of\_edges \; - \; cost\_of\_hyper$$

A general purpose integer programming solver finds the optimal solution subject to the specified constraints.

3.4.2 Edge Based Model

In this model edges are used to specify the correctness of the solution. As 0-1 integer variables associated to nodes are unweighted, they are eliminated from the model.

> *In this case a valid solution is a path that visits one and only one node for each column. All transitive edges included in the path are also included. The optimal solution is a valid solution that minimizes the cost of the edges that remain inside the solution minus the cost of the hyperedges that remains inside the solution.*

To ensure the correctness of the solution, four sets of constraints have to be defined:

- $C1$ - The solution is a path.

- $C2$ - The path visits one node in each column.
- $C3$ - All edges connecting selected nodes have to be included in the solution.
- $C4$ - Hyperedges whose nodes are all selected have to be included in the solution.

Constraint $C1$ ensures that the solution is connected. That is, for each column Q connected to more than one column P and R, if one edge leading to a node in Q is selected in the set Y_{PQ}, one edge leaving this same node must be selected in the set Y_{QR}.

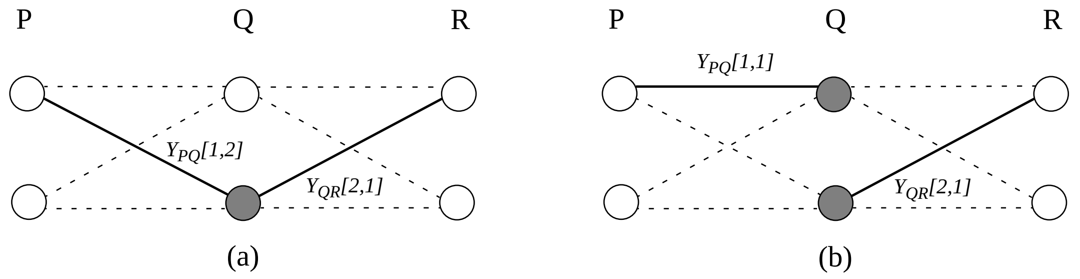


Figure 3.15 Constraint $C1$. (a) right case and (b) wrong case.

In terms of the variables and their values, it can be stated that at each node i of each column Q connected to more than one column P and R, the sum of the values of the edges that connect this node to column P, must be equal to the sum of the values of the edges that connect this node to column R.

$$\sum_{j=1}^{d} Y_{PQ}[j,i] = \sum_{j=1}^{d} Y_{QR}[i,j]; \qquad \forall i \in \{1..d\}$$

This must be accomplished for each pair of sets $Y_{PQ} - Y_{QR}$ with a column in common. In the CPG shown in Figure 3.15.a we can see that edges $Y_{PQ}[1,2]$ and $Y_{QR}[2,1]$ are selected, so:

$$Y_{PQ}[1,1] + Y_{PQ}[2,1] = Y_{QR}[1,1] + Y_{QR}[1,2] = 0$$
$$Y_{PQ}[1,2] + Y_{PQ}[2,2] = Y_{QR}[2,1] + Y_{QR}[2,2] = 1$$

which obeys this constraint. Figure 3.15.b shows a case in which the first node of Q is selected within the set Y_{PQ}, and the second node is selected within the set Y_{QR}, which is a wrong case.

Constraint $C2$ ensures that one node per column is selected in each column, and constraint $C3$ ensures that all edges that remain inside the selected path are selected as well. Both constraints can be specified together. These can be accomplished forcing that, for each non empty set of edges Y_{PQ}, exactly one edge must be selected.

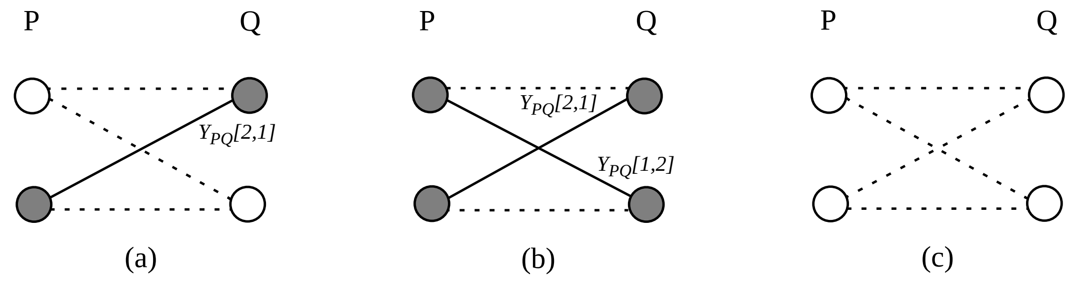


Figure 3.16 Constraints $C2$ and $C3$. (a) right case. (b) and (c) wrong cases.

This can be stated, in terms of variables and their values, that the summation of each non empty set of edges Y_{PQ} must equal one.

$$\sum_{i=1}^{d} \sum_{j=1}^{d} Y_{PQ}[i,j] \;=\; 1$$

In the CPG of Figure 3.16.a only the edge $Y_{PQ}[2,1]$ is selected, so the set of edges Y_{PQ} accomplishes that:

$$Y_{PQ}[1,1] \;+\; Y_{PQ}[1,2] \;+\; Y_{PQ}[2,1] \;+\; Y_{PQ}[2,2] \;=\; 1$$

The example in Figure 3.16.b shows a case where constraint $C2$ is not respected, and the one in Figure 3.16.c shows a case where constraint $C3$ is not accomplished.

As a consequence of these constraints, the path found in the solution can be not simple. This means that it could contain cycles, or that the same node could be more than once in the path. Note that this is not contradictory with the constraint $C2$, since exactly one different node in each column belongs to the path.

Finally, constraint $C4$ ensures the correct behavior of the hyperedges. The hyperedge belongs to the selection if all nodes linked by it have been selected. According to this model a node i in column P is selected if one of the edges $Y_{PQ}[i,j]$ that connect it to any other column Q has been selected, that is:

$$\sum_{j=1}^{d} Y_{PQ}[i,j] \;=\; 1$$

Assume that hyperedge Z_k connects h nodes in the CPG, say nodes $X_{P^1}[i^1]\cdots X_{P^h}[i^h]$, connected by sets of edges $Y_{P^1Q^1}\cdots Y_{P^hQ^h}$ respectively. It can be stated, in terms of variables and their values, that:

$$\begin{aligned}
&\sum_{j=1}^{d} Y_{P^1Q^1}[i^1,j] \;\geq\; Z_k\\
&\cdots\\
&\sum_{j=1}^{d} Y_{P^hQ^h}[i^h,j] \;\geq\; Z_k
\end{aligned}$$

If any summation equals zero, the constraint forces the hyperedge Z_k to be zero. Otherwise, and by default, the value of the hyperedge is one.

The objective function specified for this model is the same than in the previous one. The general purpose integer programming solver finds the optimal solution subject to the specified constraints.

3.5 AN EXAMPLE: THE ADI INTEGRATION KERNEL

Assume the Alternate Direction Implicit (ADI) integration kernel, whose source code can be found in Appendix A. According to the definition of phase, our tool identifies 9 phases in this program. Each phase corresponds to one of the nested loops (labeled from 1 to 9) in Figure A. The control flow analysis detects that phases 4 to 9 are within an iterative loop, and according to the profiled information, the number of iterations of this loop is 10. The resulting phase control flow graph for the ADI program is illustrated in Figure 3.17, together with the data flow analysis. Note that remapping between phase 9 and phase 4 is performed 9 times, and that the data flow analysis detects that array A is not used in phases 5 and 8.

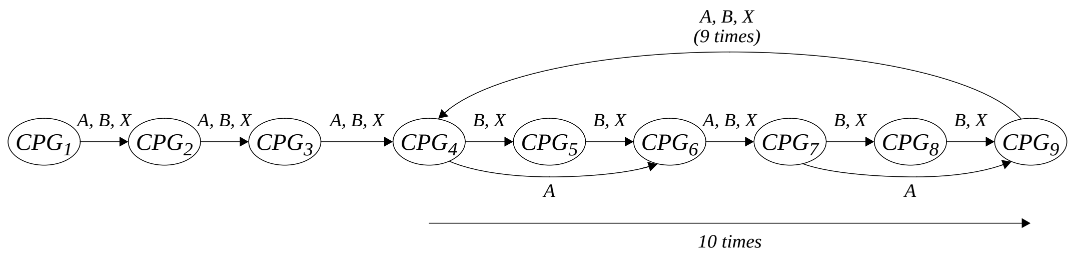


Figure 3.17 Phase control flow graph and data flow analysis for the ADI program.

For each phase i the tool generates its own CPG_i. For instance, when analyzing the 6th phase, the following 4 reference patterns are detected:

$$\begin{aligned} x(i,j) &\leftarrow x(i,j) \\ x(i,j) &\leftarrow a(i,j+1) \\ x(i,j) &\leftarrow x(i,j+1) \\ x(i,j) &\leftarrow b(i,j) \end{aligned}$$

and the following 8 reference patterns are detected in the 7th phase:

$$\begin{aligned} x(i,j) &\leftarrow x(i,j) & b(i,j) &\leftarrow b(i,j) \\ x(i,j) &\leftarrow x(i-1,j) & b(i,j) &\leftarrow a(i,j) \\ x(i,j) &\leftarrow a(i,j) & b(i,j) &\leftarrow a(i,j) \\ x(i,j) &\leftarrow b(i-1,j) & b(i,j) &\leftarrow b(i-1,j) \end{aligned}$$

The data movement information for these two phases is shown in Figure 3.18. As usual, dotted edges are $Local\_Memory\_Access$. The edges connecting different dimensions of different arrays within the same phase have assigned the weight of a $Many\_to\_Many$ data movement primitive, and the edges connecting different dimensions of the same array in different phases have assigned the weight of a *Realignemnt* data movement primitive. All other edges have assigned the weight of a One_to_One data movement primitive.

The dependence analysis detects 10 loops candidate to be parallelized. These loops are the four initialization loops, three i loops in phases $4 \cdots 6$, and three j loops in phases $7 \cdots 9$. These 10 loops have their own hyperedge in the corresponding phases, and in particular, the hyperedge of the i loop in phase 6 and the hyperedge of the j loop in phase 7 can be seen in the CPG of Figure 3.18.

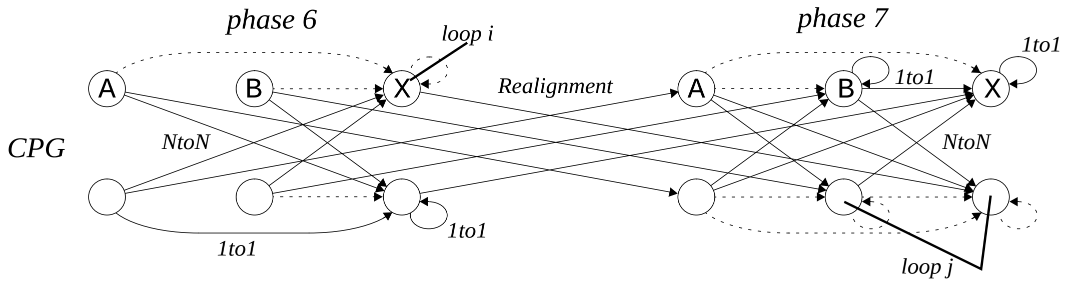


Figure 3.18 CPG for phases 6 and 7.

In order to compute the weights in the CPG, assume a target system with 32 processors and a data movement bandwidth of 1Mbyte per second. According to our current cost model, the cost of a *Many_to_Many* data movement primitive and the cost of a *Realignment* data movement primitive is the same. This depends on the array block size. As all arrays have the same size, this cost will be the same in all edges in the CPG labeled with these data movement primitives, and it is computed as:

$$\frac{31}{32} \times \frac{256 \cdot 256}{32} \times 8 \times \frac{1}{1000000} \;=\; 0.01587 \; secs$$

where 8 is the number of bytes per each double precision array element. Similarly, the cost of a *One_to_One* data movement primitive for all edges labeled with this data movement primitive is computed as:

$$256 \times 8 \times \frac{1}{1000000} \;=\; 0.00205 \; secs$$

The profiling information for this code provides us with the sequential execution time for each loop in the program, and the number of iterations of the *iter* loop. In order to weight the hyperedges, the sequential execution time for each candidate parallel loop is used. These times for the i loop in phase 6 and for the j loop in phase 7 are 0.53472 and 0.89654 seconds respectively. The weights assigned to each hyperedge is:

$$loop_i \; in \; phase \; 6 \;\rightarrow\; \frac{31}{32} \times 0.53472 \;=\; 0.51801 \; secs$$
$$loop_j \; in \; phase \; 7 \;\rightarrow\; \frac{31}{32} \times 0.89654 \;=\; 0.86852 \; secs$$

The same analysis is performed for all other phases in the ADI code. Once the CPG has been built, the minimal path problem with 0-1 integer variables is modeled. Both the node and the edge based model have been implemented. Table 3.3 shows some characteristics of these models.

	Node based model	**Edge based model**
Number of nodes	56	-
Number of edges	160	160
Number of hyper	10	10
Number of 0-1 variables	226	170
Number of constraints	535	164
Computation time	**1.7 secs.**	**0.5 secs.**

Table 3.3 CPG complexity for the ADI code in node and edge based models.

The difference in complexity of both models is significant in number of variables and constraints, as long as in computation time [1]. The solution found in both cases is dynamic, distributing the first dimension of all arrays in phases $1 \cdots 6$ and the second dimension of all arrays in phases $7 \cdots 9$. According to this distribution, one loop at each phase can be parallelized. The mapping strategy for phases 6 and 7 can be seen in Figure 3.19. Note that realignment has to be performed 10 times from phase 6 to 7, plus 9 times from phase 9 to 4.

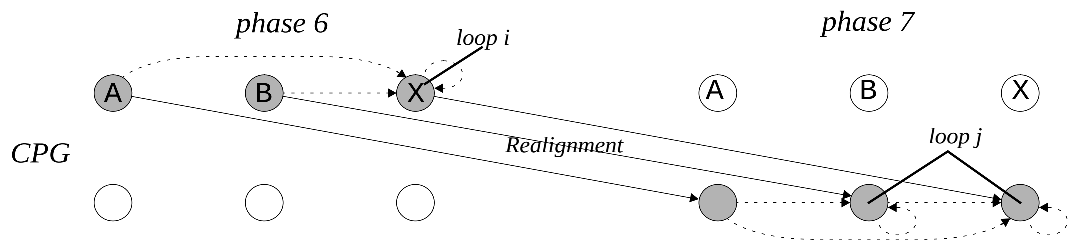


Figure 3.19 Optimal data mapping for phases 6 and 7 of the ADI code.

In Chapter 6 the correctness and accuracy of this data mapping strategy will be illustrated.

