

Temporal Data Placement

by Muhammad Umair Javed Ilam Sindhu

© Muhammad Umair Javed Ilam Sindhu

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Engineering (Computer Engineering)

Faculty of Engineering and Applied Science

May 31, 2022

St. John's

Newfoundland

Abstract

How can we provide data where it is required and when it is required to the execution units of parallel hardware? Program transformations have been a focus to improve the performance of parallel computing, whereas data optimizations like data placement, data layout transformation, data migration and data replications are overlooked especially in compiler domain. We are proposing a methodology, Temporal Data Placement, that will schedule and place data in both time and space. The use of our methodology will enhance the performance of parallel systems significantly, and it can also be automated.

Acknowledgements

I would like to thank Dr. Theodore S. Norvell, and Dr. Adrian Fiech from the bottom of my heart for their help, guidance, and support.

Contents

Abstract	i
Acknowledgements	ii
List of Figures	v
List of Abbreviations	vii
List of Notations	viii
0 Introduction	1
1 Overview of Related Work	8
1.0 Compile-time Data Placement Approaches	8
1.0.0 Cache Optimization	9
1.0.1 NUMA Optimization	15
1.1 Run-time NUMA Optimization Approaches	17
1.2 Discussion	18
2 Background	20
2.0 Memory Management	20
2.1 Polyhedral Model	24
2.1.0 Linear Program and Linear Programming	25
2.2 Polyhedral Framework and Optimization	28
2.2.0 Representation of programs in Polyhedra	29

2.2.0.0	Program Analysis Phase	29
2.2.0.1	Transformation Phase	29
2.2.0.2	Code Generation Phase	30
2.2.0.3	Affine Function	30
2.2.0.4	Parameter	30
2.2.0.5	Static Control Parts (SCoP)	30
2.2.1	Iteration Domain	32
2.2.1.0	Vector Space	34
2.2.1.1	Convex Set	35
2.2.1.2	Convex Combination	35
2.2.1.3	Convex Hull	35
2.2.1.4	Hyperplane:	35
2.2.2	Dependence In Polyhedra	39
2.3	Summary	44
3	Minimize the Maximum Distance	45
3.0	Temporal Data Placement (TDP)	46
3.0.0	The Problem Formalization	49
3.0.0.0	Affine form of Farkas Lemma	50
3.0.1	Algorithm	51
3.0.2	Polyhedral Representation	51
3.0.3	Implementation of Farkas Lemma	53
3.0.4	Optimization Tool and Solver Selection	62
3.0.5	Results	62
3.0.6	Example 2	63

3.0.6.0	Representation	65
3.0.6.1	Implementation and Results	66
3.0.7	Compound Polyhedra Example	67
3.0.7.0	Polyhedral Representation	68
3.0.7.1	Problem Constraints	71
3.0.7.2	Farkas Lemma Implementation	74
4	Minimize the Maximum Cost	78
4.0	Temporal Data Placement (TDP)	78
4.0.0	Minimize the Maximum Cost Procedure	81
4.0.1	Problem Formalization	83
4.0.2	Optimization Problem Formalization	83
4.1	Optimization Methods	84
4.2	Approximate/Heuristic Methods	85
4.2.0	Simulated Annealing (SA)	85
4.2.0.0	Iterative improvement	86
4.2.0.1	Local random search	87
4.2.0.2	Exploration	87
4.2.0.3	Exploitation	88
4.2.0.4	Algorithm	88
4.2.1	SA Implementation	89
4.2.1.0	Compound Polyhedra Example	95
5	Conclusion and Future Work	100
	References	104

List of Figures

0.0	UMA and NUMA Architecture from [motioncontroltips.com by Miles Budimir]	3
0.1	Distributed Memory Architecture from [Eskicioglu et al, 1996 (Technical Report), UOA]	4
0.2	NUMA Architecture of Four Nodes from [Guad 2015]	5
0.3	A NUMA System of Two Nodes from [Borger 2014]	5
1.0	One Dimensional Jacobi Code	10
1.1	Inter Bank Communication	11
1.2	Array Dimension and Data Distribution according to layout order.	13
2.0	Levels of Basic Memory Hierarchy	21
2.1	Graphical Representation of Inequalities	34
2.2	A Rational and Integral Polyhedron	37
2.3	Dependance	40
3.0	NUMA Machine 1	47
3.1	NUMA Machine 2	64
3.2	Optimized Data Placement	67
3.3	Rotation Problem	69

4.0	Data Placement for Time 0	90
4.1	Data Placement for Time 1	91
4.2	Presumed Data Placement	94
4.3	Optimal Data Placement	95

List of Abbreviations

MINLP	Mixed Integer Non-Linear Programming
NUCA	Non-Uniform Cache Access
NUMA	Non-Uniform Memory Access
TDP	Temporal Data Placement

List of Notations

\mathcal{T}	Set of Times
\mathcal{I}	Set of Statement Instances
\mathcal{P}	Set of Processors
\mathcal{L}	Set of Program Locations
\mathcal{M}	Set of Memories
Θ	Schedule Relation
Φ	Fetch Relation
Σ	Store Relation
Π	Computation Placement Relation
Δ	Data Placement Relation
δ_{fetch}	Fetch Access Time
δ_{store}	Store Access Time
δ_{comp}	Computation Time

Chapter 0

Introduction

One of the major requirement for any computing system is that the minimum time should be taken to execute a program. In this era, there is no question about the speed of the execution units relative to the memory systems, as they are much faster than the memory units. That means the computing systems performance is highly dependent on the availability of the data to the processing units at the time they are required. Of course, if a computing unit gets data by the time when it is needed or ahead of time, then the system performance will be highly optimized.

Initially a source program resides in the secondary memory and its addresses are known as symbolic addresses or program locations. To execute a program, first, the data is brought into the main memory, from where it is sent to the processor for execution, and the output of the computation is sent back to the memory. The binding of the program locations to the memory places can be made at three different times: compile-time, load-time and run-time. if we know which program location is

going to be placed in which memory address at compile-time, then the final address binding can be done at this time. And it will happen before loading the program into the memory by the OS. However, if it is not known at compile time where the program locations will reside in the main memory then the final address binding is delayed until the load time. In this case, the operating system manager (loader) will assign relocate-able addresses to the program locations, and it will be done after loading the program into the main memory. However, if a program keeps changing its places even after loading into the memory, then the final binding is further delayed, and it will be done by the processor at the time of execution.

From the preceding discussion, we can understand that compile-time address binding can reasonably minimize the execution time of a program. Therefore, to optimize a system performance, the first thing to do is to associate program locations to memory addresses at compile-time. However, in the presence of modern parallel hardware, it is a not an easy task. Parallel systems contain multiple processors and multiple memories in a single machine or made up with a network of dozens of processing units.

Parallel computers can be classified in two categories: shared memory multiprocessor, and distributed memory multiprocessor[Ste, 2018][Baillie, 1988]. Shared memory multiprocessor systems are further divided in two architectures: Uniform Memory Access (UMA) and Nonuniform Memory Access (NUMA) as shown in the Figure 0.0.

All processors/cores of UMA and NUMA machines shared one global memory and have direct (hardware) access to all blocks of this global memory space. However, every processor of a UMA machine can access all memory locations in the equal

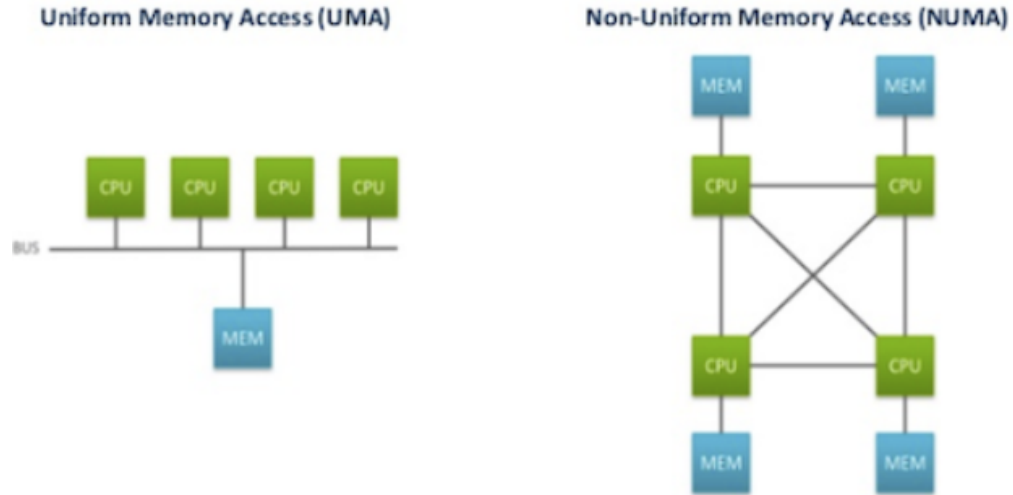


Figure 0.0: UMA and NUMA Architecture from [motioncontroltips.com by Miles Budimir]

amount of time, on the other hand, NUMA systems memory access time length is not same for all processors since it depends on which memory module the processor is trying to access. Furthermore, UMA access is slower than NUMA for large systems and NUMA system has more bandwidth than UMA as well. Whereas in distributed memory multiprocessor system each processor can access to its local memory only as shown in the Figure 0.1[Eskicioglu and Marsland, 1996].

The message passing mechanism is used to migrate data among all memory nodes through an interconnection network. An example of distributed memory architecture is computer cluster in which different nodes relate to each other through fast local area network or the internet.

Most of the high performance shared memory multiprocessor systems use NUMA architecture instead of UMA, as NUMA machines are good computing platform for

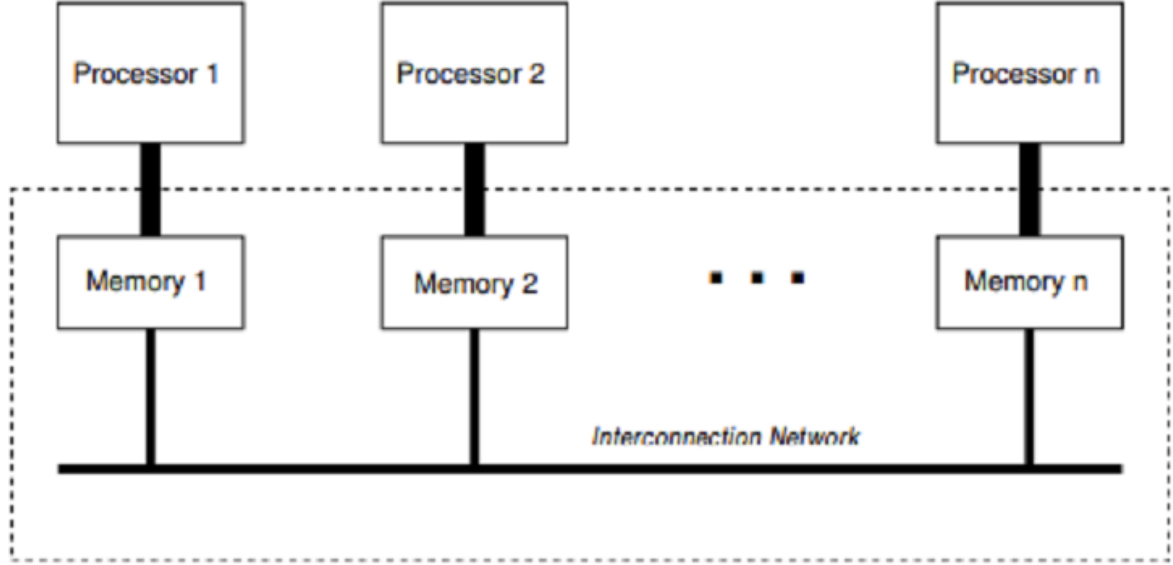


Figure 0.1: Distributed Memory Architecture from [Eskicioglu et al, 1996 (Technical Report), UOA]

solving huge scientific problems. A building block of the NUMA architecture is a node, which comprises a processing unit and a portion of address space of physical memory. Figure 0.2 shows a multicore multiprocessor system of four nodes based on NUMA architecture. Each core of a node is directly connected with the local memory and can access the other parts of the shared memory through the cross-chip connection.

With parallel hardware all/subset of processors can work on the parts of program simultaneously to compute it in a minimum possible time. However, it all depends on whether the needed data is available to a processor at the time of computation and if it is then in which memory this data is located. For example, Figure 0.3 shows a NUMA system with two nodes. It can be seen in the figure, CPU0 has two different

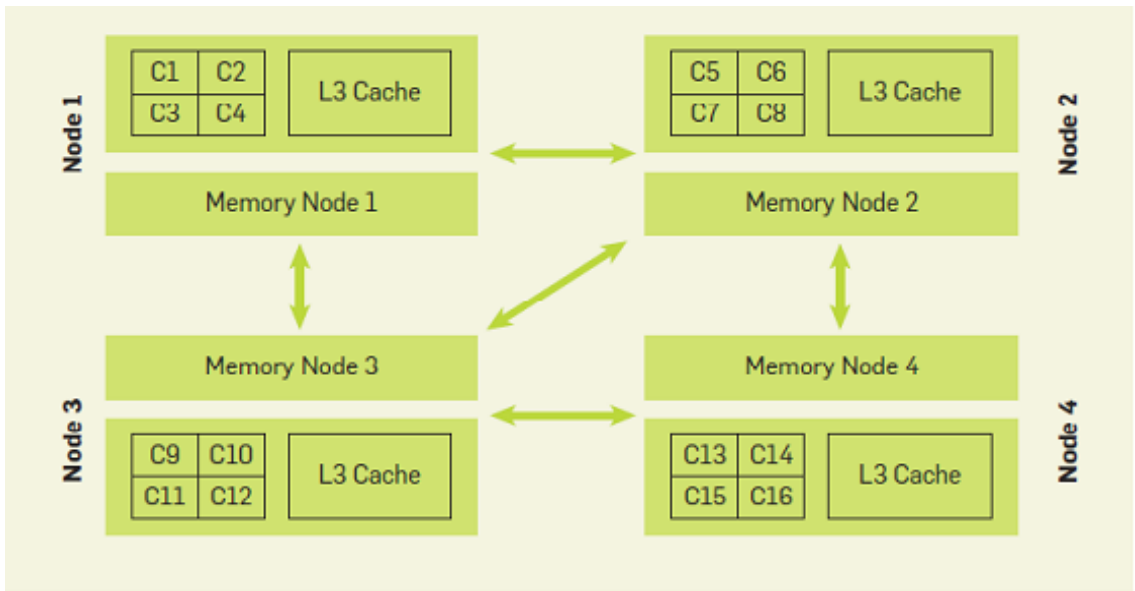


Figure 0.2: NUMA Architecture of Four Nodes from [Guad 2015]

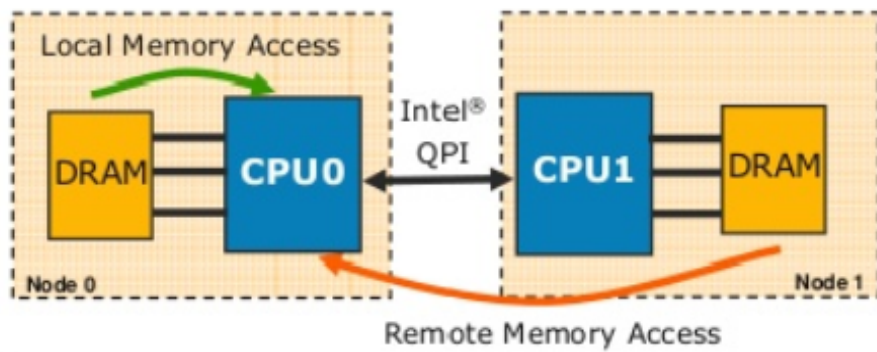


Figure 0.3: A NUMA System of Two Nodes from [Berger 2014]

memory access times: one for local memory access (access from Node 0), and second for remote memory access (access from Node 1). If CPU0 gets data from its local memory at the time of need, then it will complete execution in a minimum possible time. Otherwise, the data has more distance to travel to reach to the CPU0 which will ultimately increase the computation time. This example illustrates that the data placement at different nodes in the address space gives different execution times. And the signal path length from a processing unit to the memory is the main consideration parameter for system performance. This scenario tells us that an ineffective data placement among the memory nodes of NUMA and distributed memory computers can not help much to optimize the computation time of a program. Therefore, we need to place data in the corresponding or nearby memory node according to the needs of the processors.

We are proposing a compile-time data placement methodology for NUMA (Non-Uniform Memory Access) architectures and distributed memory machines. These machines can work parallelly on the different parts of the program, so data might need to be replicated in subset of memories or migrate from one memory to another during execution. This scenario raises an important problem:

How can we schedule, place, replicate and migrate the data to be where it is required and when it is required?

To know when and where the data is required, a system should be aware of the memory access pattern of the program. The polyhedral framework[Bastoul, 2004a][Bastoul, 2004b][Bondhugula *et al.*, 2008][Benabderrahmane *et al.*, 2010][Feautrier, 1992][Feautrier, 1988][Feautrier, 2013] can help us in this regard, it can provide the whole memory access pattern of the statements of the program at compile time in

the form of matrices, one matrix for each statement. Once we get the memory access pattern of the statements of a program, we can schedule and place data such that each processor can get needed data in local or nearby memory at the time of execution with the help of a polyhedral framework.

Our proposed compile-time approach Temporal Data Placement (TDP) is based on polyhedral framework. TDP is able to schedule, place, replicate and migrate data among the storage nodes of a parallel hardware according to the need and time of the computations. With TDP, we can provide needed data to the processors in local or near by memories at the time of execution. TDP is an effective technique to minimize the maximum distance between processors and memories, and we have successfully solved this problem by using a Mixed Integer Nonlinear Programming (MINLP) solver. Furthermore, TDP is a proficient method to minimize the maximum running time of a program. We have also successfully solved the problem of minimizing the maximum total running time of computation with the help of Simulated Annealing.

Chapter 1

Overview of Related Work

This chapter contains an overview of the related work about compile-time and run-time data placement techniques.

1.0 Compile-time Data Placement Approaches

The relevant research of the compiler optimization related to data layout transformations, data placement on Non-uniform Cache Access (NUCA) banks and memory allocation on NUMA nodes is discussed in this section. Lu et al [Lu *et al.*, 2009] and Zhang et al [Zhang *et al.*, 2011] targeted to last-level shared cache data placement optimization. Susungi et al [Susungi *et al.*, 2017] focused on memory allocation on NUMA nodes.

1.0.0 Cache Optimization

Lu et al [Lu *et al.*, 2009] have developed a compile-time framework for data locality optimization implemented on Chip MultiProcessors (CMPs). CMPs are implemented with a architecture, in which each tile has a processor core with a private L1 cache and one bank of shared L2 cache initially. But, now in the latest CMP architectures, each tile contains two levels of cache L1 and L2, as private for each core and a portion of the shared L3 cache. The Intel Core i7 and i9 processors are good examples of this architecture.

The shared last-level cache (L2/L3) is basically a Non-Uniform Cache Architecture (NUCA), which implements a bank-interleaved distribution of address space among all processor cores. Although, NUCA architecture design reduces the number of off-chip accesses, it creates some new problems such as wire delay in accessing data from remote bank. Lu et al addressed the issues raised by NUCA. They have developed a compile-time data locality optimization framework for bank-interleaved shared cache systems. Physical memory address space is mapped to the banks of a shared cache. Each core of a CMP has direct access to a bank of shared cache. The data accessed from the directly linked bank is considered as local access and it takes less time. Also, each core can access data from other banks but it would take more time, which degrades the performance of a system.

With the help of an example Figure 1.0 of one dimensional Jacobi iteration, Lu et al have shown that the performance can be enhanced significantly, by placing a non-canonical (non-conventional) data layout rather than a canonical layout of address space in the banks of shared cache. Lu et al took an example of a processing unit

```

while (condition) {
    for (i = 1; i < N-1; i++)
        B[i] = A[i-1] + A[i] + A[i+1]; //stmt S1
    for (i = 1; i < N-1; i++)
        A[i] = B[i]; //stmt S2 }

```

(a) Sequential 1D Jacobi code.

```

Compute(int threadID) {
    processor_bind(threadID);
    NB = ceiling(N / (P * L)) * L;
    while (condition) {
        for (i = threadID*NB; i < min(N, (threadID+1)*NB); i++) {
            B[index(i)] = A[index(i-1)] + A[index(i)]
                + A[index(i+1)];}
        Barrier();
        for (i = threadID*NB; i < min(N, (threadID+1)*NB); i++) {
            A[index(i)] = B[index(i)]; }
        Barrier();}
}

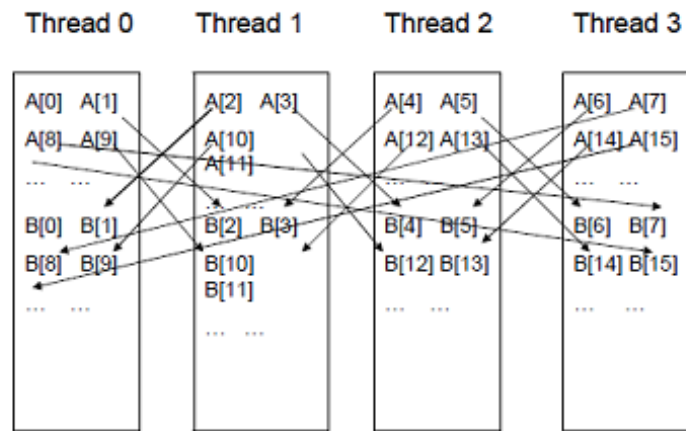
```

(b) Parallel 1D Jacobi code with non-canonical data layouts.

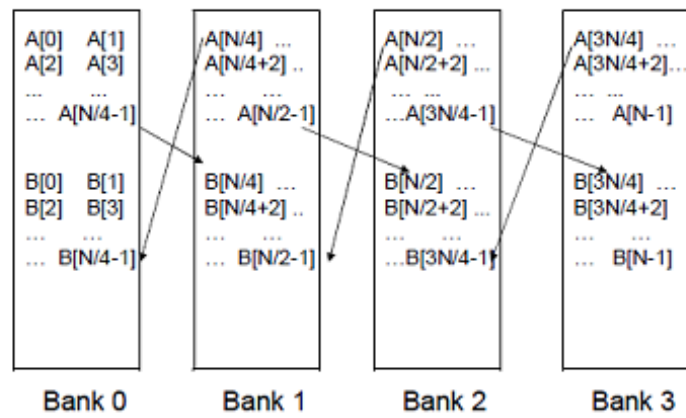
Figure 1.0: One Dimensional Jacobi Code

comprised of 4 cores and the L2 cache shared among all cores. The L2 cache contained 4 banks; each core is directly connected to one bank. In this example, if the original data layout scheme (the bank-interleaved mapping) is followed, it would result in a high inter-bank communication cost. Whereas, if the data space is evenly divided among multiple banks fewer accesses would be needed. The sequential and the parallel 1D Jacobi code are shown in the Figure 1.0.

Figure 1.1 enlightens the benefit of non-canonical data placement. Each core executes one thread in parallel with others. Let us suppose the cache line size is 2 words. In the Figure 1.1(a) data is mapped on banks according to the original data



(a)



(b)

Figure 1.1: Inter Bank Communication

layout. Each bank is allocated two elements of arrays A & B in a round robin fashion. It can be observed that there are many remote memory accesses during the execution of this code, and the system will show poor performance. But if the elements of the arrays are evenly divided into four parts and mapped to the banks, there would be only six remote bank access. So, by only changing the original data layout, a system performance can be boosted meaningfully.

Lu et al have applied an integrated methodology; the data layout transformation and the loop transformations. Their approach is based on two steps: localization and data layout transformations. The idea is that whenever a computation needs some data for processing, it should be placed in close by memory locations in some specific dimension of the array, so that no iteration of a computation accessed data outside of that specific dimension. And, to achieve this goal, data layout transformations would be required, because original data layout may not fulfill the requirement. Once the data transformations are done accordingly, then it would be easy to map some selected iterations to some specific core and to allocate needed data on some specific bank. They have introduced constraints on the computation allocation based on data requirement. The localization analysis determines those dimensions of an array that can be layout transformed. And a non-canonical layout can be gained by grouping a set of data layout transformations like strip-mining, permutation and padding. Figure 1.2 illustrates the idea of data layout transformation with respect to arrays.

We are considering the same memory architecture given in Figure 1.0 corresponding to the 1D Jacobi example. We suppose that array A contains 20 elements, and is a two-dimensional array; dimension 0 (vertical) has 4 elements and dimension 1 (horizontal) has 5 elements, as shown in Figure 1.2(a). The data of array A is mapped

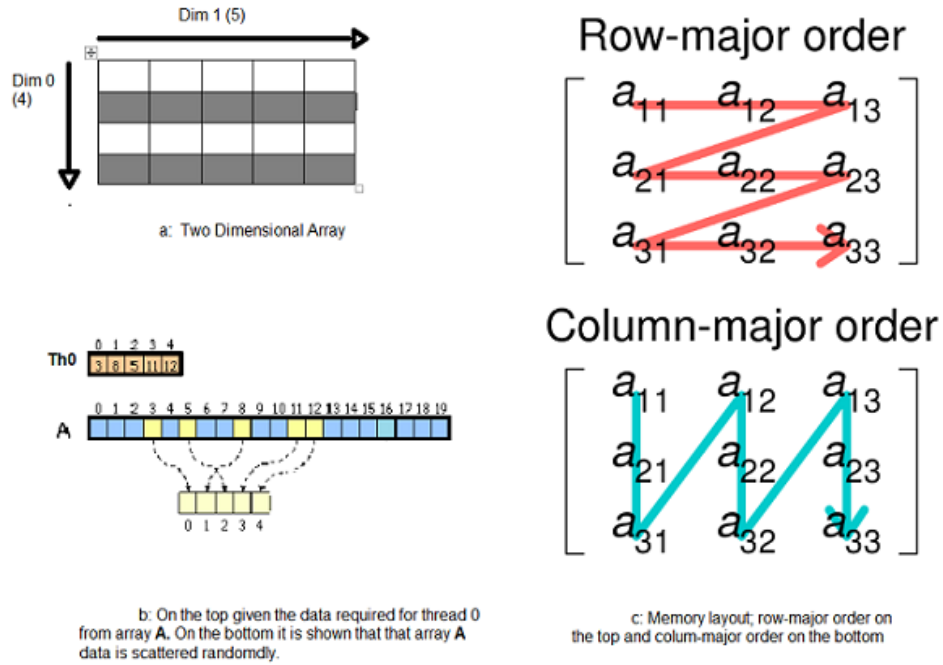


Figure 1.2: Array Dimension and Data Distribution according to layout order.

in physical memory in only one dimension as shown in Figure 1.2(b). Array data is placed according to row-major order in C language, and in column-major-order in FORTRAN, as shown in Figure 1.2(c). Figure 1.2(b) top portion gives the data elements required for thread 0 for execution. If we observe Figure 1.2(b), array A physical data mapping, the data required for thread 0 is scattered and if we follow the memory interleaved placement on banks of cache, we would find the required data for the thread in three banks and obviously thread 0 will make remote accesses. But, if we placed the data in a way that every thread gets data from its own bank, we can optimize the performance. In this example, there are 20 elements of array A. Let us suppose each thread needs 5 elements, we can map five such elements to each bank in a way that each thread gets data from its own bank.