[1]Times have been obtained executing on a Sun SuperSparc 20

There are some systems in which data remapping is not allowed, or in which it is allowed but only between procedure boundaries. Note that a static data mapping can be modeled as a particular case in our model, reducing the complexity of the problem. Our first CPG implementation was static, in which the whole program was analyzed as a single phase. In Table 3.4 there is the complexity of the same ADI code, according to the static implementation. Although the number of reference patterns in the ADI code is the same, most of them are repeated, therefore they are collapsed within the same edge. The number of 0-1 integer variables is 12, in front of 226 in the nodes based general model. The total computation time required to find the optimal solution is only 0.1 seconds.

	Static model
Number of edges	12
Number of hyper	10
Number of 0-1 variables	22
Number of constraints	30
Computation time	**0.1 secs**

Table 3.4 Complexity for the static CPG version with the ADI code.

3.6 SUMMARY

In this Chapter we have described the Communication-Parallelism Graph, the main structure of our approach to automatically derive data mapping and parallelization strategies. The novelty of the approach resides in that data movement as well as parallelism information are contained within this single data structure. This allows us to solve several dependent problems, such as alignment, distribution, and remapping, in a single step. In addition, we model our problem as an optimization problem, based on the minimal path problem, and use linear 0-1 integer programming technology, which guarantees that the solution found is optimal according to our cost model.

The data mapping features described in this Chapter are one-dimensional `BLOCK` and `CYCLIC` distributions, allowing dynamic data remapping between phases if required. In addition, our model takes into account the effects of control flow statements between phases, in order to perform a more accurate estimation of the cost of the data mapping derived.

We have described with detail the data movement and parallelization cost models assumed in our approach. And at the end of the Chapter, the formal specification of the linear 0-1 integer programming model has been provided. We have presented two different approaches: the node based model and the edge based model. The first one is more intuitive, but the computation time required to solve the model is quite high. This model is useful as an introduction to the second one, which is faster. The model used along the rest of this Thesis is the edge based model.

4 TWO-DIMENSIONAL DISTRIBUTION WITH CONSTANT TOPOLOGY

In this Chapter, we describe how to extend the CPG in order to support two-dimensional distributions, assuming that the processors topology is two-dimensional, constant, and known at compilation time. Next we explain the modifications to our two-dimensional cost model with respect to data movement and computation, and the extension in the formulation of the minimal path problem. And finally, an example using the ADI code is provided.

We believe that for most scientific programs, restricting the number of distributed dimensions of a single array to two, does not lead to any loss of effective parallelism. In a study of array reference patterns in Fortran programs from the Perfect Club and SPEC benchmark sets performed by our group in [ALG$^+$95], we reported that 3.3% of the arrays have 3 dimensions, and only 0.2% have more than 3. Even when higher-dimensional arrays show parallelism in each dimension, restricting the number of distributed dimensions does not necessarily limit the amount of parallelism that can be exploited.

4.1 EXTENDING THE COMMUNICATION-PARALLELISM GRAPH

A valid data mapping strategy in the two-dimensional distribution with constant topology, distributes two dimensions of the template, either with `BLOCK` or `CYCLIC` fashion, over a two-dimensional processor grid. Let p be the number of processors in the target architecture. Usually p is a number power of 2, i.e. $p = 2^q$. The number of processors assigned to each distributed dimension is p_1 and p_2 respectively, where $p_1 \times p_2 = p$. This grid topology is constant during the execution of the program. Note that this general case includes the trivial one, when p_1 is set to p, and p_2 is set to 1.

In order to support this model some extensions have to be defined in the CPG. In the following Sections we describe these extensions.

4.1.1 Duplication of the CPG

In the one-dimensional data distribution model, we provide a graph structure with a node for each distributable array dimension. In the two-dimensional data distribution model the idea is to have a node for each distributable array dimension in the first processors grid dimension, and another node for each distributable array dimension in the second processors grid dimension.

With this purpose, two identical CPG with different weights are built. In the first CPG copy, named CPG^1, all weights are computed assuming p_1 processors, and in the second CPG copy, named CPG^2, all weights are computed assuming p_2 processors. Each CPG copy represents one processors grid dimension. In order to distribute one array dimension across the first processors grid dimension, the corresponding node for that array has to be selected in CPG^1. Similarly, to distribute one array dimension across the second processors grid dimension, the corresponding node for that array has to be selected in CPG^2. Note that it has no sense to try to distribute the same array dimension across two different dimensions of the processors grid.

Now, a valid mapping strategy for the two-dimensional distribution with constant topology problem contains one node for each column in each CPG copy, with the additional restriction that nodes selected in CPG^1 have to be different than nodes selected in CPG^2. The data movement and parallelization effects for the selected two-dimensional data mapping is estimated as the summation of the weights of the edges plus the summation of the weights of the hyperedges that remain inside the two paths.

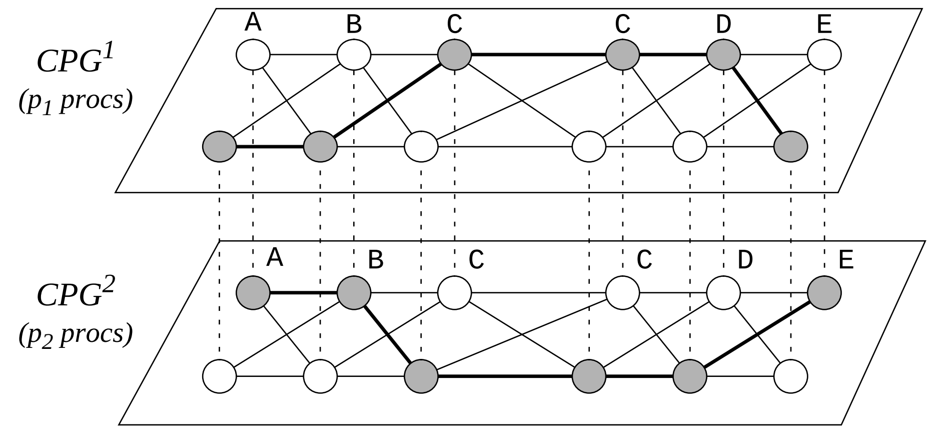


Figure 4.1 Valid solution in a 2-dimensional CPG.

In Figure 4.1 there is an example of a valid mapping, in which the second dimension of arrays A and B and the first dimension of array C in the first phase, and the first dimension of arrays C and D and the second dimension of array E in the second phase are aligned and distributed along the first dimension of the $p_1 \times p_2$ processors grid. Similarly, the first dimension of arrays A and B and the second dimension of array C in the first phase, and the second dimension of arrays C and D and the first dimension of array E in the second phase are aligned and distributed along the second dimension of the processors grid. This data mapping strategy can be specified in terms of HPF directives as:

```
!HPF$  PROCESSORS PROC_GRID(P1, P2)
!HPF$  TEMPLATE T(N, N)
!HPF$  ALIGN A(i, j) WITH T(j, i)
!HPF$  ALIGN B(i, j) WITH T(j, i)
!HPF$  ALIGN C(i, j) WITH T(i, j)
!HPF$  ALIGN D(i, j) WITH T(i, j)
!HPF$  ALIGN E(i, j) WITH T(j, i)
!HPF$  DISTRIBUTE T(BLOCK, BLOCK) ONTO PROC_GRID
```

Note that in this case, even that the processors topology is assumed to be constant along the program execution, the distribution derived might be non-static. There can be changes in the array alignment between phases, i.e. realignments. However, the processors topology will still be a $p_1 \times p_2$ grid.

4.1.2 Corrector Edges in the CPG

According to this model, the estimated data movement costs for a two-dimensional data mapping is the addition of the computed data movement costs for CPG^1 plus the computed data movement costs for CPG^2. However, the data movement in both dimensions is inter-dependent each other, therefore this estimation is not accurate. Similarly, the actual execution time saved due to a two-dimensional loop parallelization is not the addition of the saving execution time of two one-dimensional loops.

To correct this the over estimation, edges connecting information in CPG^1 to the related information in CPG^2 are inserted in the CPG. These edges are named *corrector edges* as long as their utility is to correct the over estimation induced by our model. The weight assigned to each corrector edge is a symbolic expression representing the cost that has to be corrected. Therefore, data movement corrector edges correct the over estimation amount of data movement, and parallelism corrector edges correct the over estimation in parallelism.

One data movement corrector edge should have to be inserted connecting each edge in a pattern in CPG^1 to each edge of the same pattern in CPG^2. However, the number of data movement corrector edges in this case would grow up to intractable proportions. To avoid this, we modify the data movement cost model explained in the previous Chapter, in order to perform a proper data movement cost estimation. The model assumed in the two-dimensional data mapping is described in Section 4.2.1.

With respect to the parallelism, we have to insert one parallelism corrector edge connecting each couple of nested hyperedges between both CPG copies. A parallelism corrector edge connects the outer loop of a CPG copy to the inner loop of the other CPG copy. Therefore, for each two candidate parallel loops in a nest, there is a corrector edge from CPG^1 to CPG^2, and vice versa. The first parallelism corrector edge corrects the over estimation of parallelizing the outer loop with p_1 processors, and the inner loop with p_2 processors. Similarly, the second parallelism corrector edge corrects the over estimation of parallelizing the outer loop with p_2 processors, and the inner loop with p_1 processors. As parallelism corrector edges are the only corrector edges used in our model, in following references we will call them just corrector edges.

As a general rule, a loop at level i can be the outer loop of the $i-1$ loops inside it. Assuming that in a loop nest there are n candidate parallel loops, the number of corrector edges added connecting CPG^1 to CPG^2 is:

$$n-1 \; + \; n-2 \; + \; \cdots \; + \; 1 \; = \; \sum_{i=n-1}^{1} i \; = \; \frac{n \times (n-1)}{2}$$

and the same number of corrector edges connecting CPG^2 to CPG^1. Therefore the total number of corrector edges in the CPG for a loop nest with n candidate parallel loops is:

$$n \times (n-1)$$

When two hyperedges linked by a corrector edge are selected in a solution, the corrector edge has to be selected as well. In this case, the over estimation produced by considering the addition of the weights associated to both hyperedges will be corrected by considering the weight of the corrector edge. The weight assigned to a corrector edge is described in Section 4.2.2.

4.2 TWO-DIMENSIONAL COST MODEL

The data movement cost model in the two-dimensional data distribution case has to be modified in order to perform a proper. In addition, there are some corrector edges linking hyperedges from CPG^1 to CPG^2, and vice versa. The modification of these costs, and the weighting of the corrector edges is explained in the following Sections.

4.2.1 Data Movement Cost

In our framework, only simple data movement routines are considered, i.e. routines that perform data movement in a single dimension of the template. If the reference pattern requires data movement in more than one dimension, then the reference pattern is decomposed into sub-patterns and each sub-pattern is matched with a single data movement primitive, each one performing data movement in a single dimension of the arrays. Thus, the data movement primitive of each dimension is reflected in each copy of the CPG.

However, the block size estimation has to be performed differently than in the one-dimensional distribution. In this case, we assume that there is always another dimension distributed. According to this assumption, block sizes for CPG^1 copy are divided by p_2, and vice versa. Note that this particular cost model can be extended to the case in which the topology is a $p \times 1$ processors grid. When computing data movement costs for CPG^1 (with p processors), block sizes will have to be divided by the number of processors in the other processors grid dimension, i.e. by 1. Therefore the block sizes computed will be the same than in the one-dimensional case.

For instance, consider the following reference pattern:

$$A(i, j, k) \leftarrow B(i-1, k, j)$$

3×3 edges are added connecting all nodes of array B to array A, with a one-dimensional data movement routine assigned to each edge, as shown in Figure 4.2. According to the one-dimensional pattern matching model, the edge connecting $A[1] \leftarrow B[1]$ has a *One_to_One* data movement primitive associated; all other edges have a *Many_to_Many* data movement primitive associated, except dotted ones, which are *Local_Memory_Access*.

Assuming that $p = 8$ with $p_1 = 4$ and $p_2 = 2$, costs in CPG^1 are computed assuming 4 processors, and costs in CPG^2 are computed assuming 2 processors. When computing data movement costs, the total block size for reference patterns in CPG^1 have to be divided by 2, and the total block size for reference patterns in CPG^2 have to be divided by 4. For instance, assume the data mapping specified by the following set of HPF data mapping directives:

```
!HPF$  PROCESSORS PROC_GRID(4, 2)
!HPF$  TEMPLATE T(N1, N2, N3)
```

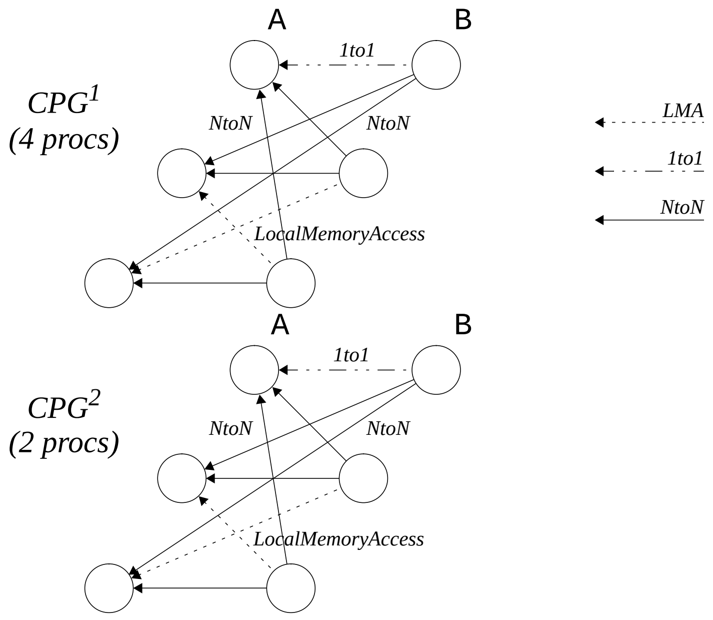


Figure 4.2 Reference pattern for a 2-dimensional CPG.

```
!HPF$  ALIGN A(i, j, k) WITH T(i, j, k)
!HPF$  ALIGN B(i, j, k) WITH T(i, k, j)
!HPF$  DISTRIBUTE T(BLOCK, BLOCK, *) ONTO PROC_GRID
```

where N_1, N_2, and N_3 are the matrix sizes for their three dimensions. In this mapping strategy the first dimension of arrays A and B are aligned and distributed in CPG^1, and the second dimension of array A and third dimension of array B are aligned and distributed in CPG^2. According to this mapping strategy, an edge labeled with a *One_to_One* data movement primitive is selected in CPG^1, and an edge labeled with a *Local_Memory_Access* data movement primitive is selected in CPG^2. The total data movement involved in this pattern is the *One_to_One* data movement primitive in CPG^1, corresponding to the first dimension of the processors grid. In Figure 4.3.a the corresponding two-dimensional data movement is illustrated. The block size estimated for this pattern in CPG^1 is:

$$\frac{1 \times B_{other}}{2} = \frac{1 \times N_2 \times N_3}{2}$$

being 1 the offset in the shift pattern, and 2 the number of processors assigned to the other dimension of matrix B. Note that this block size is the same size than in the one-dimensional data mapping, but divided by 2. Moreover, it is not necessary to know whether it is the second or the third dimension of matrix B which is distributed in CPG_2, as long as the division by 2 affects the total block size.