Zhang et al [Zhang *et al.*, 2011] developed an automatic data layout transfor-

mation approach. They addressed multithreaded applications executed on multicore systems. Modern multi-cores have shared on-chip caches, which brings additional complications while running multi-threaded applications. For optimization purposes, there should be minimum data sharing among the threads during parallel execution of a multi-threaded application. The goal is that each thread mostly accesses its local data. But, in conventional linear data layouts (such as row/column/diagonal major), data accessed by a single thread is usually scattered in memory space, although that data exists geometrically close in the data space of a multi-dimensional array. It is shown in Figure 1.2(b). Due to such placement of data, the conflicts occurred in shared on-chip caches when multiple threads accessing data at the same time. Zhang et al have addressed this issue and proposed a data layout transformation technique. To change the given data layout, their approach considers following two factors

- I. Data access pattern of each thread.
- II. The on-chip cache hierarchy of the target architecture.

Before unveiling their strategy, first, we need to go through localized array characteristics.

- I. Each thread mostly accesses those elements of the array which are placed on the same node, the node on which that thread is running. And, there are a few references made by other threads to this part of the array.

- II. If the data dedicated to a thread is further divided into sub-sections, those sub-sections will be considered as local to different computation sets (a group of iterations) at a different time. For example, a thread is supposed to execute 20 iterations of the inner loop, and it has two computation sets, ten iterations for each set. The data portion required during execution would be divided into two parts, the

first part contains the data which is required for first ten iterations and let us suppose execution of these iterations get the start at time 0, and the second part accessed by next 10 iterations, whose execution get the start at time 2.

These two characteristics imply that

- I. The reuse distance in a localized array is small
- II. The data are localized in each thread (space-wise) as well as in each computation set to be executed by a thread (time-wise).

Their approach finds localized arrays in parallelly executing loop nests, make tiles of each array in such a way that those tiles are mostly accessed by one thread. Then map these tiles to memory space. Their technique is based on 4 steps.

1. Localization Test: Identify whether an array is localized or not. If so, then proceed the following steps;
 2. Tiling Determination: Find an appropriate tiling
 3. Transformation Determination: Find and apply a transformation function, which changes an original layout to a new memory layout.
 4. Memory Mapping: Place that transformed layout to the memory space.

1.0.1 NUMA Optimization

Susungi et al [Susungi *et al.*, 2017] presented an array-centric parallel intermediate language IVIE, which complements affine transformations. The affine transformations proved successful for data reuse, to exploit parallelism and in optimizing memory accesses. Susungi et al want to enlightened the benefits of integrating NUMA-awareness in the compilation flow for performance enhancement. They also want to presented

that the data layout transformations at compile time can improve a system performance too. There are separate approaches available in IVIE, for data layout transformations and for memory allocation on NUMA nodes. The NUMA architecture is used to build large-scale systems in which physical memory is placed on the different nodes and these nodes are connected over cache-coherent high-performance links. A processor core can get data from its own node or from remote memory node, which resulted in an increase in access time (latency). So, a random or unmanaged data placement on NUMA nodes may cause memory contention when the processes or threads running on different processing nodes trying to access the data from the one identical memory node. Hence, “ NUMA-unaware programs running on NUMA platforms tend to not benefit from the additional computational power and memory bandwidth offered by the multiple nodes of the system. NUMA-awareness is commonly introduced at runtime using tools, but introducing NUMA-awareness to the application or better, to the compiled code automatically, provides much greater performance” to the system.

Besides loop transformations, a good data layout and proper placement of data in the main memory can play an important role in improving data locality. There are dedicated constructs provided in IVIE for manipulating data layout, memory allocation on NUMA nodes and iteration spaces. The IVIE is a meta-language, and for demonstration purposes, the authors have combined it with Pluto [[Bondhugula et al., 2008](#)] (a Fully Automatic Polyhedral Program Optimization System) to apply NUMA-awareness and to implement additional layout transformations.

They have provided an abstraction for the access functions. The functions execute element-wise operations. The statements take functions as arguments and return array

elements only. The arrays are given preference as first-class data structures in IVIE, arrays have dimensions and a layout, which are set at declaration time. The physical array layout in IVIE is same as the row-major layout of C arrays. The arrays can be abstracted as virtual or physical memory. The physical memory distributed over diverse NUMA nodes. The source code arrays are mapped to the physical memory after processing by Pluto. For data layout transformations, only transposition is available in IVIE. The storage ordering of the elements of the multi-dimensional array may follow a permutation of its dimension.

For the data placement on NUMA nodes, dedicated constructs are provided and the data placement can only be applied to physical arrays. There are two types of allocation policies in IVIE; Interleaved Allocation: The arrays are divided into blocks/tiles and then mapped across a set of NUMA nodes in a round-robin fashion. The minimum granularity is 4096 bytes. Any size multiple of 4096 bytes can be used. Allocation on Specific Nodes: an array is placed on one specific node. The data can also be replicated on multiple nodes by the provided construct.

1.1 Run-time NUMA Optimization Approaches

The problem of memory management on NUMA nodes has been addressed by [Majo and Gross, 2015], [Gaud *et al.*, 2015], [Dashti *et al.*, 2013], [Majo and Gross, 2011], and [Blagodurov *et al.*, 2010]. And for distributed memory management by [Zima and Chapman, 1993]. All of them proposed run-time memory optimization approaches, and targetted the issues of effective placement of data on NUMA nodes, remote memory access penalty, and the contention in shared memory resources. None of

them considered the time factor while placing or migrating data among the memory nodes.

1.2 Discussion

Most of the compile-time polyhedral based research is about non-uniform cache architecture (NUCA) optimization and targeting data layout transformations in the shared cache. The outcome of NUCA optimization is to minimize the distance between processors and banks of the shared cache. NUCA memory access optimization could be a good solution for small problems due to the small size of data but is not for large-scale engineering and scientific problems because there could be many caches misses. For large-scale problems, a processor would have to need to access RAM more often, and what if the placement of data in RAM is not well managed. Furthermore, embedded systems, FPGAs and GPUs do not have cache in their architecture.

Neither compile-time nor run-time approaches consider the time factor while placing or migrating data among the memory nodes of parallel hardware. Moreover, not enough research has been done for an effective data placement for NUMA systems in the compiler-domain, especially in the polyhedral framework. Susungi et al [Susungi *et al.*, 2017] proposed to place block size data (minimum of a page (4096 bytes) size) among the NUMA nodes with a round-robin fashion. Either in compile-time or run-time, no one has considered the time factor while placing data in NUMA memory nodes. However, we have introduced a novel idea to place data into memories according to the time-dependent requirement of the processors. The TDP is an effective technique to schedule and place data according to the need of the processors and when

they need it. It can minimize the maximum distance between memories and processors and minimize the maximum running time of a program as well.

Chapter 2

Background

2.0 Memory Management

To enhance the performance of any computing system such as desktop computers, servers, clusters and embedded computers, a critical requirement is to supply the required data swiftly to the processing unit. However, memory units suffer from a significant delay in supplying data to the processors, as the processor's computing speed is increasing very fast compared to the memory. This dilemma creates a significant delay between the processor and main memory. It also raises the question about the ability of the memory system, whether it can provide the data to the processor at the required speed or not. To cure this problem, memory hierarchies were introduced which expedite the delivery of data to the processing units. Figure 2.0 shows a basic memory hierarchy organization of a computing system.

A general hypothesis about the memory accesses pattern is that a program spends 90% of its execution time in only 10% of the code, e.g., the execution of loop kernels/

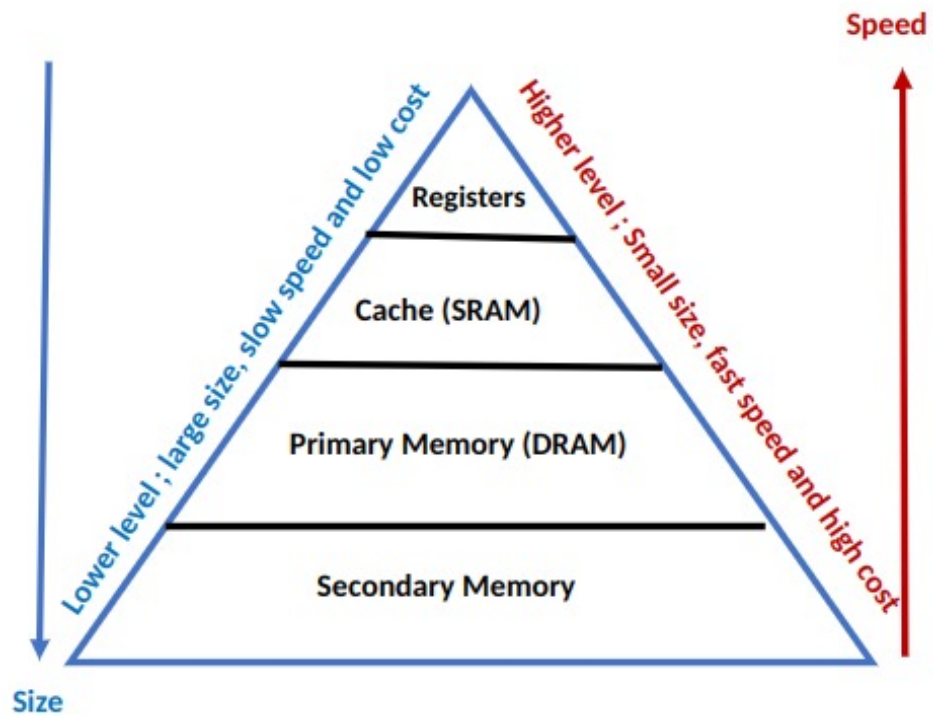


Figure 2.0: Levels of Basic Memory Hierarchy

loop nests part of the code. During a given period, most programs access sooner to a small portion of their address space. This phenomenon is known as locality of reference or principal of locality and it has two categories, temporal locality, and spatial locality. [Bastoul, 2004a] defines locality principle as follows:

Principal 1: Temporal Locality: Once a given data is accessed, it will tend to be accessed again soon.

Principal 2: Spatial Locality: Once a given data is accessed, nearby data in the address space will tend to be accessed soon.

To achieve the best performance of the system, the value or data that is going to be used in next clock cycle by the processor should be placed at the highest level of the hierarchy by the memory system. The topmost level of the hierarchy is the register, which can hold exactly one word. When two or more consecutive memory operations access the same data element, that data element should be held in the register. So, by register reuse the processor doesn't need to fetch that value from the lower level, and in this way, a processor can take the advantage of temporal locality. Register reuse optimizations are based on scalar replacement and loop transformations such as unroll-and-jam, loop interchange, and loop fusion. In case of the cache, a cache line is the unit of transfer and holds multiple words. When two or more successive memory operations access the different data elements, those data elements should be placed nearby, e.g., in the same cache line, and this is the spatial reuse of the data. Better utilization of the cache can optimize the performance of any system. Modern computing systems generally have three levels of cache, levels 1 and level 2 of the cache are private to the core of a processor in a multi-core multiprocessor system, and the level L3 cache is shared among all the cores of a processor. Loop

transformations play a significant role in exploiting the spatial reuse of data in the different levels of cache. Level 3 cache, also known as last-level cache is shared among all processing cores of a NUMA node as shown in the Figure 0.2, and each core is directly connected to a bank of L3 cache and has a lower access time to that directly linked bank. Whereas, other cores take more time to access that particular bank of cache.

A lot of work has already been done on different levels of caches for memory access time optimization purposes and still continues. However, issues of data placement on the memories of present-day parallel hardware like NUMA and distributed memory systems did not get much attention from academia and industry. Cache optimization can address small problems, because the complete data of a small problem could fit into the cache. In the case of bigger problems, a cache cannot hold complete data at once, therefore, a lot of caches misses would occur which will badly degrade the performance of the system. Moreover, computing systems exist which do not contain cache like embedded systems generally and FPGA particularly. In the light of above discussion and after having a second look at Figure 2.0, we can deduce that we are missing a very important level of memory hierarchy for consideration, and that is the Main Memory.

The commonly used performance measures of a memory system are latency and the bandwidth, which are defined by [Allen and Kennedy, 2001] as follows:

Latency is the number of processor cycles required to deliver any single data item from memory.

Bandwidth is the number of data elements that can be delivered to the processor from the memory system on each cycle.

System performance can be enhanced by putting data on the same processing node which is going to use that data for execution. Hence, a good placement of data on the NUMA and distributed computer memory nodes can reduce the latency, increase the bandwidth and subsequently enhanced the performance of a system. Plenty of attention has been given to loop-level transformations and mostly on level 1 and level 2 cache to improve memory access time. And good research is carried out for optimizing level 3 cache. Whereas, the data layout transformations and the proper placement of data on NUMA and distributed computer memory nodes has not had as much attention. Although both of these machines can play a vital role in optimizing the performance of any modern computing system. In the case of NUMA placement, an operating system is typically targeting to achieve this task and less research has been done to involve the compiler in this work. Following are the major challenges related to NUMA memory management:

- Effective placement of data on NUMA nodes.
- The contention in shared memory resources, e.g., memory controller, cross-chip interconnect or prefetchure.
- Remote memory access penalty.

2.1 Polyhedral Model

The Polyhedral Model is a mathematical model. The use of this model for computer program optimization is based on Parametric Integer Programming (PIP) [Fautrier, 1988]. In order to understand the nature of PIP, we first have to explore Linear Programming (LP) and Integer Programing. Both LP and IP are based on linear

equations, linear functions, and linear inequalities. A linear equation is an algebraic equation of degree 1, e.g., $5x + 7 = 0$ is a linear equation of one variable. It may have any number of variables, for example the equation $2x + 4y + 6w + 9 = 0$ has three variables. The basic idea is that, each term of a linear equation can be either a constant or the product of a constant and a single variable, e.g., $2x$ and 9 . "The term programming means planning activities that consume resources and/or meet requirements, with the consideration of the constraints." [Chen *et al.*, 2011]

2.1.0 Linear Program and Linear Programing

"A linear program (LP) is a mathematical model that expresses the physical, behavioristic, or economic relationship between the various elements of a decision problem in a standardized mathematical form; and linear programming is a standardized method of determining the optimal decision, action, or policy for the problem investigated. Main characteristics of linear programing are as follows:

1. There exists an objective that is to be optimized, such as maximization of profits or minimization of costs.
2. There exist alternative courses of action or decision variables, as we call them, to achieve the desired objective.
3. There exist restrictions on the amount or extent of attaining the objective, that is expressed in the form of constraints on the values of the decision variables".

[Daellenbach and Hans, 1970]

"A general linear programming problem is that of maximizing (or it could be

minimizing) a linear function

$$z = c_1x_1 + c_2x_2 \dots + c_nx_n$$

of n real variables x_1, x_2, \dots, x_n satisfying the non-negativity conditions

$$x_1 \geq 0, x_2 \geq 0, \dots, x_n \geq 0$$

and m linear constraints

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq = \geq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq = \geq b_2$$

.....

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq = \geq b_m$$

The constraints can be a mixture of the $\leq, =, \geq$ variety. The aim will be to maximize the objective or perhaps to minimize the objective. The values of the b_i, c_i, a_{ij} are assumed to be known constants, and x_i is a set of variables” [Bunday, 1984].

We can represent it in matrix form as well

$$\text{Maximize:} \quad z = c_j^T x_j$$

$$\text{Subject to:} \quad A_{ij}x_j \leq b_i$$

$$x_j \geq 0 \quad (j = 1, 2, \dots, n)$$

The linear programming solutions can provide both type of results: integral and fractional. For computer programs mostly we are interested in integral outputs, which are provided by pure integer programming. The form of the pure integer program is as follows:

$$\begin{aligned}
\text{Maximize:} \quad & z = \sum_{j=1}^n c_j x_j \\
\text{Subject to:} \quad & \sum_{j=1}^n a_{ij} x_j \leq b_i \quad (i = 1, 2, \dots, m) \\
& \quad \quad \quad (j = 1, 2, \dots, n) \\
& \quad \quad \quad x_j \geq 0 \quad \text{integral}
\end{aligned}$$

We can represent a pure integer program as follows

$$\begin{aligned}
\text{Maximize} \quad & z = cx \\
\text{Subject to} \quad & Ax \leq b \\
& \quad \quad \quad x \geq 0 \quad \text{integral}
\end{aligned}$$

where A is a $m \times n$ matrix, $c = (c_1, \dots, c_n)$ is a row vector, and $b = (b_1, \dots, b_m)^T$ is a column vector. All of these usually contain rational data, while an integral column vector $x = (x_1, \dots, x_n)^T$ consists of those variables which are to be optimized.

Linear programming provides continuous optimum and the simplex method is an effective procedure to solve all linear problems, whereas there are a number of methods have been developed to provide integer optimal solutions. The Gomorian method is the most prominent technique to solve the integer programming problems. This method is also used in the Parametric Integer Programming Project, developed by Feautrier [[Feautrier, 1988](#)]. The idea of PIP can also be used for optimization of Mixed Integer Nonlinear Programming (MINLP) problems. A MINLP programming problem can have nonlinear functions in the objective function and/or the constraints along with real and integer unknown variables.

2.2 Polyhedral Framework and Optimization

The Polyhedral Framework is a semantical algebraic representation of the code [Feautrier, 1988][Feautrier, 1992][Bastoul, 2004b][Bastoul, 2004a][Bastoul, 2012]. It can be used to build and search for complex sequences of optimizations. The iteration domain of loop nest, whose bounds and conditionals are affine functions that depend only on surrounding loop counters, parameters, and constant, can always be specified by a set of linear inequalities, which defines a polyhedron. Initially, only static control parts (SCoP), those program parts of loop nest which are statically predictable, could be expressed as polyhedra, but now the polyhedral model can operate on general data-dependent control flow analysis [Benabderrahmane *et al.*, 2010][Pouchet, 2010]. The polyhedral framework combines program representation, analysis capabilities, and transformation expressiveness. This model is an applicable alternative for accomplishing optimization and parallelization, but in a simple and more expressive way than the operational or syntactic representation. It operates instance-wise rather than statement-wise. Because of this, the polyhedral representation opens the scope for the high-performance application developers to perform optimizations on instances, and to help programmers to understand program execution minutely.

Abstract syntax trees (AST) have been used as an intermediate representation (IR) since the early days of compiler design. This representation has some limitations in finding and applying complex transformations because each statement appears only once in the abstract syntax tree representation. There is no issue when a statement has to be executed only once in the program, but if any statement has multiple executions (e.g., a statement which is enclosed by a loop), it will also appear only

once. "This problem would limit the program analysis, transformation power and manipulation flexibility of an optimization."[\[Bastoul, 2012\]](#)

2.2.0 Representation of programs in Polyhedra

There are three stages in the polyhedral framework, starting from the high-level program as input to code generation step [\[Benabderrahmane *et al.*, 2010\]](#)[\[Bastoul, 2004b\]](#)[\[Bastoul, 2012\]](#)[\[Pouchet, 2010\]](#). The polyhedral representation is explained soon in section 2.2.1.

2.2.0.0 Program Analysis Phase

First of all, a program written in a high-level language like C, Java, and Python is translated into a polyhedral representation (For details see the examples of section 2.2.1 and 2.2.2). Data dependence analysis is performed at this stage too, i.e., finding out dependence between the instances of the same statement or between the instances of two statements.

2.2.0.1 Transformation Phase

Once a program is altered into the polyhedral representation, all the transformations are made respecting all dependencies. The original semantics of the program is not changed during the transformation phase. Transformations are made to optimize the program performance. It is beneficial if the transformations are made instances-wise rather than statement-wise, and the polyhedral model is capable of doing this task. Plenty of complex transformations are required to optimize the code in scientific and engineering applications. Trees (abstract syntax tree) are very inflexible

data structures and not easy to manipulate in this regard. Whereas, the polyhedral model shows more flexibility in manipulation as compared to AST[Bastoul, 2012].

2.2.0.2 Code Generation Phase

After optimizing the code by applying transformations in polyhedral representation, it is transformed back into a high-level code.

For better understanding of the polyhedral framework, we need to go through the concept of affine functions and Static Control Parts (SCoP).

2.2.0.3 Affine Function

Affine functions are vector-valued or scalar-valued functions of one or more vectors or scalars.

$$f(x_1, \dots, x_n) = A_1x_1 + \dots + A_nx_n + b$$

The coefficients can be scalars or matrices. The constant term is a scalar or a column vector.[Weisstein, a]

2.2.0.4 Parameter

In computer languages, a parameter is a special kind of variable, whose value is unknown at compile time. The value of such a variable is given at run time. Once this value is given, that can not be changed during the execution of the program.

2.2.0.5 Static Control Parts (SCoP)

A set of consecutive statements enclosed in such as in loop/loops, whose bounds and conditionals are affine functions of the surrounding loop index variables and

parameters is called a static control part (SCoP)[Bastoul, 2004a][Bastoul, 2004b]. Also, for effective data dependence analysis, array access functions should be affine functions of the surrounding loop index variables and parameters.

A valid affine expression for a conditional or a loop bound in a SCoP with two loop iterators i, j and two parameters N, P will be of the form

$$a \cdot i + b \cdot j + c \cdot N + d \cdot P + e$$

a, b, c, d and e are arbitrary (possibly 0) integer numbers . This form of affine expression is used in setting the conditions and constraint as follow:

$$a \cdot i + b \cdot j + c \cdot N + d \cdot P + e \geq 0 \text{ or } a \cdot i + b \cdot j + c \cdot N + d \cdot P + e \leq 0$$

An example of an SCoP is as follows

```

for ( $i = 0; i < N; i ++$ )
{
  for ( $j = 0; j < N; j ++$ )
  {
    if ( $N - j > 1$ )
    {
       $\mathbf{A}[i, j] = \mathbf{A}[i - 1, j] + 1$ 
    }
  }
}

```

The bounds $i < N, j < N$ and condition $N - j > 1$ are affine expressions, and N is the parameter.

2.2.1 Iteration Domain

The polyhedral framework operates on instances. A single execution of a statement is known as an instance. The concept of instances can be better understood by the following program code

```
S1 : a = 20;
      for (i = 0; i < 5; i++)
S2 :   A[i] = a;
```

Statement S1 is independent and does not exist inside a loop. It will execute only once during the whole execution of the program, so it has only one instance. However, the statement S2 is enclosed by a loop. It has more than one instances, S2 instances are following:

```
A[0] = a;           // 1st instance
A[1] = a;           // 2nd instance
A[2] = a;           // 3rd instance
A[3] = a;           // 4th instance
A[4] = a;           // 5th Instance
```

In the polyhedral model, statements are considered as a function of the loop iteration(s). For example, the statement S2 is enclosed in a for loop whose index variable is i , instances of this statement would be denoted by $S2(i)$, not just with S2. By representing a statement as a function of the loop index variable, we can target any particular instance of the statement: e.g., $S2(3)$ is the fourth instance of the statement S2. Let's consider another statement S3, instances of this enclosed by

two loops:

```
for(i =1; i <= N; i++)
    for(j =1; j <= N; j++)
S3:        A[i, j] = 20
```

We noticed that S2 is encircled in a single loop. And the statement S3 is enclosed by two loops, it would be represented as a function of two loop integers, i and j , such as $S3(i, j)$. An iteration vector of a statement is an ordered list of the iterators (from outermost to innermost). For example, the iteration vector of S3 is (i, j) : meaning that j is the iterator of the inner loop, and i is the outer loop. And the set of values of the iteration vector is the iteration domain. The bounds of the iterators give the number of instances of that particular statement. For example, the set of loop bounds of the statement S3 can be given by following four constraints:

$$\begin{aligned}i - 1 &\geq 0 \\-i + N &\geq 0 \\j - 1 &\geq 0 \\-j + N &\geq 0\end{aligned}$$

Above given inequalities describe that how many instances, the statement S3 would have, and what are those instances. Sometimes it is not possible to know the statement instances at compile time, as in the above program code, because the inequalities of the loop nest depend on parameters. The graphical form of these inequalities is shown in Figure 2.1.