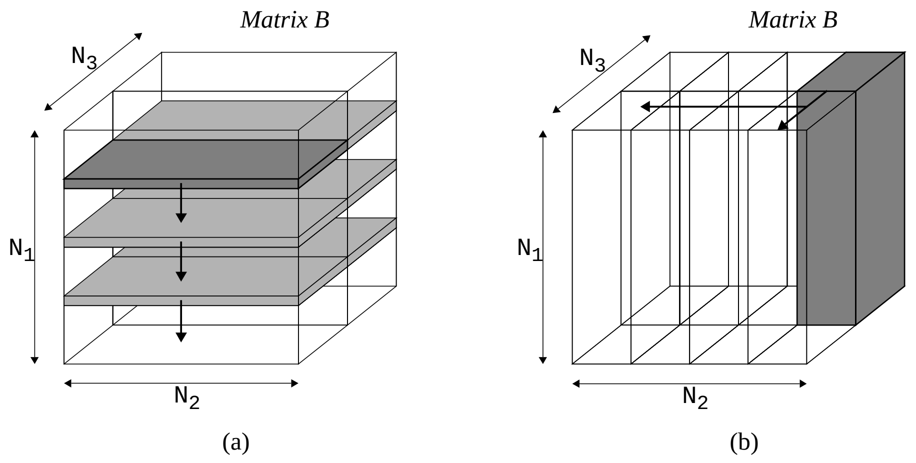


Figure 4.3 Examples of two-dimensional data movement.

Alternatively, assume the data mapping specified by the following set of HPF data mapping directives:

```
!HPF$  PROCESSORS PROC_GRID(4, 2)
!HPF$  TEMPLATE T(N1, N2, N3)
!HPF$  ALIGN A(i, j, k) WITH T(i, j, k)
!HPF$  ALIGN B(i, j, k) WITH T(i, j, k)
!HPF$  DISTRIBUTE T(*, BLOCK, BLOCK) ONTO PROC_GRID
```

According to this mapping strategy, the second dimension of arrays A and B are aligned and distributed in CPG^1, and the third dimension of arrays A and B are aligned and distributed in CPG^2. The data movement involved in this pattern is a $Many\_to\_Many$ primitive in both CPG copies. The expected data movement performed is illustrated in Figure 4.3.b. The block size of a $Many\_to\_Many$ data movement primitive in CPG^1 is:

$$\frac{\frac{N_2}{4} \times B_{other}}{2} = \frac{N_1 \times N_2 \times N_3}{4 \times 2}$$

where 4 is the number of processors assigned to CPG^1, 2 is the number of processors assigned to another dimension of matrix B, and B_{other} is $N_1 \times N_3$. Similarly, the block size of a $Many\_to\_Many$ data movement primitive in CPG^2 is computed as:

$$\frac{\frac{N_3}{2} \times B_{other}}{4} = \frac{N_1 \times N_2 \times N_3}{2 \times 4}$$

Note that the same block of data has to be moved across both processors grid dimensions.

4.2.2 Computation Cost

The estimation of the parallel loop saving execution time for two-dimensional distributions is initially computed in the CPG as the summation of the saving execution time in both CPG copies. However, when two loops are nested each other, this estimated saving execution time is excessive, and corrector edges have to be inserted in the CPG. The amount of excessive estimated execution time for two nested loops depends on the loop shape as well as on the loops distribution fashion.

For instance, assume a squared two-dimensional iteration space as shown in Figure 4.4, and that the two-dimensional processors grid is again 4×2. In the upper-left side of the Figure, the shaded portion corresponds to the saving execution time computed in CPG^1, and in the lower-left side of the Figure, the shaded portion corresponds to the saving execution time computed in CPG^2. The over estimation loop execution time in this case is:

$$\frac{3}{4} \times \frac{1}{2} \times inner\_loop\_time$$

This over estimation expression depends only on the inner loop execution time, because it is the part of code affected by the assignment of processors in both dimensions.

As a general rule, assuming that p_{out} processors are assigned to the outer loop and p_{in} processors are assigned to the inner one:

- when both loops are balanced, i.e. are rectangular or are parallelized with a `CYCLIC` distribution, then the over estimation time is computed as:

$$\frac{p_{out} - 1}{p_{out}} \times \frac{p_{in} - 1}{p_{in}} \times inner\_loop\_time$$

  which corresponds to the time computed for the example in Figure 4.4.

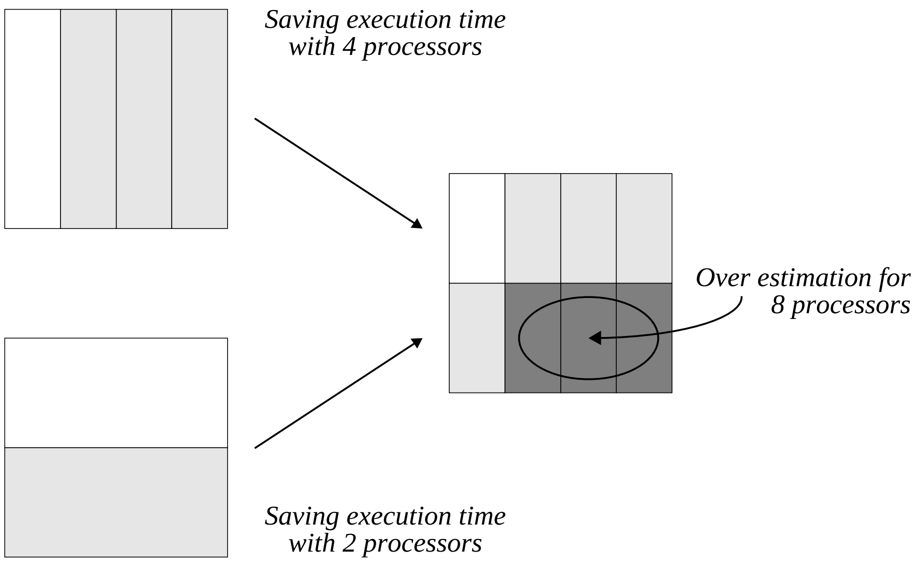

Saving execution time with 4 processors
Over estimation for 8 processors
Saving execution time with 2 processors

Figure 4.4 Over estimation of the loop execution time with 4×2 processors.

- when the outer loop is balanced, and the inner loop is triangular and parallelized with a `BLOCK` distribution, then the over estimation time is computed as:

$$\frac{p_{out}-1}{p_{out}} \times \frac{p_{in}-1}{p_{in}} \times \frac{p_{in}-1}{p_{in}} \times inner\_loop\_time$$

- alternatively, when the outer loop is triangular and parallelized with a `BLOCK` distribution, and the inner loop is balanced, then the over estimation time is computed as:

$$\frac{p_{out}-1}{p_{out}} \times \frac{p_{out}-1}{p_{out}} \times \frac{p_{in}-1}{p_{in}} \times inner\_loop\_time$$

- and finally, when both loops are triangular and parallelized with a `BLOCK` distribution, then the over estimation time is computed as:

$$\frac{(p_{out}-1)\cdot(p_{in}-1)-1}{p_{out}\cdot p_{in}} \times \frac{(p_{out}-1)\cdot(p_{in}-1)-1}{p_{out}\cdot p_{in}} \times inner\_loop\_time$$

Note that this over estimation is always positive, as long as p_{out} and p_{in} are greater than one. In the trivial one-dimensional case, where p_{in} is one, the product in the over estimation expression is zero.

For instance, assume the following nest of parallelizable loops, which has been measured by profiling to spend 10 seconds in their sequential execution:

```
do i = 1, N
        do j = 1, N
                A(i, j) = ···
                B(i, j) = ···
        enddo
enddo
```

The CPG fragment in Figure 4.5 shows the parallelism related information which includes, for each CPG copy, one hyperedge associated to loop i and another one associated to loop j. In addition, there is a corrector edge connecting the i loop in CPG^1 to the j loop in CPG^2, and another corrector edge connecting the i loop in CPG^2 to the j loop in CPG^1. Assuming that p_1 is 4, and p_2 is 2, the weights assigned to each hyperedge in CPG^1 are set to 7.5 seconds. Similarly, the weights in each hyperedges in CPG^2 are set to 5 seconds. If the hyperedge associated to loop i is selected in CPG^1, and the hyperedge associated to loop j is selected in CPG^2, the estimated saved execution time is:

$$7.5 + 5 = 12.5 \ seconds$$

which is greater than the sequential execution time. However, if the corrector edge is selected as well, then its cost has to be subtracted to this estimated execution time. According to the previous formulas, the weight assigned to each corrector edge is set to

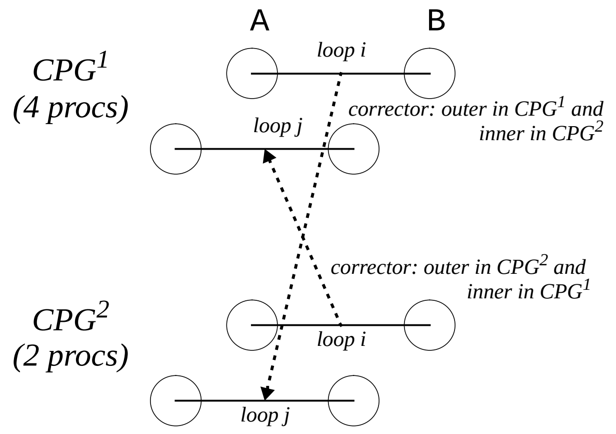

A
B
CPG1 (4 procs)
loop i
corrector: outer in CPG1 and inner in CPG2
loop j
corrector: outer in CPG2 and inner in CPG1
CPG2 (2 procs)
loop i
loop j

Figure 4.5 CPG fragment with two-dimensional parallelism information.

3.75 seconds. In this case, the estimated saved execution time will be:

$$7.5 + 5 - 3.75 = 8.75 \; seconds$$

4.3 MODELING THE PROBLEM

The definition of a valid solution for the two-dimensional distribution with constant topology problem has to be extended in order to take into account both CPG copies, and the corrector edges.

> *A valid solution is a path in each CPG copy that visits one and only one node for each column, with the additional constraint that the nodes selected in one path have to be different to the nodes selected in the other path.*

As in the one-dimensional edge based model, one 0-1 integer variable is associated to each edge, hyperedge, and corrector edge. Note that there is an edge and a hyperedge at each CPG copy, therefore a new superscript is added to each variable denoting the CPG copy it corresponds to. In this case, let $Y^b{}_{PQ}[i, j]$ be the variable associated to the edge that connects node i of column P to node j of column Q in the $b - th$ CPG copy, for each $b \in 1..2$. The same notation is used for hyperedges, where $Z^b{}_m$ represents the $m - th$ cariable associated to the hyperedge in the $b - th$ CPG copy. Corrector edges connect hyperedges of both CPG copies, therefore there is a single set of these variables. Let W_m denote the 0-1 integer variable associated to the $m - th$ corrector edge. Its value will be one if both hyperedges it links belong to the path, and zero otherwise.

4.3.1 New Constraints

To ensure the correctness of the solution, the four constraint sets defined in Chapter 3 have to be specified for each CPG copy, and modified in order to include the new superscripts defined. In addition, the following two additional sets of constraints have to be specified:

- $C5$ - All corrector edges connecting selected hyperedges have to be included in the solution.
- $C6$ - Nodes selected in each CPG copy are distinct.

Constraints $C5$ ensure the correct behavior of each corrector edge. In a similar way than in $C4$, a corrector edge belongs to the path if both hyperedges it links have been selected. In terms of variables and their values, let W_m be the variable associated to the corrector edge connecting hyperedge ${Z^1}_p$ to ${Z^2}_q$ associated to loop p in CPG^1 and loop q in CPG^2. It can be stated in terms of variables and their values that:

$$\begin{aligned} {Z^1}_p &\geq W_m \\ {Z^2}_q &\geq W_m \\ {Z^1}_p + {Z^2}_q &\leq W_m + 1 \end{aligned}$$

which respects this constraint. When any hyperedge is not selected, the first two constraints force the corrector edge to be zero. And when both hyperedges are selected, the third constraint forces it to be one.

Constraints $C6$ ensure that paths do not have nodes in common. This can be modeled imposing that the summation of edges connecting each node of both CPG copies to any other column is smaller than or equal to one. In terms of variables and their values, for each node i in column P connected to another column Q by Y_{PQ}, the summation of the values of the variables associated to the edges of both CPG copies that connect this node to column Q has to be smaller than or equal to one.

$$\sum_{b=1}^{2} \sum_{j=1}^{d} {Y^b}_{PQ}[i,j] \leq 1; \qquad \forall i \in \{1..d\}$$

In the CPG shown in Figure 4.6.a the edges ${Y^1}_{PQ}[1,2]$ and ${Y^2}_{PQ}[2,1]$ have been selected, so:

$$\begin{aligned} {Y^1}_{PQ}[1,1] + {Y^1}_{PQ}[1,2] + {Y^2}_{PQ}[1,1] + {Y^2}_{PQ}[1,2] &\leq 1 \\ {Y^1}_{PQ}[2,1] + {Y^1}_{PQ}[2,2] + {Y^2}_{PQ}[2,1] + {Y^2}_{PQ}[2,2] &\leq 1 \end{aligned}$$

which respects this constraint, as long as both summations equal one. In Figure 4.6.b the edges ${Y^1}_{PQ}[1,2]$ and ${Y^2}_{PQ}[1,1]$ have been selected, which is a wrong case.

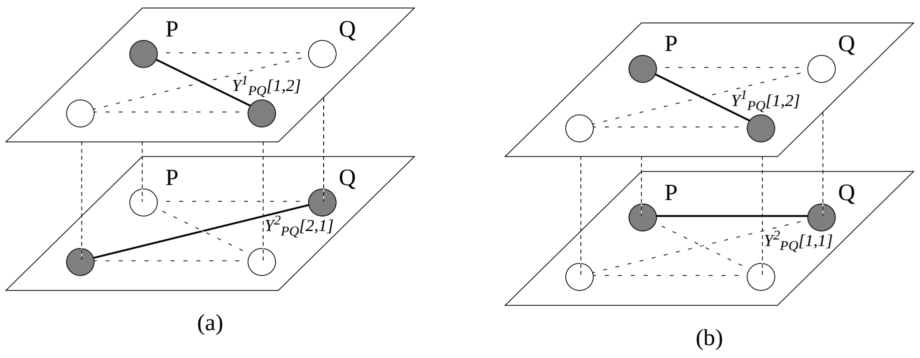


Figure 4.6 Constraint $C5$. (a) right case and (b) wrong case.

Now, the objective function has to consider the correcting costs of all corrector edges selected. Therefore the objective function expression to minimize will be specified as:

$$min\_cost = cost\_of\_edges - cost\_of\_hyper + cost\_of\_corrector$$

where $cost\_of\_corrector$ represents the scalar product of the vector of all 0-1 integer variables associated to corrector edges by their respective weights.