A polyhedron can be defined as follows:

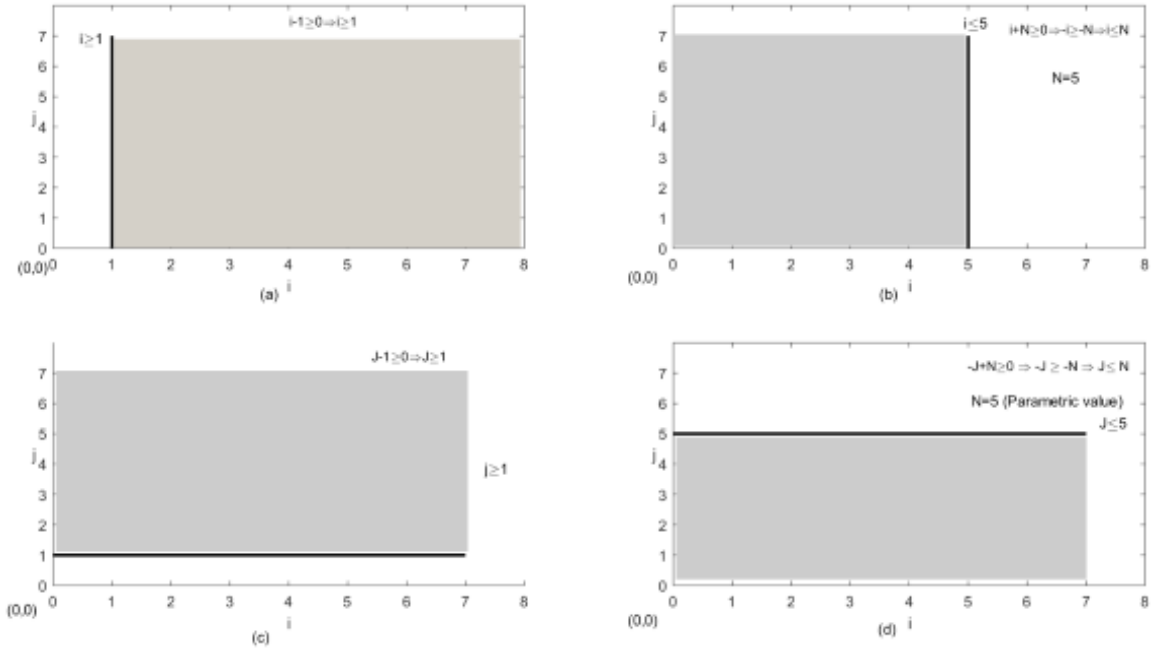


Figure 2.1: Graphical Representation of Inequalities

Definition 2.0 A polyhedron is the set of solutions of a system of affine constraints:

$$P = \{x | \mathbf{A}x + \mathbf{b} \geq 0\} \quad (2.0)$$

\mathbf{A} an integral matrix, \mathbf{b} an integral vector. [Fautrier, 2013]

The iteration domain of the statement S3 is defined by four inequalities which make a polyhedron. Before defining the polyhedron more formally, we need to understand some other basic concepts as follows:

2.2.1.0 Vector Space

A vector space S is a set that is closed under finite vector addition and scalar multiplication. [Weisstein, b]

2.2.1.1 Convex Set

A set S in the vector space \mathbb{R}^n is said to be convex, or is called a convex set, if for any two points x_1 and x_2 in S , the line segment $[x_1, x_2] = \{\lambda x_1 + (1 - \lambda)x_2 \mid 0 \leq \lambda \leq 1\}$, joining x_1 and x_2 , belongs to S . I.e. S is convex if and only if $x_1, x_2 \in S \implies [x_1, x_2] \subseteq S$ for all x_1 and x_2 .

2.2.1.2 Convex Combination

Let $k \geq 1$, and let x_1, \dots, x_k be points in \mathbb{R}^n . For any set of scalars $\lambda_1, \dots, \lambda_k$ with $\lambda_1, \dots, \lambda_k \geq 0$ and $\lambda_1 + \dots + \lambda_k = 1$, $\lambda_1 x_1 + \dots + \lambda_k x_k$ is called a convex combination of x_1, \dots, x_k .

2.2.1.3 Convex Hull

The convex hull of a set S is the set of all convex combinations of the points of S . The convex hull of the points x_1, \dots, x_k is denoted and defined as the set:

$$\text{Conv}(\{x_1, \dots, x_k\}) = \{\sum_{i=1}^k \lambda_i x_i \mid \lambda_i \geq 0, \text{ for } i = 1, \dots, k, \text{ and } \sum_{i=1}^k \lambda_i = 1\};$$

2.2.1.4 Hyperplane:

A set $H = \{x \in \mathbb{R}^n \mid \mathbf{a}^T x = b\}$ with $\mathbf{a} \neq 0$ is called a hyperplane in \mathbb{R}^n ; the vector \mathbf{a} is called the normal of H . Note that for any two points \mathbf{x}_1 and \mathbf{x}_2 in the hyperplane H , holds that $\mathbf{a}^T(\mathbf{x}_1 - \mathbf{x}_2) = b - b = 0$, so that \mathbf{a} is the perpendicular to H . With each hyperplane $H = \{x \in \mathbb{R}^n \mid \mathbf{a}^T x = b\}$ two (closed) half spaces are associated:

$$H^+ = \{x \in \mathbb{R}^n | \mathbf{a}^T x \leq b\} \text{ and } H^- = \{x \in \mathbb{R}^n | \mathbf{a}^T x \geq b\}$$

Note that $H^+ \cap H^- = H$, and that $H^+ \cup H^- = \mathbb{R}^n$.

A polyhedral can also be defined as:

Definition 2.1 *The intersection of a finite number of closed half-spaces is a polyhedron.*

Let us come back to our example. It can be seen in Figure 2.1(a), that the inequality $i \geq 1$, breaks space in two halves: these halves are defined by a constraint on i -axis. One half space is from $i = 1$ onward, and this is the concerning space to find out the instances of this example. Values of i index are positive everywhere in this space, whereas j -axis has both negative and positive parts. The other half space is on the left side of 1, from $i < 1$ backward. The other three inequalities also divide the space into two halves, as can be seen in Figure 2.1(b),(c), and (d). Defined spaces of the inequalities are shown by the shaded area. So, we have a finite number of inequalities, and the intersection of these half-spaces makes a polyhedron, which is shown in Figure 2.2.

The intersection of those spaces give us a closed or bounded polyhedral space, and a bounded polyhedron is known as a polytope. Figure 2.2(e) and (f) both are polytopes. Figure 2.2(e) depicts a rational polyhedron: infinite rational instances exist in this polytope, for example (1.5, 1.5), (2.5, 2.75), (2, 3.75), (4, 3.25), and (4, 3.25) as shown in the figure. But we are concerned with a finite number of instances: the finite number of executions of a statement. Integer polyhedron provides us the finite number of integer instances of the statement S3, as shown in Figure 2.2(f) with

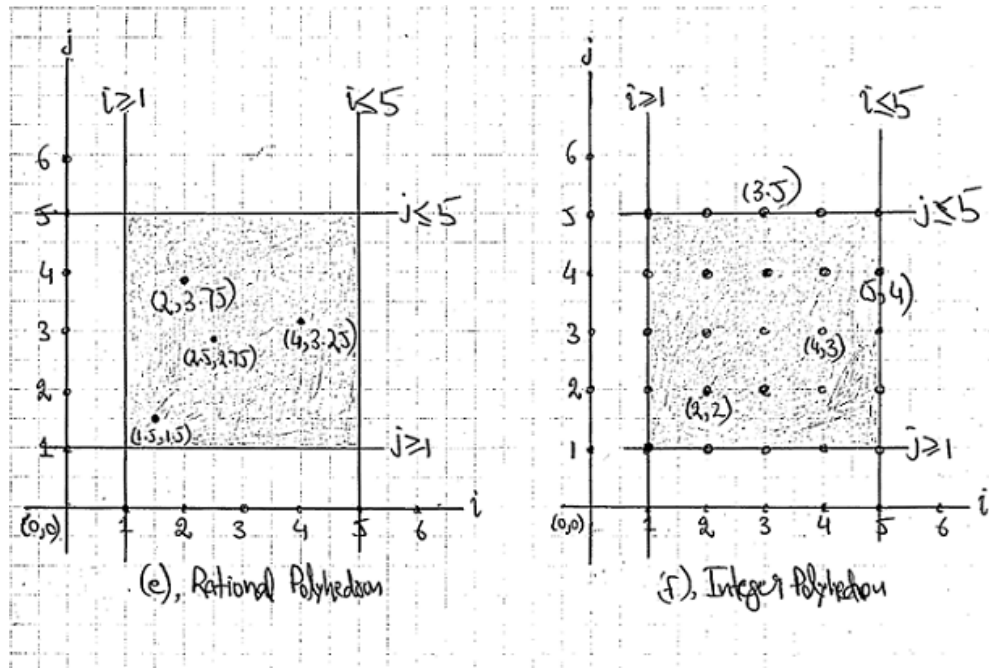


Figure 2.2: A Rational and Integral Polyhedron

bold dots on every instance. Now, it is very easy to see what the instances of the statement S3 are in this polyhedron. Instances of the statement S3 are given below in both cases: one when the parameter N is unknown and second when the parametric value of N is set to 5.

(1,1)	(1,2)	(1,3)	(1,4)	(1, N)
(2,1)	(2,1)	(2,3)	(2,4)	(2, N)
(3,1)	(3,2)	(3,3)	(3,4)	(3, N)
.....
(N ,1)	(N ,2)	(N ,3)	(N ,4)	(N , N)

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)

(3,1) (3,2) (3,3) (3,4) (3,5)

(4,1) (4,2) (4,3) (4,4) (4,5)

(5,1) (5,2) (5,3) (5,4) (5,5)

After setting $N = 5$; the statement S3 has 25 instances. We defined earlier, the iteration vector of the statement S3 as (i, j) , but we can observe now that the iteration vector (i, j) , is not enough to describe the iteration domain of S3. So, we need to add more dimensions in this iteration vector, such as a parameter N and a constant. The new iteration vector of the statement S3 is $(i, j, N, 1)$ which will be called the homogeneous iteration vector. The set of values of this iteration vector gives the complete domain of S3. And the iteration domain of the statement S3 can also be represented in mathematical form as follows:

$$D_{S3} = \{(i, j) \in Z^2 \mid 1 \leq i \leq N \wedge 1 \leq j \leq N\}$$

The matrix notation is used to elaborate and to operate on this polyhedral model. The set of constraints of the running statement S3 can be expressed in the form of domain matrix \times iteration vector ≥ 0 . The inequalities or the constraints are the following

$$i - 1 \geq 0$$

$$-i + N \geq 0$$

$$j - 1 \geq 0$$

$$-j + N \geq 0$$

We can transform these inequalities in matrix form by taking the complete iteration vector $(i, j, N, 1)$ of the statement S3, and by showing the coefficients of each element of the iteration vector in each inequality. The coefficients are 0 or 1 in this example. It can be shown as below:

$$\begin{aligned}
1i - 1 &\geq 0 \implies 1i + 0j + 0N - 1 \geq 0 \\
-1i + 1N &\geq 0 \implies -1i + 0j + 0N + 0 \geq 0 \\
1j - 1 &\geq 0 \implies 0i + 1j + 0N - 1 \geq 0 \\
-1j + 1N &\geq 0 \implies 0i + 1j + 1N + 0 \geq 0
\end{aligned}$$

We can extract $\mathbf{Ax} \geq \mathbf{b}$, from above representation is as follows:

$$\begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

where \mathbf{A} is the matrix:

$$\begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 1 & 0 \end{bmatrix}$$

\mathbf{x} is an iteration vector (represented in the form of a column vector): $\mathbf{x} = (i, j, N, 1)^T$, and \mathbf{b} is the column vector of constants: $\mathbf{b} = (0, 0, 0, 0)^T$.

2.2.2 Dependence In Polyhedra

The polyhedral framework deals with instances of the statements rather than the statements, as the conventional approach works with the statements for data dependence analysis. Let us consider some examples to understand that how this framework works with data dependence. The program code

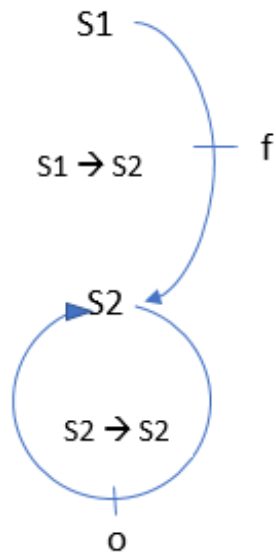


Figure 2.3: Dependence

```

for (i = 0; i < N ; i++)
{
    for (j = 0; j < N; j++)
    {
S1:        A[i , j] = B[i , j] + 2 × D[j]
    }
}
for (k = 0; k < N; k++)
{
    for (m = 0; m < N; m++ )
    {
S2:        C[k] = C[k] + A[m , k]
    }
}
  
```

}

The Figure 2.3 shows dependence relation between the statements S1 and S2, and the statement S2 on itself. There is a flow dependency from the instances of the statement S1 to the instances of the statement S2, and there exists flow and output dependence on the instances of the statement S2 on itself.

We have seen in the iteration domain section how program code can be represented in the polyhedral framework. The dependence polyhedron of the edge from S1 \rightarrow S2 is shown below, both in the algebraic and the matrix form. The iteration vector of this edge is $(i, j, k, m, N, 1)$.

$$\begin{array}{r}
 1 \quad i \geq 0 \\
 2 \quad -i + N - 1 \geq 0 \\
 3 \quad j \geq 0 \\
 4 \quad -j + N - 1 \geq 0 \\
 5 \quad k \geq 0 \\
 6 \quad -k + N - 1 \geq 0 \\
 7 \quad -m \geq 0 \\
 8 \quad -m + N - 1 \geq 0 \\
 \hline
 9 \quad i - m = 0 \\
 10 \quad j - k = 0
 \end{array}
 \left(
 \begin{array}{cccccc}
 1 & 0 & 0 & 0 & 0 & 0 \\
 -1 & 0 & 0 & 0 & 1 & -1 \\
 0 & 1 & 0 & 0 & 0 & 0 \\
 0 & -1 & 0 & 0 & 1 & -1 \\
 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & -1 & 0 & 1 & -1 \\
 0 & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & -1 & 1 & -1 \\
 \hline
 1 & 0 & 0 & -1 & 0 & 0 \\
 0 & 1 & -1 & 0 & 0 & 0
 \end{array}
 \right)
 \left[
 \begin{array}{c}
 i \\
 j \\
 k \\
 m \\
 N \\
 1
 \end{array}
 \right]
 \left(
 \begin{array}{c}
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \hline
 = 0 \\
 = 0
 \end{array}
 \right)$$

The left most column is giving the row number to the each inequality/equality. The next block is representing the algebraic form of the dependence polyhedron of the edge S1 \rightarrow S2. After that there is a matrix, let us call this matrix as $A^{10 \times 6}$, and the

column after this matrix is the iteration vector, let us call it x , and the last column is also is a vector of constants, let us call it b . Then this representation can be denoted as follows

$$A^{10 \times 6} x^{6 \times 1} \geq b^{10 \times 1}$$

This notation is used in the polyhedral tools for representation and manipulation. The rows of matrix A are also the vectors which are denoted by A_1, A_2, \dots, A_{10} . The elements of the iteration vector x are $(i, j, k, m, N, 1)$. The vector b has zero (0) value for all of its elements. A vector can be represented in both ways, as a row vector or a column vector. The vectors of matrix A are represented in the form of rows and the vector x and b are represented in the form of a column. The first four rows of inequalities show the iteration domain of the statement S1. Notice that the statement S1 is surrounded by two loops i and j . Two inequalities can specify the bounds of a loop: one for the lower bound and other for upper bound. So, there are two inequalities for i loop, two for j loop and 4 in total to specify the iteration domain of the statement S1. The next four rows from 5 to 8 give the iteration domain of the statement S2, and the last two rows of equality, 9 and 10 reflecting the dependence relation from the instances of the statement S1 to the instances of the statement S2. We would have a better understanding of this dependence relation between the instances of the statements S1 and S2 by an algebraic form of the equality 9 and 10.

$$9 : 1i + 0j + 0k - 1m + 0N + 0 = 0 \Rightarrow 1i - 1m = 0 \Rightarrow 1i = 1m \Rightarrow i = m \Rightarrow m = i$$

$$10 : 0i + 1j - 1k + 0m + 0N + 0 = 0 \Rightarrow 1j - 1k = 0 \Rightarrow 1j = 1k \Rightarrow j = k \Rightarrow k = j$$

These two equations describe that every instance of the statement S2 depends on an element of array A, which is assigned to an instance of the statement S1. The relation of an instance of the statement S2(k, m) with an instance of the statement S1(i, j) is given in the above equations. This means that the execution of any such instance of S2 cannot be started until that specific instance of A[i, j] is not assigned by the statement S1, where m becomes equal to i and k becomes equal to j .

The other dependency edge of this program segment is S2 \rightarrow S2 as shown in the Figure 2.3. For this edge, the source vertex is same as the destination vertex. Let us write the dependence polyhedron for this edge. The source loop iterators are k and m , let us name destination loop iterators with k' and m' to differentiate the source and the destination. The iteration vector for this polyhedron is $(k, m, k', m', N, 1)$.

The dependence polyhedron of this edge is

$$\begin{array}{r}
 1 \quad k \geq 0 \\
 2 \quad -k + N - 1 \geq 0 \\
 3 \quad m \geq 0 \\
 4 \quad -m + N - 1 \geq 0 \\
 5 \quad k' \geq 0 \\
 6 \quad -k' + N - 1 \geq 0 \\
 7 \quad m' \geq 0 \\
 8 \quad -1m' + N - 1 \geq 0 \\
 \hline
 9 \quad k - k' = 0 \\
 10 \quad m - m' + 1 = 0
 \end{array}
 \left(\begin{array}{cccccc}
 1 & 0 & 0 & 0 & 0 & 0 \\
 -1 & 0 & 0 & 0 & 1 & -1 \\
 0 & 1 & 0 & 0 & 0 & 0 \\
 0 & -1 & 0 & 0 & 1 & -1 \\
 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & -1 & 0 & 1 & -1 \\
 0 & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & -1 & 1 & -1 \\
 \hline
 1 & 0 & -1 & 0 & 0 & 0 \\
 0 & 1 & 0 & -1 & 1 & 0
 \end{array} \right)
 \left[\begin{array}{c}
 k \\
 m \\
 k' \\
 m' \\
 N \\
 1
 \end{array} \right]
 \left(\begin{array}{c}
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \geq 0 \\
 \hline
 = 0 \\
 = 0
 \end{array} \right)$$

The statement S2 of this program segment is $C[k] = C[k] + A[m, k]$. The source for the edge $S2 \rightarrow S2$ is $C[k] = C[k] + A[m, k]$, and the destination is $C[k'] = C[k'] + A[m', k']$. The loop indexes (k, m) represent current instance and (k', m') represent next instance. The instances (k', m') depend on the instances (k, m) in such a way that for N iterations of the inner loop (m), the outer loop (k) value will remain same for the computation of $C[k]$. So $k' = k$, as the row number 9 depicts this relation.

$$1k + 0m - 1k' + 0m' + 0N + 0 = 0 \Rightarrow 1k - 1k' = 0 \Rightarrow 1k = 1k' \Rightarrow k' = k$$

Whereas, for every next instance the value of m increases by 1. The relation of m' to m is reflected by row number 10.

$$\begin{aligned} 0k + 1m + 0k' - 1m + 0N + 1 &= 0 \Rightarrow \\ 1m - 1m' + 1 &= 0 \Rightarrow 1m + 1 = 1m' \Rightarrow m' = m + 1 \end{aligned}$$

2.3 Summary

In this chapter, we briefly talked about the background of memory management, the basis of the polyhedral framework, linear and nonlinear programming, and optimization with the polyhedral framework. The transformation of a high-level program into a polyhedral modal and the application of dependence analysis with this model are illustrated with examples. In the next chapter, we will show how the Temporal Data Placement (TDP) methodology works by using a Mixed Integer Non-Linear Programming solver with the help of different examples to minimize the maximum distance problem.

Chapter 3

Minimize the Maximum Distance

Research has been carried out aiming to minimize the distance between processors and memories in the compiler domain based on the polyhedral framework. Lu et al [Lu *et al.*, 2009] have developed a compile-time framework for non-uniform cache architecture (NUCA) in which data are placed according to non-canonical layout in the banks of shared cache along with the loop transformations. Therefore, the processors get most of the required data in their local banks and rarely need to access the remote banks of the shared cache. Zhang et al [Zhang *et al.*, 2011] also targets NUCA for multithreaded applications using a data layout transformation strategy. In their approach the compiler places data according to the access pattern of different threads rather than the conventional linear data layout like row/column/diagonal major, while considering the on-chip cache hierarchy of the target multicore architecture.

Both preceding techniques address non-uniform cache architecture, which could

be a good solution for small problems because the cache can hold a small amount of data. However, for large-scale scientific and engineering problems we must re-visit how data is placed in main memory. Moreover, FPGA, GPU, and embedded systems do not have cache in their architecture.

Additionally, modern personal computers and server-class systems are built on non-uniform memory architecture (NUMA), which is a good platform for solving large-scale problems. A NUMA architecture consists of several nodes, and each node is composed of the processing unit(s) and a portion of RAM. In polyhedral framework, Susungi et al [Susungi *et al.*, 2017] have proposed a compiler-based approach for NUMA systems. They have developed a parallel intermediate language IVIE, which places the data on NUMA nodes according to the two policies: interleaved allocation, and allocation on a specific node. In interleaved allocation, IVIE has got such constructs that mapped data blocks (minimum of a page (4096 bytes)) on NUMA nodes in a round-robin fashion. The second technique can map an array of data to a specific node to the NUMA nodes. Other than the compile-time approach, considerable research has been done for run-time address binding optimization for NUMA systems, which is discussed in the literature overview chapter.

3.0 Temporal Data Placement (TDP)

Our approach called Temporal Data placement is a compile-time technique and targeting NUMA[Norvell *et al.*, 2017][Sindhu *et al.*, 2019] and distributed memory systems. It provides a systematic way to place data in the local or nearby memory to those processing nodes which are going to compute that data. TDP is capable of placing

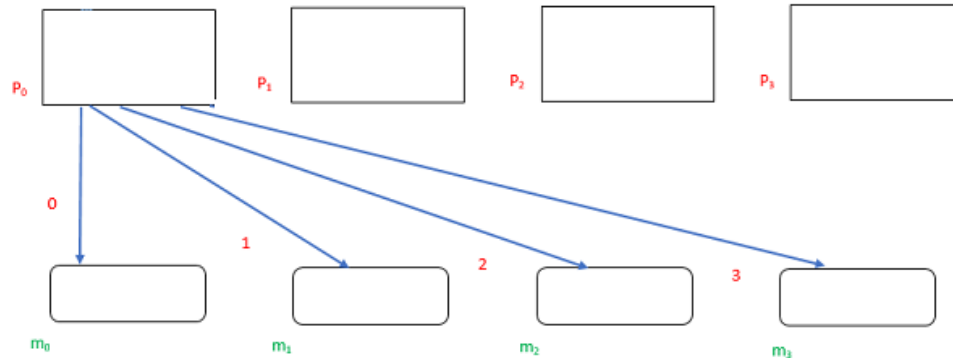


Figure 3.0: NUMA Machine 1

some specific element(s) of an array in a particular memory node rather than the whole array or a page. In TDP we are considering only static variables. Usually, a compiler maps the program's static variables to memory addresses with the help of a function. We are going to use a relation rather than a function for the sake of data replication, because a relation allows variables to be represented by more (or less) than one memory address.

How can TDP place data locations so that data is close to the processors that use it? Suppose we have a linear topology architecture of a NUMA machine with four processors and memories pairs, as shown in Figure 3.0. The processors with their indexes are represented at the top of the figure from p_0 to p_3 , and the memories along with the indexes are shown from m_0 to m_3 at the bottom.

Each processor of a NUMA machine can access data from all memories; however, access cost depends on the distance. We are assuming that the distance between processor and memory is an absolute value of the difference in their index numbers. For the example hardware, the distances range from 0 to 3. In Figure 3.0, the blue arrows annotated with red numbers are going out from p_0 to all memories showing the

impact of distance in terms of the difference between processor and memory number. The distance will be 0 if a processor gets data from its local memory like p_0 gets data from m_0 , p_1 gets data from m_1 , and so on. However, if the required data for a processor is placed at a remote memory node, then the data has to travel a longer distance, which will increase the access cost. Let us consider the following example to explain that, how can we minimize the maximum distance data needs to travel. Suppose we want to execute the following program code on the example machine.

I(0): $a[0] := a[0] + 1$

I(1): $a[1] := a[1] + 1$

I(2): $a[2] := a[2] + 1$

I(3): $a[3] := a[3] + 1$

There are four instances/computations ι_0 to ι_3 and four program locations l_0 to l_3 . Each instance is storing and fetching data to/from the same index number of program location like the instance ι_0 stores/fetches to/from l_0 , ι_1 stores/fetches to/from l_1 , and so on. As we have four processors in the underlying machine, we can compute one instance on each processor simultaneously. We suppose that each processor is computing its corresponding instance, so p_0 is computing ι_0 , p_1 is computing ι_1 , and so on. The details of set representation, access relation, computation/instance placement, and data placement are given in the section of polyhedral representation.

We seek to find a placement of the data that minimizes the maximum distance required to any store or fetch operation. In the running example, this optimal maximum distance will be 0.

3.0.0 The Problem Formalization

We formulate the data placement problem as an optimization problem with the following known and unknown variables:

- Known: \mathcal{I} , \mathcal{P} , \mathcal{L} , and \mathcal{M} are respectively the set of instances, processors, program locations, and memories. Π , Σ , Φ are computation placement, store, and fetch relations, respectively. Π relates instances and processor, Σ and Φ relate instances to locations.
- Unknown: Δ , f , and d are respectively the data placement relation (between locations and memories), a function that indicates which memory should be used for fetches, and d is an integer representing an upper bound on the distance data must travel.
- Constraints: Distances of stores can not exceed d

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma(\iota, l) \wedge \Delta(l, m) \Rightarrow |p - m| \leq d \quad (3.0)$$

For each read there is a memory no farther away than d , and f indicates which memory that is:

$$\begin{aligned} \forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Phi(\iota, l) \wedge m = f(\iota, p, l) \\ \Rightarrow m \in \mathcal{M} \wedge |p - m| \leq d \wedge \Delta(l, m) \end{aligned} \quad (3.1)$$

- Objective function: d

To make this problem tractable and to make its solution easier to use in code generation we will assume that all sets and relations are finite unions of \mathbb{Z} -polyhedra and that f is a collection of affine functions, one for each array.

We will represent this problem as a nonlinear optimization problem. However, the problem has existential and universal quantifiers. To eliminate these quantifiers we will use the affine form of the Farkas lemma [Feautrier, 1992]. This lemma transforms a system of affine inequalities with quantifiers into a system of affine equalities by adding Farkas multipliers; non-negative variables that range from 0 to 1.

3.0.0.0 Affine form of Farkas Lemma

Let D be a nonempty polyhedron defined by p affine inequalities

$$a_k x + b_k \geq 0, \quad k = 1, p \tag{3.2}$$

then any affine form ψ is nonnegative everywhere in D iff it is a positive affine combination [Feautrier, 1992]:

$$\psi(x) = \lambda_0 + \sum_k \lambda_k (a_k x + b_k), \quad \lambda_k \geq 0 \tag{3.3}$$

3.0.1 Algorithm

There are three steps of the algorithm

1. Transform the problem into a polyhedral form. This transformation will convert the problem into an algebraic notation in the form of inequalities.
2. Apply Farkas Lemma to the output of step 1. This step gives us constraints in the form of equations, along with some more unknowns: one λ variable for each inequation of the problem set and λ_0 .
3. The output of the step 2 will be the input to a mixed-integer nonlinear optimizer, which will give us the values of all unknowns; the Δ matrix, the distance d , coefficients defining the affine function f , and the λ variables.