4.4 AN EXAMPLE: TWO-DIMENSIONAL DISTRIBUTION FOR THE ADI KERNEL

Consider again the Alternate Direction Implicit (ADI) integration kernel, whose source code can be found in Appendix A. Assume that the target platform is a system with 32 processors, arranged in a two-dimensional 8×4 processors grid. In order to compute the weights in the CPG, assume a data movement bandwidth of 1Mbytes per second.

The CPG for the two-dimensional data distribution is duplicated, however each CPG copy is the same than in the one-dimensional case. The only difference is the cost of the weights associated to the edges and hyperedges, and the addition of the corrector edges in the two-dimensional data distribution.

The cost of a $Many\_to\_Many$ data movement primitive in CPG^1 is computed considering 8 processors, but assuming that the other array dimension is distributed as well. Therefore this cost is computed as:

$$\frac{\frac{256 \cdot 256}{8} \times 8 \times \frac{1}{1000000}}{4} = 0.01638 \ secs$$

The cost of the edges labeled with a $Many\_to\_Many$ data movement primitive in CPG^2 is the same in this case, as long as dividing the matrix size by 8 and then divide by 4 is the same than dividing the matrix size by 4 and then divide by 8.

Similarly, the cost of a *One_to_One* data movement primitive for all edges labeled with this data movement primitive in CPG^1 is computed as:

$$\frac{256 \times 8 \times \frac{1}{1000000}}{4} = 0.00051 \ secs$$

and the cost of a *One_to_One* data movement primitive for all edges labeled with this data movement primitive in CPG^2 is computed as:

$$\frac{256 \times 8 \times \frac{1}{1000000}}{8} = 0.00025 \ secs$$

Note that in this case the cost in the second grid dimension, i.e. CPG^2, is half the cost in the first grid dimension, i.e. CPG^1.

The weights assigned to each hyperedge in this case are computed like in one-dimensional distributions, except that the number of processors assumed in CPG^1 is 8, and 4 in CPG^2. The only loop nest with more than one candidate parallel loop is an initialization phase whose profiled execution time can be considered null, therefore corrector edges are not required.

Once the CPG has been built, the minimal path problem with 0-1 integer variables is modeled with the edges based model. Table 4.1 shows the characteristics of this model. The computation time spent in finding the optimal solution according to this model is 1.6 seconds [1].

The optimal solution generated by the solver is static, distributing the first dimension of all the arrays across the second dimension of the processors grid, i.e. across 4 processors, and the second dimension of all the arrays across the first dimension of the processors grid,

[1]Times have been obtained executing on a Sun SuperSparc 20

Number of edges	320
Number of hyper	20
Number of corrector	2
Number of 0-1 variables	342
Number of constraints	386
Computation time	**1.6 secs.**

Table 4.1 CPG complexity for two-dimensional distributions of the ADI code.

i.e. across 8 processors. The optimal mapping strategy for phases 6 and 7 is illustrated in Figure 4.7.

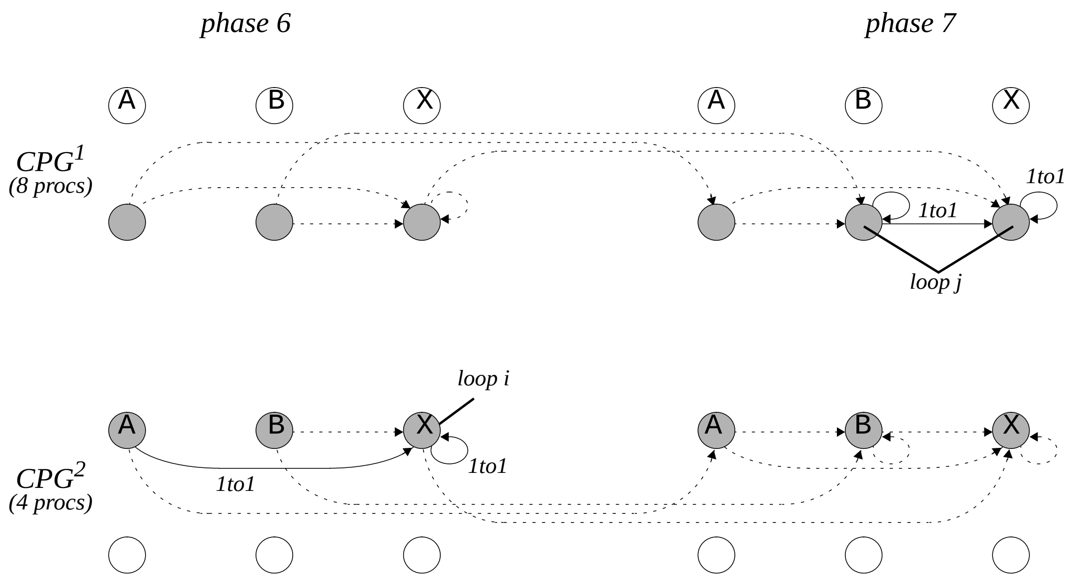


Figure 4.7 Optimal two-dimensional data mapping for phases 6 and 7 of the ADI code.

4.5 SUMMARY

In this Chapter we have extended the Communication-Parallelism Graph in order to support two-dimensional data mappings, assuming that the processors topology is constant, and known at compilation time. The basic idea under this model is that the CPG is duplicated. Assuming that the topology is a $p_1 \times p_2$ processors grid, all weights in CPG^1 are computed assuming p_1 processors, and weights in CPG^2 are computed assuming p_2 processors.

The cost model for two-dimensional data mappings has been modified with respect to the one-dimensional case. For data movement costs, we decompose the reference pattern into two one-dimensional data movement primitives. The block size for each data movement primitive is computed assuming that there is always another dimension distributed. For the parallelism cost model, we have introduced the corrector edges, that connect each couple of nested hyperedges. If two nested hyperedges are selected for the solution, the corrector edge is selected as well. The weight assigned to the corrector edge corrects the over estimation execution cost of selecting two nested hyperedges.

Finally, we have extended the minimal path problem formulation, in order to guarantee the correctness of the two-dimensional solution. This includes a set of constraints to force that both paths have to be different each other, and a set of constraints to ensure the correct behavior of corrector edges.

5 TWO-DIMENSIONAL DISTRIBUTION WITH VARIABLE TOPOLOGY

In this Chapter, we deal with the general two-dimensional distribution, in which the processors topology might change along the program execution. We assume that the processors geometry can be either one-dimensional or two-dimensional with different grid topologies. The new extension of the CPG is described, as well as the whole problem formulation in terms of 0-1 integer variables. The minimal path problem is reformulated in order to cover both the one-dimensional data mapping and the two-dimensional data mapping.

5.1 MULTIPLE COMMUNICATION-PARALLELISM GRAPHS

A valid data mapping strategy distributes one or two dimensions of the template over a processors grid topology that can be dynamic. The number of available processors p is known at compile time, and it is assumed to be a number power of 2, i.e. $p = 2^q$. Therefore the one-dimensional topology is a $p \times 1$ processors grid, and the two-dimensional topology is any couple $p_1 \times p_2$ processors grid, where $p_1 \times p_2 = p$.

The idea of this model is to build as many CPG as topologies may be considered. This new data structure is named a multiple-CPG. The symbolic information contained in each CPG is identical, but the weights assigned at each edge and hyperedge are different and computed according to the number of processors assumed in the corresponding topology.

For regularity, the one-dimensional data mapping is modeled as a two-dimensional $p \times 1$ processors grid.

Actually, in our implementation we only consider two different topologies: the one-dimensional $p \times 1$ topology, and a squared two-dimensional $p_1 \times p_2$ topology with $p_1 = p_2 = 2^{\frac{q}{2}}$. If q is an odd number, then p_1 is set to $2 \times p_2$. A valid general two-dimensional data mapping strategy has to select one node for each column at each phase, in both CPG copies within a single topology. The nodes selected at each CPG copy within a single topology have to be different each other, and the topology selected at each phase might change between phases if required. One change in the topology of an array requires a redistribution, therefore additional data movement edges have to be inserted in the CPG allowing this kind of change, and estimating the effects of the corresponding data movement.

For instance, Figure 5.1 contains a valid general two-dimensional data mapping strategy, which may be specified with the following HPF data mapping directives:

```
        REAL A(N, N), B(N, N), C(N, N), D(N, N), E(N, N)
!HPF$   PROCESSORS PROC_GRID_1D(P)
!HPF$   PROCESSORS PROC_GRID_2D(P1, P2)
!HPF$   TEMPLATE T1(N, N), T2(N, N), T3(N, N)
!HPF$   DYNAMIC T2(N, N)
!HPF$   ALIGN A(i, j) WITH T1(j, i)
!HPF$   ALIGN B(i, j) WITH T1(j, i)
!HPF$   ALIGN C(i, j) WITH T2(i, j)
!HPF$   DISTRIBUTE T1(BLOCK, *) ONTO PROC_GRID_1D
!HPF$   DISTRIBUTE T2(BLOCK, *) ONTO PROC_GRID_1D
!HPF$   ALIGN D(i, j) WITH T3(i, j)
!HPF$   ALIGN E(i, j) WITH T3(j, i)
!HPF$   DISTRIBUTE T3(BLOCK, BLOCK) ONTO PROC_GRID_2D
        ··· i-th phase ···
!HPF$   REDISTRIBUTE T2(BLOCK, BLOCK) ONTO PROC_GRID_2D
        ··· j-th phase ···
```

In this case, the second dimension of arrays A and B, and the first dimension of array C in the first phase are aligned and distributed on a one-dimensional processors grid of p processors. Then the array C is redistributed, and the first dimension of arrays C and

D, and the second dimension of array E in the second phase are aligned and distributed on the first dimension of a $p_1 \times p_2$ processors grid, and the second dimension of arrays C and D, and the first dimension of array E are aligned and distributed on the second dimension of the same two-dimensional processors grid.

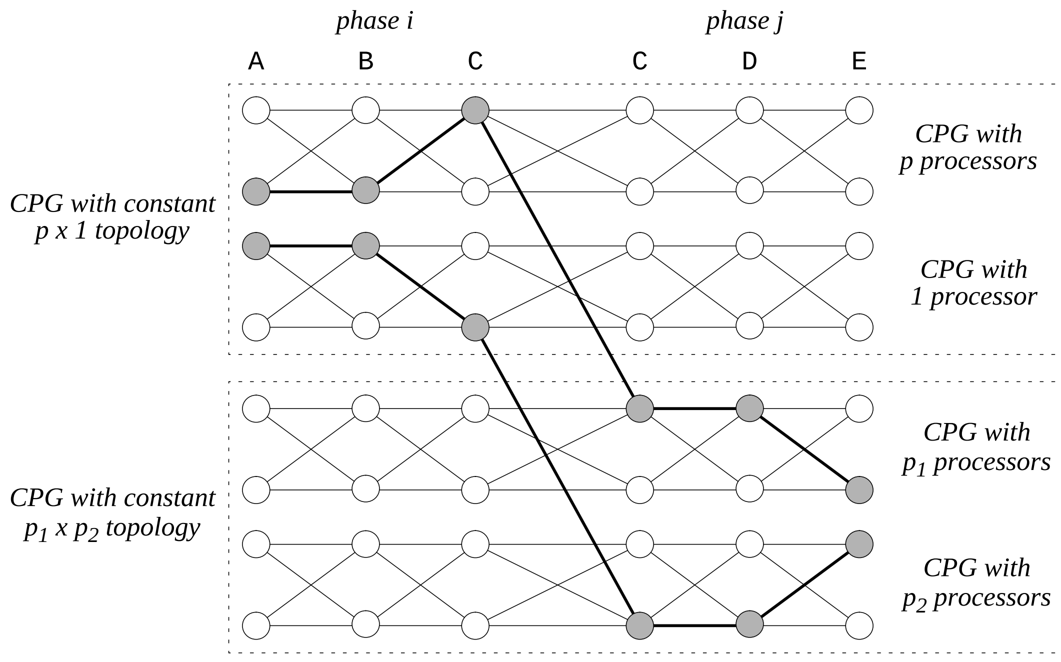

phase i
phase j
A
B
C
C
D
E
CPG with constant p x 1 topology
CPG with constant p_1 x p_2 topology
CPG with p processors
CPG with 1 processor
CPG with p_1 processors
CPG with p_2 processors

Figure 5.1 Valid solution in a general two-dimensional CPG.

Note that in the first phase all selected nodes belong to the one-dimensional topology, while in the second phase all selected nodes belong to the two-dimensional topology. In addition, nodes selected in one CPG copy within a given topology are different than nodes selected in the other CPG copy of the same topology.

5.1.1 Redistribution Information

In the general two-dimensional data mapping, the processors topology can change if desired. As usual, this change is allowed only between phases. Therefore, the CPG has to provide the required information to model this possibility. This information is included in terms of redistribution edges. These edges have to reflect the data movement effects of changing the topology.

For instance, if an array used in phase i is used in a later phase, say phase j, realignment information is added connecting all nodes of this array from phase i to phase j as explained in Chapter 3. If one of these edges is selected, the topology is maintained, and the alignment can change. In addition, redistribution information is inserted to allow a change in the processors topology. This information is included connecting all nodes of one array at phase i of a topology, to all nodes of the same array at phase j of the other topology. The set of realignment and redistribution edges can be seen in Figure 5.2.

The first set of edges connect all nodes of one array in the first CPG copy of one topology, to all nodes of the same array in the first CPG copy of the other topology. Similarly, another set of edges connect all nodes of one array in the second CPG copy of one topology, to all nodes of the same array in the second CPG copy of the other topology.

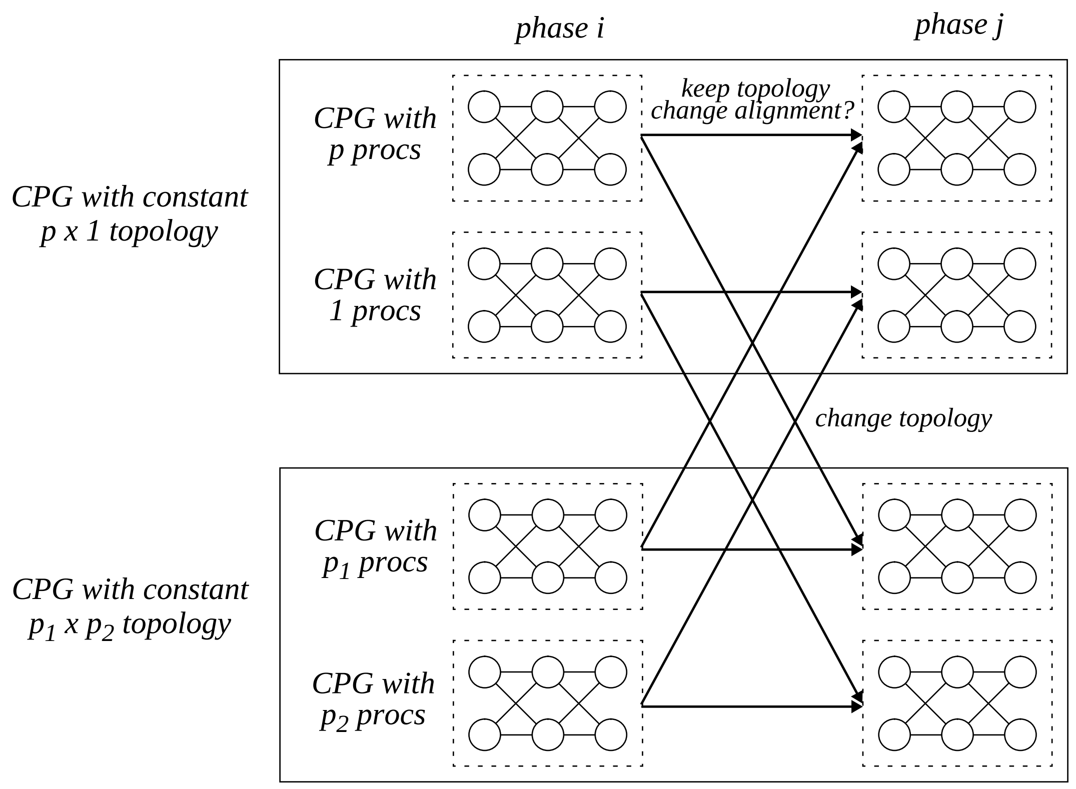

phase i
phase j
CPG with constant p x 1 topology
CPG with p procs
CPG with 1 procs
keep topology change alignment?
change topology
CPG with constant p_1 x p_2 topology
CPG with p_1 procs
CPG with p_2 procs

Figure 5.2 Realignment and redistribution connections from the $p \times 1$ topology.

The set of realignment and redistribution edges leaving the first copy of the first topology can be seen in detail in Figure 5.3.

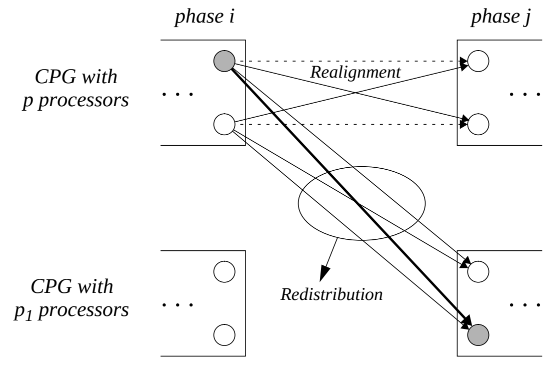


Figure 5.3 Set of edges included in realignment and redistribution connections.

Note that according to our model, when an array changes the distribution topology, all dimensions of this array have to be redistributed. This means that if one dimension of one array aligned to the first dimension of the processors grid in a topology has to be redistributed, the array dimension aligned to the second dimension of the processors grid has to be redistributed as well. In addition, it is not necessary to connect the first CPG copy in the first topology to the second CPG copy in the second topology. As can be seen in Figure 5.3, realignment can be performed at the same time than redistribution with a single set of redistribution edges.

5.2 COST MODEL FOR REDISTRIBUTIONS

The weight assigned to each edge is a symbolic expression representing a *Redistribution* data movement primitive. Note that this cost is the same regardless the array changes the alignment or not.

According to our cost model in which two-dimensional data movement is decomposed into sub-patterns, and each sub-pattern is matched with a single data movement primitive, the cost of a *Redistribution* data movement primitive is equivalent to a $Many\_to\_Many$ data movement primitive at each array dimension. Therefore, the weight assigned at each edge is a function of the block size at each processor, i.e. the total array size divided by the total number of processors.

5.3 MODELING THE PROBLEM

At this point, the multi-CPG is composed by two topologies, each of which contains two CPG copies. The definition of a valid solution for the general two-dimensional data mapping is the following:

A valid solution has to select a single topology for each phase, although the topology selected might change between phases. For a given phase and a given topology, two paths have to be selected, one at each CPG copy within the topology. The path has to include one node for each column in the phase, and the nodes selected have to be different.

In order to distinguish the CPG copy a variable belongs to, the CPG superscript will contain two digits: the first digit a for $a \in 1, 2$ references the topology number, and the second digit b for $b \in 1, 2$ references the CPG copy within the corresponding topology. Therefore, let $Y^{ab}_{PQ}[i, j]$ be the variable associated to the edge connecting node i of column P to node j of column Q, in the $b - th$ CPG copy of the $a - th$ topology. Similarly, let $V^{ab}_{PQ}[i, j]$ be the variable associated to the redistribution edge connecting node i of column P in the $b - th$ CPG copy of the $a - th$ topology to node j of column Q in the $b - th$ CPG copy of the other topology. Even when redistribution edges behave like regular data movement edges, the set of 0-1 integer variables associated to redistribution edges is called V for simplicity in the subscripts. Note that the superscript ab denotes the source CPG copy of the edge, therefore let $a'b$ denote the destination CPG copy, where $a' = (3 - a)$. Finally, let Z^{ab}_m represent the variable associated to the $m - th$ hyperedge in the $b - th$ CPG copy of the $a - th$ topology. In terms of corrector edges, the first CPG topology represents a $p \times 1$ grid, so no corrector edges for this topology are required. Therefore let W_m denote the variable associated to the $m - th$ corrector edge in the second topology.

The original sets of constraints defined in Chapters 3 and 4 have to be modified in order to guarantee the correctness of the general two-dimensional data mapping, therefore the model will be rewritten in the following Sections. This model is valid for both one-dimensional and two-dimensional data distributions.

5.3.1 New Set of Constraints

To ensure the correctness of the solution in the general two-dimensional data mapping, the following sets of constraints have to be specified:

- $C1$ - The solution is a couple of paths.
- $C2$ - Each path visits one node in each column.
- $C3$ - All edges connecting selected nodes have to be included in the solution.
- $C4$ - All hyperedges whose nodes are all selected have to be included in the solution.
- $C5$ - All corrector edges connecting selected hyperedges have to be included in the solution.
- $C6$ - Nodes selected in each path are distinct.
- $C7$ - Both paths belong to the same topology.

Constraints $C1 - C6$ are basically the same than in previous Chapters, but adapted to the structure of the multi-CPG. The main issue that has to be considered in this model is that some columns P in the CPG^{ab} copy are connected to column Q in the same CPG^{ab} copy throughout the set of edges Y_{PQ}^{ab} and to column Q in the $CPG^{a'b}$ copy throughout the set of edges V_{PQ}^{ab}. Now a valid path has to choose one out of both set of edges.

Constraints $C1$ guarantee that a path in a CPG copy has to be connected. Thus for each column Q connected to more than one column P and R, if one edge leading to a node in Q is selected in the set Y_{PQ}^{ab} or in the set $V_{PQ}^{a'b}$ when it exists, one edge leaving this same node must be selected in the set Y_{QR}^{ab} or in the set V_{QR}^{ab} when it exists.

In terms of the variables and their values, it can be stated at each CPG^{ab} copy, that for each node i of each column Q connected to more than one column P and R, the sum of the values of variables associated to the edges that connect this node to column P, must be equal to the sum of the values of variables associated to the edges that connect this node to column R.

$$\sum_{j=1}^{d} Y_{PQ}^{ab}[j,i] + V_{PQ}^{a'b}[j,i] \;=\; \sum_{j=1}^{d} Y_{QR}^{ab}[i,j] + V_{QR}^{ab}[i,j]; \qquad \forall i,a,b$$

This must be accomplished for each pair of sets $Y_{PQ} - Y_{QR}$ with a column in common.

As in the previous Chapter, constraints $C2$ and $C3$ can be specified together. These can be modeled forcing one edge to be selected in a single dimension of any topology, for each non empty set of edges Y_{PQ}^{ab} and V_{PQ}^{ab}.

This can be stated, in terms of variables and their values, posing that the summation of each non empty set of edges Y_{PQ}^{ab} and V_{PQ}^{ab} in each dimension b of both topologies a has to be equal to one.

$$\sum_{a=1}^{2} \sum_{i=1}^{d} \sum_{j=1}^{d} Y_{PQ}^{ab}[i,j] + V_{PQ}^{ab}[i,j] \;=\; 1; \qquad \forall b \in \{1,2\}$$

This must be accomplished for each set of edges Y_{PQ}.

Constraints $C4$ ensure the correct behavior of the hyperedges at each CPG^{ab} copy. The hyperedge belongs to the selection if all nodes linked by it have been selected. According to this model a node i in column P is selected in CPG^{ab} if one of the edges $Y_{PQ}^{ab}[i,j]$ or $V_{PQ}^{ab}[i,j]$ that connect it to any other column Q has been selected.

Assume that hyperedge Z_k^{ab} connects h nodes in CPG^{ab}, say nodes $i^1 \cdots i^h$ in columns $P^1 \cdots P^h$ respectively. It can be stated, in terms of variables and their values, that:

$$\begin{array}{l} \sum_{j=1}^{d} Y_{P^1Q^1}^{ab}[i^1,j] + V_{P^1Q^1}^{ab}[i^1,j] \;\geq\; Z_k^{ab} \\ \cdots \\ \sum_{j=1}^{d} Y_{P^hQ^h}^{ab}[i^h,j] + V_{P^hQ^h}^{ab}[i^h,j] \;\geq\; Z_k^{ab} \end{array}$$

for each hyperedge k at each CPG^{ab} copy.

Constraints $C5$ ensure the correct behavior of each corrector edge. Note that corrector edges link hyperedges only between CPG^{21} and CPG^{22}. In terms of variables and their values, let W_m be the variable associated to the corrector edge connecting hyperedge ${Z^{21}}_p$ to ${Z^{22}}_q$. It can be stated that:

$$\begin{aligned} {Z^{21}}_p &\geq W_m \\ {Z^{22}}_q &\geq W_m \\ {Z^{21}}_p + {Z^{22}}_q &\leq W_m + 1 \end{aligned}$$

for each corrector edge between CPG^{21} and CPG^{22}.

Constraints $C6$ ensure that paths do not have nodes in common, or in other words, that an array dimension is distributed at most once. This can be managed ensuring that the summation of edges connecting each node in all CPG^{ab} to any other column, is lower or equal than one.

In terms of variables and their values, for each node i in column P connected to another column Q by Y^{ab}_{PQ} or V^{ab}_{PQ}, the summation of the values of the variables associated to the edges of all CPG^{ab} copies that connect this node to column Q has to be lower or equal than one.

$$\sum_{a=1}^{2}\sum_{b=1}^{2}\sum_{j=1}^{d} Y^{ab}_{PQ}[i,j] + V^{ab}_{PQ}[i,j] \leq 1; \qquad \forall i \in \{1..d\}$$

for each column P in the multi-CPG.

And finally, constraints $C7$ guarantee that both selected paths belong to the same topology. This can be modeled, for each topology, ensuring that the summation of edges in one dimension b of a topology, equals the summation of edges in the other dimension b' of the same topology.

In terms of variables and their values, for each set of edges Y^{ab}_{PQ} and for each topology a, the sum of the values of variables associated to the edges in CPG^{ab} must be equal to the sum of the values of the edges in $CPG^{ab'}$.

$$\sum_{i=1}^{d}\sum_{j=1}^{d} Y_{PQ}^{ab}[i,j] \;=\; \sum_{i=1}^{d}\sum_{j=1}^{d} Y_{PQ}^{ab'}[i,j]; \qquad \forall a \in \{1,2\}$$

This must be accomplished for each set of edges.

5.4 AN EXAMPLE: GENERAL TWO-DIMENSIONAL DISTRIBUTION FOR THE ADI KERNEL

If we consider again the Alternate Direction Implicit (ADI) integration kernel and look for the optimal solution, the tool will suggest a two-dimensional data mapping like the one illustrated in Chapter 4. In this Section, the original ADI kernel is slightly modified in order to provoke a different optimal solution.

With this purpose, the three initialization phases are included within an iterative loop that provides more computational weight, and a data flow dependence is added in the nested j loop within the second phase. Therefore this j loop is not candidate to be parallelized. With this change, we intend to give a higher weight to the initialization phases, and force a row distribution for them.

According to the general two-dimensional model, two different topologies are built, each one consisting of two CPG copies. Assuming that the target system has 32 processors available, the first topology corresponds to a 32×1 processors grid, and the second topology corresponds to a 8×4 processors grid. As usual, the bandwidth assumed is 1Mbyte per second.

The weights for edges and hyperedges in the 32×1 topology are the same than in Chapter 3, and the weights of edges, hyperedges, and corrector edges in the 8×4 topology are the same than in Chapter 4. However, in this model there are several redistribution edges connecting both topologies. The cost of a redistribution between any couple of CPG copies considers all 32 processors, and it is the same for all arrays. This can be computed as:

$$\frac{256 \times 256}{32} \times 8 \times \frac{1}{1000000} \;=\; 0.01638\ secs$$

The resulting graph is used to model the minimal path problem with 0-1 integer variables. The characteristics of the general two-dimensional data mapping model are summarized in Table 5.1. The computation time spent to find the optimal solution increases up to 7.5 seconds [1].

	General Model
Number of edges	1088
Number of hyper	36
Number of 0-1 variables	1124
Number of constraints	767
Computation time	**7.5 secs**

Table 5.1 CPG complexity for the ADI program in the general two-dimensional data mapping model.

In this case, the optimal solution derived by the tool is a row distribution for the three initialization phases, and a two-dimensional distribution for the remainder phases. The optimal data mapping strategy for phases 3 and 4 can be seen in Figure 5.4. In the Figure, superscripts in CPG^{ab} specify the $b-th$ copy of the $a-th$ topology. Note that a change of topology, i.e. redistribution, is required for all arrays used in both phases 3 and 4.