3.0.2 Polyhedral Representation

Our proposed algorithm is built on the polyhedral framework [Benabderrahmane *et al.*, 2010][Bastoul, 2004b][Bastoul, 2004a][Bastoul, 2012], which is based on algebraic representations of relations. This algorithm deals with a known number of processors, memories, instances and the program locations. So the set of the instances, processors, memories, and the program locations will be fixed and known at compile-time.

Firstly, we will transform a problem into a polyhedral form. The inequalities can be directly extracted from the program text and underlying hardware. The running example sets; \mathcal{I} , \mathcal{P} , \mathcal{L} , and \mathcal{M} are represented by the following inequalities respectively.

$$\iota \geq 0 \quad \text{and} \quad -\iota + 3 \geq 0$$

$$p \geq 0 \quad \text{and} \quad -p + 3 \geq 0$$

$$l \geq 0 \quad \text{and} \quad -l + 3 \geq 0$$

$$m \geq 0 \quad \text{and} \quad -m + 3 \geq 0$$

Instances are assigned to processors by an affine relation Π . We have four instances in the running example, and we place each instance on each processor, $\Pi(\iota, p) : p = \iota$. This can be represented by the following two inequalities:

$$\iota - p \geq 0 \quad \text{and} \quad -\iota + p \geq 0$$

The fetch and store access relations are between instances and program location: $\Phi, \Sigma \subseteq \mathcal{I} \times \mathcal{L}$. A program location can be placed in more than one memory. In the running example, every instance ι fetches to its corresponding program location. So, the fetch relation is, $\Sigma(i, l) : \iota = l$. We can represent this fetch relation as follows

$$\iota - l \geq 0 \quad \text{and} \quad -\iota + l \geq 0$$

In the case of the store, we have to store the output to every memory where that program location is represented. In the example, every instance stores to its corresponding program location. The store relation is, $\Phi(i, l) : \iota = l$ or:

$$\iota - l \geq 0 \quad \text{and} \quad -\iota + l \geq 0$$

The placement of locations in memories is represented by $\Delta : \mathcal{L} \leftrightarrow \mathcal{M}$. The delta relation is unknown, and we want to find a suitable Δ matrix to minimize

the maximum distance between processors and memories. For this problem a delta relation comprising of two half-spaces is sufficient to achieve the optimal result. We can represent this relation as follows

$$\Delta_{00}l + \Delta_{01}m + \Delta_{02} \geq 0$$

$$\Delta_{10}l + \Delta_{11}m + \Delta_{12} \geq 0$$

All of above given Δ variables are unknowns.

For a different problem, if two half-spaces are not sufficient to get the optimal result, then we keep adding more half spaces one by one until we get the optimal output.

Finally the unknown affine function f can be represented by four real variables and the constraints are as follows

$$au + bp + cl + e - m \geq 0$$

$$-au - bp - cl - e + m \geq 0$$

3.0.3 Implementation of Farkas Lemma

We can represent the left-hand side of the store constraint 3.0 of the problem, $\Pi(\iota, p) \wedge \Sigma(\iota, l) \wedge \Delta(l, m)$, by a polyhedron of four dimensions. Let us define this polyhedron according to definition 2.0 and equation 3.2 of the affine form of Farkas Lemma, named as D_s :

$$D_s = \{\mathbf{x} \in \mathbb{N}^4 \mid \mathbf{A}_s \mathbf{x} + \mathbf{b}_s \geq 0\}$$

where $\mathbf{x} = (\iota, p, l, m)$, \mathbf{A} is an integral matrix, and \mathbf{b} is an integral vector.

$$\mathbf{A}_s = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & \Delta_{00} & \Delta_{01} \\ 0 & 0 & \Delta_{10} & \Delta_{11} \end{bmatrix} \quad \mathbf{b}_s = \begin{pmatrix} 0 \\ 3 \\ 0 \\ 3 \\ 0 \\ 3 \\ 0 \\ 3 \\ 0 \\ 0 \\ 0 \\ 0 \\ \Delta_{02} \\ \Delta_{12} \end{pmatrix}$$

Similarly we can form a polyhedron for the left-hand side of the fetch constraint 3.1 of the problem, $\Pi(\iota, p) \wedge \Phi(\iota, l) \wedge m = f(\iota, p, l)$. Let us call it D_f :

$$D_f = \{\mathbf{x} | \mathbf{A}_f \mathbf{x} + \mathbf{b}_f \geq 0\}$$

$$\mathbf{A}_f = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ a & b & c & -1 \\ -a & -b & -c & 1 \end{bmatrix} \quad \mathbf{b}_f = \begin{pmatrix} 0 \\ 3 \\ 0 \\ 3 \\ 0 \\ 3 \\ 0 \\ 0 \\ 0 \\ 0 \\ e \\ -e \end{pmatrix}$$

where $a, b, c,$ and e are real variables.

From the right-side of the implication of the store constraint [3.0](#), $|p - m| \leq d$, we get the following two constraints

$$-p + m + d \geq 0 \quad \text{and} \quad p - m + d \geq 0$$

So, for the sake of Farkas lemma application, we need to further divide constraint [3.0](#) into the following two sub-constraints:

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma(\iota, l) \wedge \Delta(l, m) \Rightarrow -p + m + d \geq 0 \quad (3.4)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma(\iota, l) \wedge \Delta(l, m) \Rightarrow p - m + d \geq 0 \quad (3.5)$$

Now we can apply Farkas lemma for each of the above constraints separately according to equation 3.3 of the Farkas lemma.

$$\psi(x) = \lambda_0 + \sum_k \lambda_k (a_k x + b_k)$$

The polyhedron D_s has fourteen affine inequalities, so $p = 14$. And, we can write the above formula for 3.4 as follows

$$-p + m + d = \lambda_0 + \sum_k \lambda_k ((\mathbf{A}_s)_k \mathbf{x} + (\mathbf{b}_s)_k)$$

Now, by expanding the above formula:

$$\begin{aligned} -p + m + d = & \lambda_0 + \lambda_1 \iota - \lambda_2 \iota + 3\lambda_2 + \lambda_3 p - \lambda_5 p + 3\lambda_5 + \lambda_6 l - \lambda_7 l + \lambda_7 \mathfrak{Z} + \lambda_7 m - \\ & \lambda_8 m + \lambda_8 \mathfrak{Z} + \lambda_9 \iota - \lambda_9 p - \lambda_{10} \iota + \lambda_{10} p + \lambda_{11} \iota - \lambda_{11} l - \lambda_{12} \iota + \lambda_{12} l \\ & \lambda_{13} \Delta_{00} l + \lambda_{13} \Delta_{01} m + \lambda_{13} \Delta_{02} + \lambda_{14} \Delta_{10} l + \lambda_{14} \Delta_{11} m + \lambda_{14} \Delta_{12} \end{aligned}$$

By combing the like terms

$$\begin{aligned} -p + m + d = & \lambda_1 \iota - \lambda_2 \iota + \lambda_9 \iota - \lambda_{10} \iota + \lambda_{11} \iota - \lambda_{12} \iota + \\ & \lambda_3 p - \lambda_4 p - \lambda_9 p + \lambda_{10} p + \\ & \lambda_5 l - \lambda_6 l - \lambda_{11} l + \lambda_{12} l + \lambda_{13} l \Delta_{00} + \lambda_{14} l \Delta_{10} + \\ & \lambda_7 m - \lambda_8 m + \lambda_{13} m \Delta_{01} + \lambda_{14} m \Delta_{11} + \\ & \lambda_0 + 3\lambda_2 + 3\lambda_4 + 3\lambda_6 + 3\lambda_8 + \lambda_{13} \Delta_{02} + \lambda_{14} \Delta_{12} \end{aligned}$$

By factoring out ι , p , l , and m respectively

$$\begin{aligned} -p + m + d = & \iota(\lambda_1 - \lambda_2 + \lambda_9 - \lambda_{10} + \lambda_{11} - \lambda_{12}) + \\ & p(\lambda_3 - \lambda_4 - \lambda_9 + \lambda_{10}) + \\ & l(\lambda_5 - \lambda_6 - \lambda_{11} + \lambda_{12} + \lambda_{13} \Delta_{00} + \lambda_{14} \Delta_{10}) + \\ & m(\lambda_7 - \lambda_8 + \lambda_{13} \Delta_{01} + \lambda_{14} \Delta_{11}) + \end{aligned}$$

$$\lambda_0 + 3\lambda_2 + 3\lambda_4 + 3\lambda_6 + 3\lambda_8 + \lambda_{13}\Delta_{02} + \lambda_{14}\Delta_{12}$$

Now, by equating coefficients of i , p , l , m , and constants terms of both sides of the formula, we get the following five equations as the output of Farkas Lemma implementation.

1. $\lambda_1 - \lambda_2 + \lambda_9 - \lambda_{10} + \lambda_{11} - \lambda_{12} = 0$
2. $\lambda_3 - \lambda_4 - \lambda_9 + \lambda_{10} = -1$
3. $\lambda_5 - \lambda_6 - \lambda_{11} + \lambda_{12} + \lambda_{13}\Delta_{00} + \lambda_{14}\Delta_{10} = 0$
4. $\lambda_7 - \lambda_8 + \lambda_{13}\Delta_{01} + \lambda_{14}\Delta_{11} = 1$
5. $\lambda_0 + 3\lambda_2 + 3\lambda_4 + 3\lambda_6 + 3\lambda_8 + \lambda_{13}\Delta_{02} + \lambda_{14}\Delta_{12} = d$

Then we need to apply this same procedure to the constraint 3.5. We can write equation 3.3 of the Farkas lemma for 3.5 as:

$$p - m + d = \lambda_{15} + \sum_k \lambda_{k+15}((\mathbf{A}_s)_k \mathbf{x} + (\mathbf{b}_s)_k)$$

This time the first lambda variable will be λ_{15} , and k will go from 16 to 29.

$$\begin{aligned} p - m + d = & \lambda_{15} + \lambda_{16}(\iota) + \lambda_{17}(-\iota + 3) + \lambda_{18}(p) + \lambda_{19}(-p + 3) + \lambda_{20}(l) + \lambda_{21}(-l + 3) + \\ & \lambda_{22}(m) + \lambda_{23}(-m + 3) + \lambda_{24}(\iota - p) + \lambda_{25}(-\iota + p) + \lambda_{26}(\iota - l) + \lambda_{27}(-\iota + l) \\ & \lambda_{28}(\Delta_{00}l + \Delta_{01}m + \Delta_{02}) + \lambda_{29}(\Delta_{10}l + \Delta_{11}m + \Delta_{12}) \end{aligned}$$

By applying distributive property

$$\begin{aligned} p - m + d = & \lambda_{15} + \lambda_{16}\iota - \lambda_{17}\iota + 3\lambda_{17} + \lambda_{18}p - \lambda_{19}p + 3\lambda_{19} + \lambda_{20}l - \lambda_{21}l + 3\lambda_{21} + \\ & \lambda_{22}m - \lambda_{23}m + 3\lambda_{23} + \lambda_{24}\iota - \lambda_{24}p - \lambda_{25}\iota + \lambda_{25}p + \lambda_{26}\iota - \lambda_{26}l - \lambda_{27}\iota + \lambda_{27}l \\ & \lambda_{28}\Delta_{00}l + \lambda_{28}\Delta_{01}m + \lambda_{28}\Delta_{02} + \lambda_{29}\Delta_{10}l + \lambda_{29}\Delta_{11}m + \lambda_{29}\Delta_{12} \end{aligned}$$

Now, by combining the like terms and by equating coefficients of i , p , l , m , and constants terms of both sides of the formula, we get the following five equations

6. $\lambda_{16} - \lambda_{17} + \lambda_{24} - \lambda_{25} + \lambda_{26} - \lambda_{27} = 0$

7. $\lambda_{18} - \lambda_{19} - \lambda_{24} + \lambda_{25} = 1$
8. $\lambda_{20} - \lambda_{21} - \lambda_{26} + \lambda_{27} + \lambda_{28}\Delta_{00} + \lambda_{29}\Delta_{10} = 0$
9. $\lambda_{22} - \lambda_{23} + \lambda_{28}\Delta_{01} + \lambda_{29}\Delta_{11} = -1$
10. $\lambda_{15} + 3\lambda_{17} + 3\lambda_{19} + 3\lambda_{21} + 3\lambda_{23} + \lambda_{27}\Delta_{02} + \lambda_{28}\Delta_{12} = d$

The right-hand side of the fetch constraint [3.1](#), $m \in M \wedge |p - m| \leq d \wedge \Delta(l, m)$, gives the following six inequalities:

$$\begin{aligned}
m \in M &\equiv (m \geq 0 \quad \text{and} \quad -m + 3 \geq 0) \\
|p - m| \leq d &\equiv (-p + m + d \geq 0 \quad \text{and} \quad p - m + d \geq 0) \\
\Delta(l, m) &\equiv (\Delta_{00}l + \Delta_{01}m + \Delta_{02} \geq 0 \quad \text{and} \\
&\quad \Delta_{10}l + \Delta_{11}m + \Delta_{12} \geq 0)
\end{aligned}$$

Based on the above inequalities, we can divide fetch constraint [3.1](#) into the following six sub-constraints:

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathbb{R}. \Pi(\iota, p) \wedge \Phi(\iota, l) \wedge m = f(\iota, p, l) \Rightarrow m \geq 0 \quad (3.6)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathbb{R}. \Pi(\iota, p) \wedge \Phi(\iota, l) \wedge m = f(\iota, p, l) \Rightarrow -m + 3 \geq 0 \quad (3.7)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathbb{R}. \Pi(\iota, p) \wedge \Phi(\iota, l) \wedge m = f(\iota, p, l) \Rightarrow -p + m + d \geq 0 \quad (3.8)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathbb{R}. \Pi(\iota, p) \wedge \Phi(\iota, l) \wedge m = f(\iota, p, l) \Rightarrow p - m + d \geq 0 \quad (3.9)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathbb{R}. \Pi(\iota, p) \wedge \Phi(\iota, l) \wedge m = f(\iota, p, l) \Rightarrow \Delta_{00}l + \Delta_{01}m + \Delta_{02} \geq 0 \quad (3.10)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathbb{R}. \Pi(\iota, p) \wedge \Phi(\iota, l) \wedge m = f(\iota, p, l) \Rightarrow \Delta_{10}l + \Delta_{11}m + \Delta_{12} \geq 0 \quad (3.11)$$

Now, we need to apply Farkas lemma separately for each of the above constraints.