[1]Times have been obtained executing on a Sun SuperSparc 20

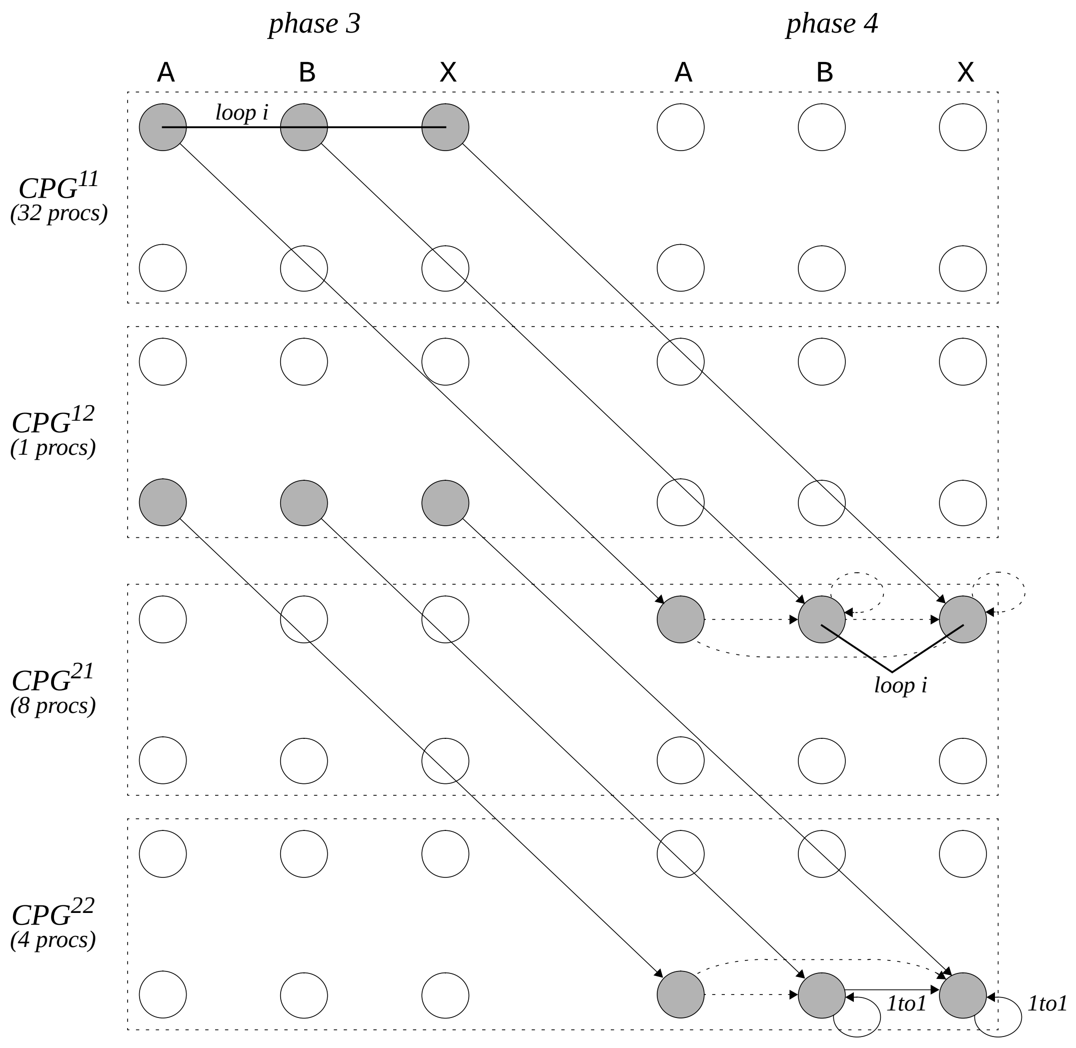


Figure 5.4 Optimal one and two-dimensional data mapping for phases 3 and 4 of the ADI code.

5.5 SUMMARY

In this Chapter we have described the Communication-Parallelism Graph that supports general two-dimensional data mappings, in which the processors grid topology can change during the execution of the program. The idea is that a different CPG is built for each topology considered, and redistribution edges that allow a change of topology are inserted between phases in the CPG.

This new CPG with multiple copies of the original one, is called multi-CPG. Although our implementation is limited to two different topologies, the idea of the model can be extended to more topologies.

In addition, we have redefined the linear 0-1 integer programming formulation of the minimal path problem in order to support both the one-dimensional and the two-dimensional models.

6 EXPERIMENTAL RESULTS

In this Chapter we describe some experiments performed in order to illustrate several aspects of the tool. First we intend to show the accuracy of the predictions of the tool. This is performed by executing the parallelized code on a real parallel machine and comparing the measured times to the estimated ones. Second, the complexity of the approach will illustrated in terms of computation time spent to find the optimal solution. Solutions are computed for both one-dimensional and two-dimensional data mappings.

6.1 PERFORMANCE PREDICTION TOOL

In order to validate the accuracy of the prediction of the tool, some of the solutions generated have been compared to the actual execution of the parallelized program on a Silicon Graphics ORIGIN 2000 with 32 processors. The ORIGIN 2000 is a non-uniform memory access distributed-memory multiprocessor with a high capacity 4 Mbytes cache memory for each processor, that may act as a first level distributed memory. We have performed several experiments, trying different data mapping strategies, changing the number of processors, and changing the data sizes in some programs.

In all predictions, we have assumed a bandwidth of 10^8 bytes per second, unless otherwise stated. Profiling information has been obtained executing the sequential code on a single processor of the same ORIGIN system.

The run time behavior of the parallelized code depends on the data mapping and parallelization strategy selected for the source program, on some architectural parameters of the target machine, but also on the compiler capabilities. To obtain accuracy in our predictions, the parallel code has been generated by hand in order to avoid compiler optimizations not supported in our model.

The programs selected for these experiments are the Alternating Direction Implicit (ADI) integration kernel, the ERLEBACHER program written by Thomas M. Eidson at the Institute for Computer Applications in Science and Engineering (ICASE), and the SHALLOW benchmark weather prediction program from the xHPF benchmark set [1], written by Paul N. Swarztrauber from the National Center for Atmospheric Research. For the purpose of this evaluation, programs ERLEBACHER and SHALLOW have been inlined, since our tool does not perform inter-procedural analysis.

6.1.1 ADI

The Alternating Direction Implicit (ADI) integration kernel has been used in several Sections of this Thesis as an example code. The original source code can be found in Appendix A.

The ADI kernel has three initialization phases, followed by an iterative loop with six more phases within it. The first three phases within the iterative loop contain neighbor data movement across the second array dimension, and can be parallelized by rows. Alternatively the remainder three phases within the iterative loop contain neighbor data movement across the first array dimension, and can be parallelized by columns.

Instead of using an HPF compiler that may apply some optimizations not dealt by our model, we have implemented by hand the parallel single program multiple data version of several data mapping strategies for the ADI kernel, according to our model. The strategies considered are:

[1] The xHPF benchmark set is available by anonymous ftp at ftp.infomall.org in directory tenants/apri/Bench

- *Row* static distribution, which allows the parallelization of the three initialization phases, and the first three phases within the iterative loop.
- *Column* static distribution, which allows the parallelization of one initialization phases, and the last three phases within the iterative loop. When the cost of moving data is high, both *row* and *column* strategies are effective.
- *Dynamic* data distribution, which allows the parallelization of one loop in each phase, but requires data redistribution inside the iterative loop. When the data movement cost is low, this data distribution strategy is the most effective.
- *Two-dimensional* $p_1 \times p_2$ static data distribution, which allows the parallelization of one loop in each phase. Loops that parallelize rows are executed with p_1 processors, and loops that parallelize columns are executed with p_2 processors.

In Appendix B there is a description of the single program multiple data program used in the executions.

The run time behavior of these implementations are listed in Table 6.1. All units in the Table are expressed in seconds. The prediction has been performed using a profiled sequential time of 13.793 seconds. The size of the matrices is 256×256, and the number of iterations of the outer loop has been changed to 100 in order to obtain a better resolution.

		2	**4**	**8**	**16**	**32**
ROW	Measured	9.90	8.87	8.75	8.22	15.11
	Predicted	10.82	9.32	8.58	8.20	8.02
COL	Measured	9.97	7.90	6.93	6.70	6.64
	Predicted	9.88	7.91	6.93	6.43	6.19
DYN	Measured	6.85	4.00	2.26	1.18	1.85
	Predicted	7.68	4.03	2.07	1.05	0.52
2-D	Measured	9.97	6.63	4.13	3.73	2.19
	Predicted	9.88	6.89	3.94	3.48	1.88

Table 6.1 Comparison of measured and predicted execution times for *row*, *column*, *dynamic*, and two-dimensional data mapping.

We have implemented the code parameterized for 2, 4, 8, 16, and 32 processors. According to our model, two-dimensional data distributions have been generated such that processors are apportioned as equally as possible across each distributed dimension.

Note that all predictions are within a 10% of the actual measured execution time, except codes that incur in false sharing. False sharing occurs in executions with 32 processors and when matrices are distributed by rows. These codes are the one-dimensional row data distribution, and similarly, the dynamic data mapping in phases where arrays are distributed by rows.

In addition, we have compared the execution times of our data mapping strategy to the optimal solution found by the POWER FORTRAN Accelerator, the native SGI source-to-source optimizing FORTRAN preprocessor that discovers parallelism in FORTRAN codes and converts those programs to parallel code. The results of this comparison is shown in Table 6.2. Note that our optimal solution reduces the execution time between 20% and 70% with the ADI code.

	2	**4**	**8**	**16**	**32**
POWER FORTRAN Accelerator	8.84	5.07	3.32	2.67	6.18
Dynamic Mapping (measured)	6.85	4.00	2.26	1.55	1.83
Improvement	22.5%	21.1%	31.9%	41.9%	70.3%

Table 6.2 Comparison between the parallelization strategy proposed by PFA and the one selected by our tool.

6.1.2 ERLEBACHER

The ERLEBACHER program is a 3D tridiagonal solver based on the ADI integration kernel. The inlined program consists of 38 phases that perform symmetric forward and backward computations along each dimension of four main 3-dimensional arrays. In the first computation flow, data is moved across the first array dimension. Therefore parallelism exists along the second and third dimensions. Similarly, the second and third computation flows move data across the second and third dimensions respectively.

There are several equivalent parallelization strategies as long as the computation along each dimension is symmetric. However and due to the profiled sequential execution time, the parallelization strategy suggested by our tool is to distribute the third dimension of the arrays in the first and second computation flows, and to distribute the second dimension of the arrays in the third computation flow.

The resulting predicted and measured parallel times of the optimal data distribution strategy using 2, 4, 8, 16, and 32 processors can be seen in Figure 6.1. All times are expressed in seconds. Predicted parallel times have been performed using a profiled sequential execution time of 0.873 seconds. Note that predicted times in this example are within a 5% of the actual measured execution times.

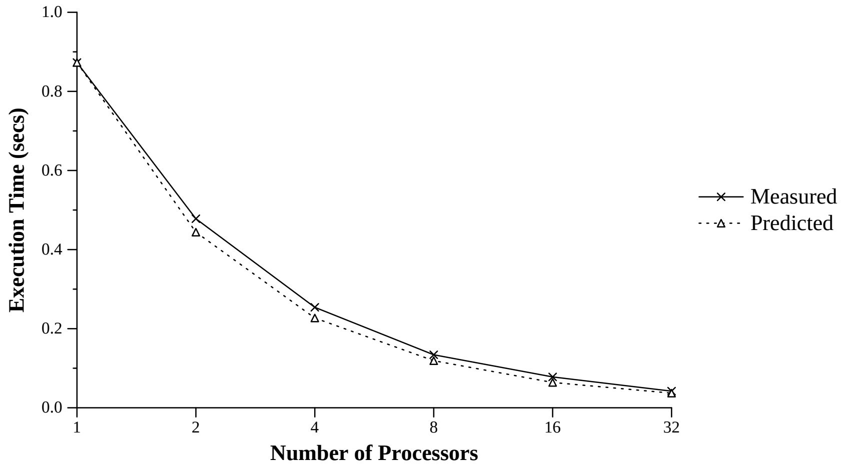


Figure 6.1 Predicted and measured execution times for the ERLEBACHER program with 2, 4, 8, 16, and 32 processors.

6.1.3 SHALLOW

The SHALLOW water equations model is a 512 lines code, with a set of 512×512 sized arrays. The inlined program consists of 27 phases, most of them within an iterative loop of $NCYCLES$ iterations. The computation is composed by sets of three phases: a loop nest that computes the core of some two-dimensional arrays, one loop that computes the first and last rows, and one loop that computes the first and last columns. All these loops are parallelizable, and contains nearest-neighbour data movements.

A static two-dimensional squared data mapping would reduce the amount of data to be moved around, however, the computation of loops that traverse single rows or columns will be parallelized by the root of the number of processors. Alternativelly, and according to our model, both the row and column static data mappings require more data movement overhead, but can slightly reduce the computation time. Due to the profiling issues, the optimal data mapping suggested by our tool is the static distribution of the second dimension of all the arrays.

The resulting predicted and measured parallel times of the optimal data distribution strategy using 2, 4, 8, 16, and 32 processors can be seen in Figure 6.2. All times are expressed in seconds. Predicted parallel times have been performed using a profiled sequential execution time of 45.152 seconds. Note that predicted times in this example are within a 5% of the actual measured execution times.

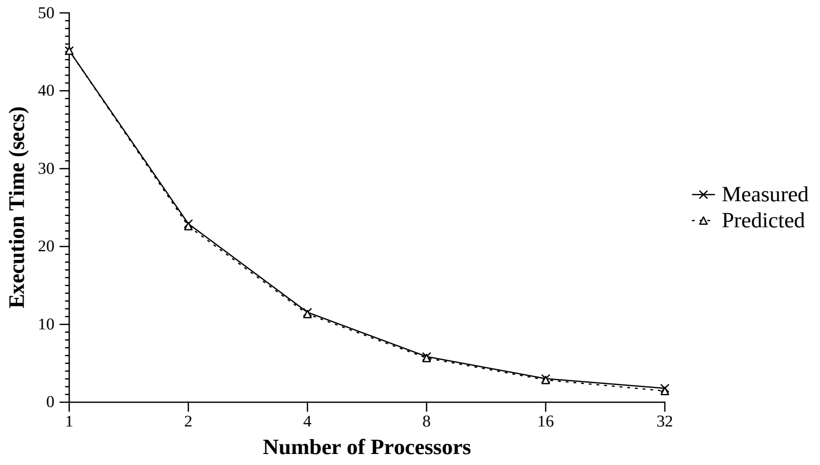


Figure 6.2 Predicted and measured execution times for the SHALLOW program with 2, 4, 8, 16, and 32 processors.

6.2 COMPLEXITY OF THE APPROACH

In this Section we intend to illustrate the complexity of our model in terms of computation time required to find the optimal data mapping strategy. In addition to the ADI integration kernel, the ERLEBACHER program, and the SHALLOW benchmark, we have selected programs *baro*, *tomcatv*, and *x*42 from the xHPF benchmark set, and the routine *rhs* which is the more time consuming routine from the APPSP NAS benchmark, also included in the xHPF benchmark set. Program *baro* has been inlined, and routine *rhs* has been transformed to a single program. All times presented in this Section have been obtained using a Sun SuperSparc 20.

The set of programs selected are listed in Table 6.3. The table includes information about the number of code lines, the total number of loops, the number of loops candidate to be parallelized, the number of phases in each program, the number of different arrays and their dimensionality, and the number of different reference patterns between arrays. These characteristics are the parameters that determine the complexity of the final optimization problem.

Program	**Lines**	**Loops**	**Parall**	**Phases**	**Arrays**	**Dims**	**Patts**
adi	50	15	10	9	3	2	48
erle	449	83	68	41	13	3	151
rhs	535	37	37	4	4	4	24
tomcatv	178	18	9	11	7	2	77
*x*42	302	36	29	19	19	2	196
baro	1153	98	86	24	38	2	428
shallow	365	39	38	27	14	2	282

Table 6.3 Characteristics of the selected programs.

Table 6.4 shows the number of 0-1 integer variables and the number of constraints required to formulate the minimal path problem for one-dimensional data mappings. In addition, the last column shows the total CPU time (in seconds) required to find the optimal solution. All these computations have been performed asssuming 16 processors and a bandwidth of 10^8 bytes per second.

Program	**edges**	**hyper**	**constr**	**time**
adi	160	10	164	0.50
erle	1359	68	804	4.90
rhs	336	37	176	0.70
tomcatv	248	10	249	0.80
*x*42	700	29	760	4.20
baro	1484	83	1608	15.80
shallow	936	38	1004	6.60

Table 6.4 Characteristics of the one-dimensional model.

Note that all steps of the data mapping process (including remapping) have been performed with the specified time. The more time consuming application is *baro* with 15.8 seconds. *shallow* and *erlebacher* take 6.6 and 4.9 seconds respectively, and all other programs are below one second.

In the two-dimensional data mapping model assuming a constant topology, the number of 0-1 integer variables is duplicated. However the computation time spent in finding the optimal solution increases exponentially. In Table 6.5 the number of edges, hyperedges, corrector edges, and constraints, and the total computation time spent to find the optimal solution can be seen.

Program	**edges**	**hyper**	**correct**	**constr**	**time**
adi	320	20	2	386	1.60
erle	2718	136	74	2014	1542
rhs	672	74	138	543	77.70
tomcatv	496	20	2	583	2.50
*x*42	1400	58	30	1775	329
baro	2968	166	66	3703	8 hours
shallow	1872	76	22	2335	2 days

Table 6.5 Characteristics of the two-dimensional model with constant topology.

Note that the computation time does not depend only on the number of integer 0-1 variables nor on the number of constraints. In this model, the minimal path problem structure is much harder to solve. For instance, the number of variables and constraints in the two-dimensional model for *rhs* is lower than in the one-dimensional model for *erlebacher*. However the computation time in the two-dimensional model of *rhs* is 15 times higher than the one-dimensional model of *erlebacher*. Obviously, the time required to solve *adi*, *tomcatv*, *rhs*, or even *x*42 is acceptable. All other times are out of range. After Table 6.6, we will show how to reduce these times.

Finally, in Table 6.6 the characteristics on the general two-dimensional model is shown, but only for programs that turns out to be feasible in terms of time. As usual, times are expressed in seconds.

Program	**edges**	**hyper**	**correct**	**constr**	**time**
adi	1088	40	2	795	7.90
rhs	2048	148	138	922	1 hour
tomcatv	1536	40	2	1168	16.30
*x*42	4304	116	30	3481	5 hours

Table 6.6 Characteristics of the general two-dimensional model for a subset of the selected programs.

We have observed that our general purpose linear 0-1 integer programming solver tends to find the optimal solution at the beginning of the search, although it requires many more iterations to explore the whole search space. The number of iterations performed by the solver can be provided by the user as a run-time parameter to limit the computation time. In Table 6.7 the amount of seconds spent by the solver to find a good solution is shown. In addition, we show the optimality of the data mapping strategy found.

Note that in this case, program *baro* spends less than 20 minutes in finding a solution with an estimated performance of 87% the estimated performance of the optimal solution. With a little bit more than 12 minutes, a solution is provided for *erlebacher* with an optimality of 82%. Programs *shallow* and *x*42 require 5 and 3 minutes respectively to obtain solutions with an optimality of 88% and 92%. Program *rhs* finds the optimal solution with only 23.8 seconds, compared to one hour required when no limit is provided to the solver. The other programs find the optimal solution with a matter of seconds.

Program	**time**	**optimality**
adi	7.8	100%
erle	770	82%
rhs	23.8	100%
tomcatv	14.3	100%
x42	197	92%
baro	1191	87%
shallow	327	88%

Table 6.7 Behavior of the set of selected programs when limiting the number of iterations of the solver.

6.3 SUMMARY

In this Chapter we have illustrated the accuracy of our model in terms of predictability tool. The original sequential codes have been transformed by hand in order to force the execution to behave as assumed in our model, and thus avoiding undesired optimizations performed by the HPF compiler. We have shown the precision of the predictions with the ADI integration kernel, the ERLEBACHER program, and the SHALLOW benchmark. In addition, we have compared the data mapping strategy suggested by our tool for the ADI code to the optimal solution generated by the Power Fortran Accelerator, the native SGI source-to-source optimizing FORTRAN preprocessor. The results of this comparision demonstrate the usefulness of our tool.

We have also shown the complexity of the approach in terms of computation time spent to find the optimal solution. The one-dimensional model provides the optimal solution very fast, while the time spent in the two-dimensional model increases exponentially. We have observed that the linear 0-1 integer programming solver tends to find the optimal solution at the beginning of the search, although it requires many more iterations to explore the whole search space. Therefore we propose to reduce the computation time required to find the optimal solution by limiting the number of iterations performed. The behavior of this limitation has also been shown.

7 CONCLUSIONS AND FUTURE WORK

Automatic data mapping in the context of a parallelizing environment for massive parallel processors systems is an important topic of current research. The choice of a good data distribution is important as it determines the amount of remote data accesses and the potential parallelism in a data parallel program. The optimal data distribution depends on the program structure, the compiler capabilities, characteristics of the target machine, and the program's data sizes.

Due to their influence on the amount of inter-processor communication, the choice of data mapping and remapping have a significant impact on the performance of the parallel program. In addition, there is often a trade-off between minimizing interprocessor data movement and load balancing on processors. Crucial aspects such as data movement, parallelism, and load balance, have to be taken into consideration in a unified way to efficiently solve the data distribution problem.

Our thesis proposes a new framework for an automatic data mapping tool in the context of a parallelizing environment for massive parallel processor systems. The applications considered for parallelization are usually regular problems, in which data structures are dense arrays. The tool analyzes Fortran 77 programs and decides a data mapping and parallelization strategy for this program. This data mapping is used to annotate the original sequential Fortran program using HPF data mapping and loop parallelization directives.

The data mapping strategy generated can be static or dynamic, one or two-dimensional, with either `BLOCK` or `CYCLIC` distribution fashion. The solution suggested takes into account the effects of control flow statements between phases, and includes the alignment and distribution of all the arrays at each phase, a set of remapping actions between phases when profitable, and the loop parallelization strategy induced by the data mapping.

One-dimensional Communication-Parallelism Graph

We have defined a new data structure named Communication-Parallelism Graph (CPG), that contains all the information required to consider in an unified way all steps of the data mapping problem. This single data structure holds all data movement and parallelism information inherent in each phase of the program, plus additional information between phases denoting the data movement cost occurred if the distribution of one array in one phase is different than the distribution of the same array in another phase.

Nodes in the CPG are organized in columns. In order to build the CPG, programs are initially divided into phases, and each array in a phase defines a column with as many nodes as the array dimensionality. Therefore, each node represents a distributable dimension of one array. Over this set of nodes, data movement information is added in terms of edges. One edge connects two nodes if the alignment and distribution of these two nodes has effects related to data movement. Similarly, parallelism information is added in the CPG in terms of hyperedges. All array dimensions that have to be distributed for a loop to be parallelized are linked to a hyperedge.

A valid data mapping is made up of one node from each column. The data mapping strategy associated is the distribution of the array dimension associated to the selected nodes. The effects of this data mapping strategy are determined by the set of edges and hyperedges that remain inside the selected set of nodes. Edges express data movement and hyperedges express parallelization. Therefore the execution time of the parallelized program is estimated as the sequential execution time, plus the time overhead of data movement, minus the time saved due to parallelization.

Two-dimensional Communication-Parallelism Graph

Given a two-dimensional processors grid topology, the two-dimensional data mapping model builds two CPG copies. Each copy is associated to one dimension of the processors grid, and costs are computed according to the number of processors assigned to that dimension. In this case, the cost model has to be modified with respect to the one-dimensional case. A valid two-dimensional data mapping has to select one node from each column of each CPG copy, with the additional restriction that nodes selected in one CPG copy have to be different than nodes selected in the other CPG copy.

If the data-mapping has to support different processors grid topologies, two CPG copies are built for each two-dimensional topology modeled. The name of this set of CPG copies in multi-CPG. All arrays within a phase have to be distributed with the same topology, but this can change between phases. Therefore additional data movement information is added connecting arrays between phases at different topologies. In this case, a valid two-dimensional data mapping has to select one node from each column within a single topology, with the additional restriction that nodes selected in one CPG copy have to be different than nodes selected in the other CPG copy. The topology selected at each phase can change between phases.

Optimization Problem Model

The CPG contains all data movement and parallelism related information of a whole program. Assuming a number of processors, all symbolic information included in the CPG is replaced by its estimated cost, computed in seconds. Therefore the CPG contains all its weights in seconds. The optimal data mapping is the one that minimizes the summation of data movement cost minus the summation of parallelization benefits.

Linear programming provides some techniques to solve some optimization problems. In our framework, we translate the whole data mapping problem into a single minimal path problem with a set of additional constraints that ensures the correctness of the solution. The techniques used guarantee that the solution provided is optimal, therefore heuristic algorithms are avoided. A general purpose linear 0-1 integer programming solver is used to find the optimal solution.

The solution derived includes the alignment and distribution of each array at each phase, plus remapping actions between phases if profitable, plus the set of loops that have to be parallelized according to the data mapping strategy. All this information is computed independently by many other authors. However, in our framework this information is exactly computed in a single step.

Experimental Results

Experimental results illustrate the accuracy of our model in terms of predictability tool. The precision of the predictions using the ADI integration kernel, the ERLEBACHER program, and the SHALLOW benchmark is shown. In addition, we have compared the run time behavior of the optimal data mapping strategy for the ADI code suggested by our tool to the execution of the optimal solution generated by the native SGI source-to-source optimizing FORTRAN preprocessor. Our solution is at least 20% better for all different number of processors evaluated.

In terms of complexity of the approach we have shown the feasibility of the approach with the one-dimensional data mapping model. However, in the two-dimensional model the computation time spent to find the optimal solution increases exponentially, and is some cases the time is very high. We have proposed to reduce this computation time by limiting the number of iterations of the linear 0-1 integer programming solver. This simplification can produce sub-optimal solutions, however we have observed that the solver finds the optimal solution in most cases even when the number of iterations have been limited.

Publications

The work presented in this thesis is based on our experience in the implementation of another automatic data distribution tool, called DDT. In the first implementation [AGG$^+$94], DDT supported static data distributions, including inter and intra-dimensional alignment. The tool was then improved in order to perform inter-procedural analysis, and to support dynamic data distributions [AGG$^+$95]. A whole description of the tool can be found in [AGG$^+$97]. The techniques proposed for distributed memory systems are currently applied to cache-coherent shared memory parallel systems, in which the large cache sizes can perform as a local memory system. This is described in [AGGL96].

In [GAL95b, GAL95a] we presented the CPG as a novel approach towards static automatic data distribution. The mappings supported at that point considered the whole program as a single phase. The minimal path problem was formally specified in [GAL96c]. This basic model was extended in [GAL96b] in order to support dynamic data mappings, and in [GAL96a] it was enhanced in order to model both `BLOCK` and `CYCLIC` distributions, and take into account the effects of control flow statements between phases. In [GAL97] we presented some initial results towards the two-dimensional data mapping assuming that the underlying processors topology is constant.

As a previous work, our group performed an exhaustive study and selection of application codes from different benchmark suites. The results and conclusions have been presented at [ALG$^+$93, ALG$^+$94, ALG$^+$95] Finally, a discussion on current HPF implementations will be published in [PAGT97].

Future Work

Our main interest in the future work is to study techniques to reduce the computation time required to find a solution, keeping the condition that the solution has to be optimal. One approach is the study and implementation of a specific purpose linear 0-1 integer programming solver, taking into account the regularity of our model. Another option is to develop one exhaustive search algorithm with aggressive pruning techniques. Both alternatives will be implemented and compared.

A lot of additional aspects should be considered in the problem formulation in order to improve the accuracy of the model, and therefore the quality of the solutions generated. Data movement optimizations, such as detection and elimination of redundant communication, overlapping of communication and computation, and combination of communication messages have to be included in the model. Other parallelism related optimizations will enhance the functionality of the tool, such as loop transformations to eliminate data dependences, or exploitation of pipelined computations.

The framework can also be extended in order to perform inter-procedural analysis, and to accept both Fortran 77 and Fortran 90 programs. In addition, an automatic code restructuring and generation module may be implemented to automate the whole process.

It would be interesting to integrate this tool into an unified parallelization environment useful enough to assist the programmer in efficiently write his parallel application codes. This environment may be an interactive graphical user interface that provides with more comprehensive information.

Finally, we plan to apply the techniques developed in this work to other parallel system architectures, such as distributed-shared memory systems, in which cache effects can change the behavior of the parallel program. These techniques can also be applied to other computer science fields, such as distributed database systems or special purpose parallel multimedia systems.

REFERENCES

[ACG$^+$94] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crumey, C-W. Tseng, and S. Warren. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, Vol 2(3):48–58, 1994.

[AGG$^+$94] E. Ayguadé, J. Garcia, M. Gironès, J. Labarta, J. Torres, and M. Valero. Detecting and using affinity in an automatic data distribution tool. In K. Pingali et al., editors, *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 61–75, Ithaca, NY, August 1994. Springer-Verlag.

[AGG$^+$95] E. Ayguadé, J. Garcia, M. Gironès, M.L. Grande, and J. Labarta. Data redistribution in an automatic data distribution tool. In C.-H. Huang et al., editors, *Proceedings of the 8th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 407–421, Columbus, OH, August 1995. Springer-Verlag.

[AGG$^+$97] E. Ayguadé, J. Garcia, M. Gironès, M.L. Grande, and J. Labarta. DDT: A research tool for automatic data distribution. *Scientific Programming*, Vol 6(1):73–94, Spring 1997.

[AGGL96] E. Ayguadé, J. Garcia, M.L. Grande, and J. Labarta. Data distribution and loop parallelization for shared-memory multiprocessors. In *Proceedings of the 9th Annual Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996. Sprinver-Verlag.

[AL93] J. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN'93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[ALG$^+$93] E. Ayguadé, J. Labarta, J. Garcia, M. Gironès, and M. Valero. A study of data sets and affinity in the Perfect Club. In H.J. Sips, editor, *Proceedings of the 4th International Workshop on Compilers for Parallel Computers*, pages 5–16, Delft, The Netherlands, December 1993.

[ALG$^+$94] E. Ayguadé, J. Labarta, J. Garcia, M. Gironès, and M. Valero. Detecting affinity for automatic data distribution. In M. Mango Furnari, editor, *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications*, pages 32–46, Capri, Italy, October 1994. World Scientific.

[ALG$^+$95] E. Ayguadé, J. Labarta, J. Garcia, M. Gironès, and M. Valero. Analyzing reference patterns in automatic data distribution tools. *International Journal on Parallel Processing*, Vol 23(6), 1995.

[APR95] Applied Parallel Research, Inc. *xHPF Version 2.0. User's Guide*, second edition, January 1995.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[BKK$^+$94a] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In K. Pingali et al., editors, *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[BKK94b] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Montreal, Canada, August 1994.

[CFZ93] B. M. Chapman, T. Fahringer, and H. P. Zima. Automatic support for data distribution on distributed memory multiprocessor systems. In *Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[CGS93] S. Chatterjee, J.R. Gilbert, and R. Schreiber. The alignment-distribution graph. In U. Banerjee et al., editors, *Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

[CGSS94] S. Chatterjee, J.R. Gilbert, R. Schreiber, and T.J. Sheffler. Array distribution in data-parallel programs. In K. Pingali et al., editors, *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, Vol 2, October 1988.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, Vol 1(1), Fall 1992.

[CON94] CONVEX Computer Corporation. *SPP1000 Systems Overview*, 1994.

[CP93] P. Crooks and R.H. Perrott. An automatic data distribution generator for distributed memory MIMD machines. In *Proceedings of the 4th International Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[CP95] P. Crooks and R.H. Perrott. Language constructs for data partitioning and distribution. *Scientific Programming*, Vol 4(1), Spring 1995.

[DHR93] A. Dierstein, R. Hayer, and T. Rauber. Automatic data distribution and parallelization. In *Proceedings of the 4th International Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[Fea93] P. Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[FHK$^+$90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report CRPC-TR-90-141, Department of Computer Science, Rice University, Houston, TX, December 1990.

[GAL95a] J. Garcia, E. Ayguadé, and J. Labarta. A novel approach towards automatic data distribution. In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995.

[GAL95b] J. Garcia, E. Ayguadé, and J. Labarta. A novel approach towards automatic static data distribution. In *2nd Workshop on Automatic Data Layout and Performance Prediction*, Houston, TX, April 1995.

[GAL96a] J. Garcia, E. Ayguadé, and J. Labarta. Dynamic data distribution with control flow analysis. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996.

[GAL96b] J. Garcia, E. Ayguadé, and J. Labarta. A framework for automatic dynamic data mapping. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, New Orleans, LA, October 1996.

[GAL96c] J. Garcia, E. Ayguadé, and J. Labarta. Using a 0-1 integrer programming model for automatic static data distribution. *Parallel Processing Letters*, Vol 6(1):159–171, 1996.

[GAL97] J. Garcia, E. Ayguadé, and J. Labarta. Two-dimensional data distribution with constant topology. In *3rd Workshop on Automatic Data Layout and Performance Prediction*, Barcelona, Spain, January 1997.

[GB93] M. Gupta and P. Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[Gup92] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, September 1992.

[HPC97] High performance computing and communications: Advancing the frontiers of information technology. A Report by the Committee on Computing, Information, and Communications. National Science and Technology Council, USA, 1997.

[HPF93] High Performance Fortran Forum HPFF. High Performance Fortran language specification. version 1.0. *Scientific Programming*, Vol 2(1 and 2), May 1993.

[HS91] C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. In *Proceedings of the 4th Annual Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[Kar87] A.H. Karp. Programming for parallelism. *IEEE Computer*, May 1987.

[KK95] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995.

[KLS90] K. Knobe, J.D. Lukas, and G.L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, Vol 8, February 1990.

[KMT90] K. Kennedy, K. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope editor. Technical Report CRPC-TR90096, Center for Research on Parallel Computation, Rice University, Houston, TX, October 1990.

[KMT91] K. Kennedy, K.S. McKinley, and C.-W. Tseng. Analysis and transformation in the ParaScope editor. In *Proceedings of the 5th ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KP96] W. Kelly and W. Pugh. Minimizing communication while preserving parallelism. In *Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, PE, May 1996.

[Kre93] U. Kremer. NP–completeness of dynamic remapping. In *Proceedings of the 4th International Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[Kre95] U. Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Center for Research on Parallel Computation, Rice University, Houston, TX, October 1995.

[LC90] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[LC91] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, Vol 2(3), July 1991.

[Lee95] P. Lee. Techniques for compiling programs on distributed memory multicomputers. *Parallel Computing*, Vol 21(12), December 1995.

[LIN94] LINDO Systems Inc. *LINGO Optimization Modeling Language*, April 1994.

[NDG95] Q. Ning, V.V. Dongen, and G.R. Gao. Automatic data and computation decomposition for distributed memory machines. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 1995.

[NW88] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, England, 2nd edition, 1988.

[PAGT97] R.H. Perrott, E. Ayguadé, J. Garcia, and J. Torres. Data parallel language implementations: A survey. *Scientific Programming*, Vol 6(2):(to appear), 1997.

[PB95] D.J. Palermo and P. Banerjee. Automatic selection of dynamic partitioning schemes for distributed-memory multicomputers. In C.-H. Huang et al., editors, *Proceedings of the 8th Annual Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[RS91] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, Vol 2(4), October 1991.

[Sil96] Silicon Graphics Computer Systems SGI. *Power Challenge Technical Report*, 1996.

[SSGC95] T.J. Scheffler, R. Schreiber, J.R. Gilbert, and S. Chatterjee. Aligning parallel arrays to reduce communication. In *Frontiers95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[SSP$^+$95] T.J. Scheffler, R. Schreiber, W. Pugh, J.R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. In C.-H. Huang et al., editors, *Proceedings of the 8th Annual Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995. Springer-Verlag.

[TMC91] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, 1991.

[Tse93] C.-W. Tseng. *An Optimizing Fortran D Compiler for Distributed-Memory Machines.* PhD thesis, Center for Research on Parallel Computation, Rice University, Houston, TX, January 1993.

[Who92] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[XN94] H. Xu and L.M. Ni. Optimizing data decomposition for data parallel programs. In *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1994.

A

ADI: THE ALTERNATE DIRECTION IMPLICIT INTEGRATION KERNEL

```
      program adi
          double precision x(256, 256), a(256, 256), b(256, 256)
          parameter (MAXITER = 10)
c
c         initialize
c
          do i = 1, 256
              a(i, 1) = 0.0
              b(i, 1) = 3.0
              x(i, 1) = 4.0
          enddo

          do j = 2, 255
              do i = 1, 256
                  a(i, j) = 1.0
                  b(i, j) = 3.0
                  x(i, j) = 5.0
              enddo
          enddo

          do i = 1, 256
              a(i, 256) = 1.0
              b(i, 256) = 3.0
              x(i, 256) = 4.0
          enddo
```

```
        do iter = 1, MAXITER
c
c          ADI forward & backward sweeps along rows
c
           do j = 2, 256
               do i = 1, 256
                   x(i, j) = x(i, j) - x(i, j-1) * a(i, j) / b(i, j-1)
                   b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j-1)
               enddo
           enddo

           do i = 1, 256
               x(i, 256) = x(i, 256) / b(i, 256)
           enddo

           do j = 255, 1, -1
               do i = 1, 256
                   x(i, j) = (x(i, j) - a(i, j+1) * x(i, j+1)) / b(i, j)
               enddo
           enddo
c
c          ADI forward & backward sweeps along columns
c
           do j = 1, 256
               do i = 2, 256
                   x(i, j) = x(i, j) - x(i-1, j) * a(i, j) / b(i-1, j)
                   b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i-1, j)
               enddo
           enddo

           do j = 1, 256
               x(256, j) = x(256, j) / b(256, j)
           enddo
           do j = 1, 256
               do i = 255, 1, -1
                   x(i, j) = (x(i, j) - a(i+1, j) * x(i+1, j)) / b(i, j)
               enddo
           enddo
        enddo
        print *, x
    end
```

B

TRANSFORMED ALTERNATE DIRECTION IMPLICIT INTEGRATION KERNEL

In this Appendix the implementation of the single program multiple data code used in the experimental results Chapter is described. We have transformed the original program in order to control the exact placement of the data during the execution. The transformation consists on adding parallelization directives according to the ORIGIN 2000 programming model, and on performing some well known loop transformations. For the description of the transformations, we have selected phases 2, 4, and 5 of the ADI integration kernel, as they cover the main aspects of the implementation.

Phase 2 is a fully parallelizable nest of loops, which initialize arrays a, b, and x. The original sequential code contains two loops: the first one iterates from 2 to 255, and the second loop iterates along the whole array dimension.

```
do j = 2, 255
     do i = 1, 256
          a(i, j) = 1.0
          b(i, j) = 3.0
          x(i, j) = 5.0
     enddo
enddo
```

Assuming a one-dimensional column `BLOCK` distribution with 4 processors, each processor owns 64 columns of each array. However the iteration space generates iterations 2 to 255. In order to force the owner computes rule, we have to ensure that processor 0 executes from column 2 to 64, processor 1 executes from column 65 to 128, processor 2 executes from column 129 to 192, and processors 4 executes from column 193 to 255. The single program multiple data code generated strip-mines the loop in order to guarantee the appropriate iteration space partition driven by the owner computes rule. The generated program is the following:

```
C$DOACROSS LOCAL(lb$j, ub$j, i, j)
       do my$j = 0, 3
            lb$j = max(my$j * 64 + 1, 2)
            ub$j = min((my$j + 1) * 64, 255)
            do j = lb$j, ub$j
                 do i = 1, 256
                      a(i, j) = 0.0
                      b(i, j) = 1.0
                      x(i, j) = 5.0
                 enddo
            enddo
       enddo
```

where the C$DOACROSS statement directs the compiler to generate special code to run iterations of the DO loop in parallel. The C$DOACROSS directive applies only to the next statement which must be a DO loop. By default, the scheduling used partitions the iterations among the processes by dividing them into contiguous pieces and assigning one piece to each process. Therefore we guarantee that each processor executes its corresponding chunk of iterations. The LOCAL clause provokes each processor to have its own local copy of the variable. A variable can be local if its value does not depend on any other iteration and if its value is used only within a single iteration.

Instead of computing chunks of iterations from the array sizes, functions *max()* and *min()* have been inserted in the computation of the *j* loop bounds, for the code to be single program multiple data. Note that function *max()* will only get the value 2 for processor 0, and function *min()* will only be 255 for processor 4.

This hand coded transformation is equivalent to the recently available AFFINITY clause of the C$DOACROS directive. The affinity schedulling controls the mapping of iterations of a parallel loop for execution onto the underlying threads. However, we have preferred the hand coded version due to implementation efficiency.

If the distribution is two-dimensional, and assuming 16 processors in a 4×4 processors grid, rows and columns of the arrays are partitioned across 4 processors. In this case, both the i and the j loop have to be strip-mined.

```
C$DOACROSS NEST(my$j, my$i) LOCAL(lb$i, ub$i, lb$j, ub$j, i, j)
       do my$j = 0, 3
          do my$i = 0, 3
             lb$j = max(my$j * 64 + 1, 2)
             ub$j = min((my$j + 1) * 64, 255)
             do j = lb$j, ub$j
                lb$i = my$i * 64 + 1
                ub$i = (my$i + 1) * 64
                do i = lb$i, ub$i
                   a(i, j) = 0.0
                   b(i, j) = 1.0
                   x(i, j) = 5.0
                enddo
             enddo
          enddo
       enddo
```

The `C$DOACROSS NEST` directive specifies that the entire set of iterations across the (myj, myi) loops can be executed concurrently. The restriction is that both loops must be perfectly nested.

The next phase whose transformation is worth thinking about, is phase 4, in which there is a flow dependence in the j loop. The original sequential code is the following:

```
       do j = 2, 256
          do i = 1, 256
             x(i, j) = x(i, j) - x(i, j-1) * a(i, j) / b(i, j-1)
             b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j-1)
```

```
                enddo
           enddo
```

Sequential loops that use shared data are executed by a single processor in most parallel architectures. This means that if the data array dimension accessed in the sequential loop is initially distributed, it will be moved to the local memory of that processor. To avoid this, we have partitioned the iteration space in such a way that the processor who owns the corresponding chunk is the one that performs the computation. Synchronization has to be introduced to guarantee that the loop is executed preserving the sequential execution.

If the data mapping strategy suggests to distribute the second array dimension, and again assuming one-dimensional BLOCK distribution with 4 processors, the generated single program multiple data code is the following:

```
       token = 0
C$DOACROSS LOCAL(lb$j, ub$j, i, j)
       do my$j = 0, 3
1111        if ( token .ne.  my$j ) goto 1111
            lb$j = max(my$j * 64 + 1, 2)
            ub$j = (my$j + 1) * 64
            do j = lb$j, ub$j
                 do i = 1, 256
                      x(i, j) = x(i, j) - x(i, j-1) * a(i, j) / b(i, j-1)
                      b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j-1)
                 enddo
            enddo
            token = token + 1
       enddo
```

Note that the shared variable *token* is used as a spin-lock that ensures the sequential ordered execution of the inner loop nest. When processor p finishes the computation, the token is incremented and processor $p+1$ can start the computation of its chunk.

Similarly, in the two-dimensional data distribution case and assuming 16 processors in a 4×4 grid topology, the execution of the j loop has to be performed sequentially, but the i loop can be executed in parallel.

```
          do i = 1, 4
               token$i(i) = 0
          enddo
   C$DOACROSS NEST(my$j, my$i) LOCAL(lb$i, ub$i, lb$j, ub$j, i, j)
          do my$j = 0, 3
               do my$i = 0, 3
   1111             if ( token$i(my$i+1) .ne.  my$j ) goto 1111
                    lb$j = max(my$j * 64 + 1, 2)
                    ub$j = (my$j + 1) * 64
                    do j = lb$j, ub$j
                         lb$i = my$i * 64 + 1
                         ub$i = (my$i + 1) * 64
                         do i = lb$i, ub$i
                              x(i, j) = x(i, j) - x(i, j-1) * a(i, j) / b(i, j-1)
                              b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j-1)
                         enddo
                    enddo
                    token$i(my$i+1) = token$i(my$i+1) + 1
               enddo
          enddo
```

In this case, vector *token*$i() of shared variables is required in order to synchronize the sequential execution of columns within a row. One element of this token is provided for each set of processors in the first array dimension.

Finally, phase 5 is a single loop that only traverses the last column of arrays x and b.

```
          do i = 1, 256
               x(i, 256) = x(i, 256) / b(i, 256)
          enddo
```

Again, we have to force the processor or processors that owns this column to be the one that computes this loop. Assuming the `BLOCK` one-dimensional column distribution with 4 processors, a conditional statement has to be inserted to guarantee that processor number 3 is the only one that will execute all the iterations of the loop.

```
C$DOACROSS LOCAL(i)
       do my$i = 0, 3
            if ( my$i .eq.  3 ) then
                 do i = 1, 256
                      x(i, 256) = x(i, 256) / b(i, 256)
                 enddo
            endif
       enddo
```

Similarly, in the two-dimensional data distribution with 16 processors, the same condition holds but for each processor owning elements in the last column. The inner i loop can be parallelized, as can be seen in the following fragment of code:

```
C$DOACROSS NEST(my$j, my$i) LOCAL (lb$i, ub$i, i)
       do my$j = 0, 3
            do my$i = 0, 3
                 if ( my$j .eq.  3 ) then
                      lb$i = my$i * 64 + 1
                      ub$i = (my$i + 1) * 64
                      do i = lb$i, ub$i
                           x(i, 256) = x(i, 256) / b(i, 256)
                      enddo
                 endif
            enddo
       enddo
```

Note that the loop body has not been modified in any case. We have only transformed the iteration space in order force that each processor computes the corresponding iterations, and that according to the owner computes rule, the arrays remain distributed as desired.