We can represent equation 3.3 for constraint 3.6 as:

$$-m = \lambda_{30} + \sum_k \lambda_{k+30} ((\mathbf{A}_f)_k \mathbf{x} + (\mathbf{b}_f)_k)$$

The polyhedron D_f comprises of twelve affine inequalities. The lambda variables start from λ_{30} , and k will go from 31 to 42. Now by expanding the above formula for the inequalities:

$$\begin{aligned} m = & \lambda_{30} + \lambda_{31}\iota - \lambda_{32}\iota + 3\lambda_{32} + \lambda_{33}p - \lambda_{34}p + \lambda_{34}3 + \lambda_{35}l - \lambda_{36}l + 3\lambda_{36} + \\ & \lambda_{37}\iota - \lambda_{37}p - \lambda_{38}\iota + \lambda_{38}p + \lambda_{39}\iota - \lambda_{39}l - \lambda_{40}\iota + \lambda_{40}l \end{aligned}$$

$$\lambda_{41}au + \lambda_{41}bp + \lambda_{41}cl - \lambda_{41}m + \lambda_{41}e - \lambda_{42}au - \lambda_{42}bp - \lambda_{42}cl + \lambda_{42}m - \lambda_{42}e$$

By combining the like terms

$$\begin{aligned} m = & \iota(\lambda_{31} - \lambda_{32} + \lambda_{37} - \lambda_{38} + \lambda_{39} - \lambda_{40} + \lambda_{41}a - \lambda_{42}a) + \\ & p(\lambda_{33} - \lambda_{34} - \lambda_{37} + \lambda_{38} + \lambda_{41}b - \lambda_{42}b) + \\ & l(\lambda_{35} - \lambda_{36} - \lambda_{39} + \lambda_{40} + \lambda_{41}c - \lambda_{42}c) + \\ & m(\lambda_{41} + \lambda_{42}) + \\ & \lambda_{30} + 3\lambda_{32} + 3\lambda_{34} + 3\lambda_{36} + \lambda_{41}e - \lambda_{42}e \end{aligned}$$

Now, by equating coefficients of i , p , l , m , and constant terms of both sides of the formula. We will get following five equations

11. $\lambda_{31} - \lambda_{32} + \lambda_{37} - \lambda_{38} + \lambda_{39} - \lambda_{40} + \lambda_{41}a - \lambda_{42}a = 0$
12. $\lambda_{33} - \lambda_{34} - \lambda_{37} + \lambda_{38} + \lambda_{41}b - \lambda_{42}b = 0$
13. $\lambda_{35} - \lambda_{36} - \lambda_{39} + \lambda_{40} + \lambda_{41}c - \lambda_{42}c = 0$
14. $-\lambda_{41} + \lambda_{42} = 1$
15. $\lambda_{30} + 3\lambda_{32} + 3\lambda_{34} + 3\lambda_{36} + \lambda_{41}e - \lambda_{42}e = 0$

Similarly, by applying Farkas lemma on the rest of the five constraints [3.7](#) to [3.11](#), we get the following 25 equations. Five equations from each constraint.

16. $\lambda_{44} - \lambda_{45} + \lambda_{50} - \lambda_{51} + \lambda_{52} - \lambda_{53} + \lambda_{54}a - \lambda_{55}a = 0$
17. $\lambda_{46} - \lambda_{47} - \lambda_{50} + \lambda_{51} + \lambda_{54}b - \lambda_{55}b = 0$
18. $\lambda_{48} - \lambda_{49} - \lambda_{52} + \lambda_{53} + \lambda_{54}c - \lambda_{55}c = 0$
19. $-\lambda_{54} + \lambda_{55} = -1$
20. $\lambda_{43} + 3\lambda_{45} + 3\lambda_{47} + 3\lambda_{49} + \lambda_{54}e - \lambda_{55}e = 0$
21. $\lambda_{57} - \lambda_{58} + \lambda_{63} - \lambda_{64} + \lambda_{52} - \lambda_{66} + \lambda_{67}a - \lambda_{68}a = 0$
22. $\lambda_{59} - \lambda_{60} - \lambda_{63} + \lambda_{64} + \lambda_{67}b - \lambda_{68}b = -1$
23. $\lambda_{61} - \lambda_{62} - \lambda_{65} + \lambda_{66} + \lambda_{67}c - \lambda_{68}c = 0$

24. $-\lambda_{67} + \lambda_{68} = 1$
25. $\lambda_{56} + 3\lambda_{58} + 3\lambda_{60} + 3\lambda_{62} + \lambda_{67}e - \lambda_{68}e = d$
26. $\lambda_{70} - \lambda_{71} + \lambda_{76} - \lambda_{77} + \lambda_{78} - \lambda_{79} + \lambda_{80}a - \lambda_{81}a = 0$
27. $\lambda_{72} - \lambda_{73} - \lambda_{76} + \lambda_{77} + \lambda_{80}b - \lambda_{81}b = 1$
28. $\lambda_{74} - \lambda_{75} - \lambda_{78} + \lambda_{79} + \lambda_{80}c - \lambda_{81}c = 0$
29. $-\lambda_{80} + \lambda_{81} = -1$
30. $\lambda_{69} + 3\lambda_{71} + 3\lambda_{73} + 3\lambda_{75} + \lambda_{80}e - \lambda_{81}e = d$
31. $\lambda_{83} - \lambda_{84} + \lambda_{89} - \lambda_{90} + \lambda_{91} - \lambda_{92} + \lambda_{93}a - \lambda_{94}a = 0$
32. $\lambda_{85} - \lambda_{86} - \lambda_{89} + \lambda_{90} + \lambda_{93}b - \lambda_{94}b = 0$
33. $\lambda_{87} - \lambda_{88} - \lambda_{91} + \lambda_{92} + \lambda_{93}c - \lambda_{94}c = \Delta_{00}$
34. $-\lambda_{93} + \lambda_{94} = -\Delta_{01}$
35. $\lambda_{82} + 3\lambda_{84} + 3\lambda_{86} + 3\lambda_{88} + \lambda_{93}e - \lambda_{94}e = \Delta_{02}$
36. $\lambda_{96} - \lambda_{97} + \lambda_{102} - \lambda_{103} + \lambda_{104} - \lambda_{105} + \lambda_{106}a - \lambda_{105}a = 0$
37. $\lambda_{98} - \lambda_{99} - \lambda_{102} + \lambda_{103} + \lambda_{106}b - \lambda_{107}b = 0$
38. $\lambda_{100} - \lambda_{101} - \lambda_{104} + \lambda_{105} + \lambda_{106}c - \lambda_{107}c = \Delta_{10}$
39. $-\lambda_{106} + \lambda_{107} = -\Delta_{11}$
40. $\lambda_{95} + 3\lambda_{97} + 3\lambda_{99} + 3\lambda_{101} + \lambda_{106}e - \lambda_{107}e = \Delta_{12}$

Now the running example gets a shape of a mixed-integer non-linear programming (MINLP) optimization problem. It has forty equality constraints, in which ten are linear, and thirty non-linear constraints. The problem has 119 unknown variables, of which 108 are lambda variables (Farkas multipliers). These variables are real with 0 as the lower bound and 1 as the upper bound. Six Δ variables are integer in nature with a lower bound of negative infinity and an upper bound of positive infinity. The a , b , c , and e are real variables and ranges from negative infinity to positive infinity.

The d is an integer variable which is an objective function of the problem as well. It's lower bound is 0 and the upper bound is 3, as we have four processors and memories in the underlying hardware. As the problem contains real, and integer along with the linear and non-linear constraints, so it is a MINLP problem.

3.0.4 Optimization Tool and Solver Selection

We are using MATLAB language with an add-on of OPTI ToolBox[Currie and Wilson, 2012]. OPTI provides the solvers compiled with MEX (MATLAB executable) interfaces. This allows a MATLAB user to call them just like any other MATLAB function.

There are a couple of MINLP non-commercial/academic solvers available, like Basic Open Source Nonlinear Mixed Integer Programming (BONMIN), Nonlinear Optimization With Mesh Adaptive Direct Search (NOMAD), and Solving Constraint Integer Programs (SCIP)[Gamrath *et al.*, 2020]. We tried our MINLP problems over all of these solvers and found SCIP is the finest.

3.0.5 Results

By computing the running optimization problem over SCIP solver, we got the following output

$$\begin{aligned} \Delta_{00} &= -1, & \Delta_{01} &= 1, & \Delta_{02} &= 0 \\ \Delta_{10} &= 1, & \Delta_{11} &= -1, & \Delta_{12} &= 0 \end{aligned}$$

whereas our delta relation is

$$\Delta_{00}l + \Delta_{01}m + \Delta_{02} \geq 0$$

$$\Delta_{10}l + \Delta_{11}m + \Delta_{12} \geq 0$$

By substituting these delta values, we get

$$(-1)l + (1)m + 0 \geq 0 \longrightarrow -l + m \geq 0$$

$$(1)l + (-1)m + 0 \geq 0 \longrightarrow l - m \geq 0$$

This output implies that $l = m$, which means each program location is placed to the corresponding memory, like l_0 to memory node m_0 , l_1 to m_1 , l_2 to m_2 , and l_3 to m_3 . So each processor will get its required data from its local memory. Hence, the maximum distance data needs to travel to reach the processing units is 0. And, the solver gave us the objective function (d) value 0, which is expected according to the placement of data to the memories. And this output of the objective function is optimal.

Furthermore, the output of the function $f(i, p, l)$, which indicates which memory should be used for fetches is as follows

$$a = 0, \quad b = 1, \quad c = 0, \quad e = 0$$

By substituting these values in the constraints of affine function f :

$$al + bp + cl + e - m \geq 0 \rightarrow (0)l + (1)p + (0)l + 0 - m \geq 0 \rightarrow p - m \geq 0$$

$$-al - bp - cl - e + m \geq 0 \rightarrow -(0)l - (1)p - (0)l - 0 + m \geq 0 \rightarrow -p + m \geq 0$$

This output shows that $m = p$ and p ranges from 0 to 3, which satisfies the second constraint of the problem as well.

3.0.6 Example 2

Let us consider another example in which the optimal distance will be 1.

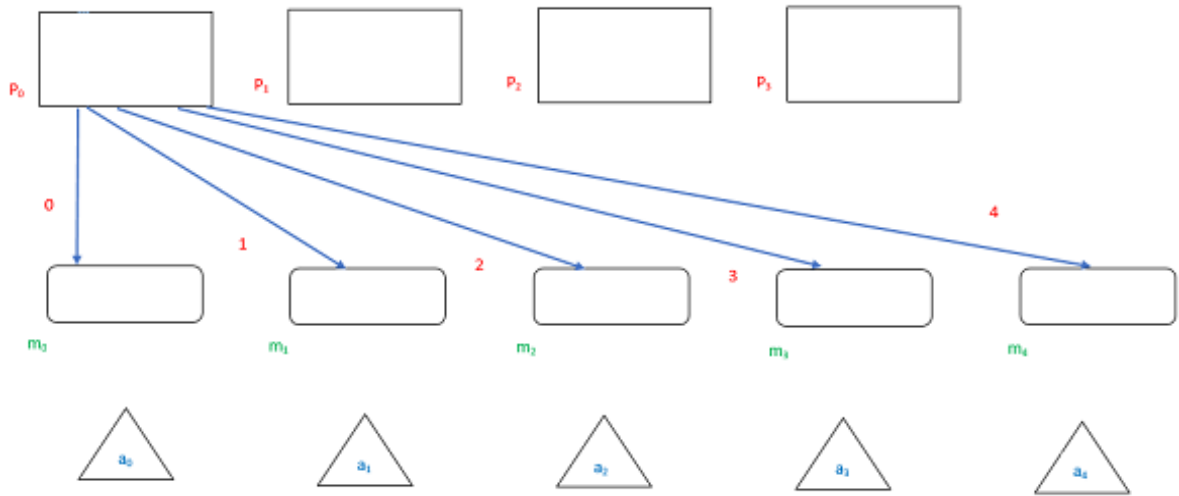


Figure 3.1: NUMA Machine 2

I(0): $a[0] := a[1]$
 I(1): $a[1] := a[2]$
 I(2): $a[2] := a[3]$
 I(3): $a[3] := a[4]$

There are four instances, and each instance is accessing two program locations. Each instance is storing to a same index number of program location and fetching from the very next program location. Suppose we have a linear hardware of four processors and five memories. In the Figure 3.1 processors, memories, and program locations are annotated with their indexes. For the example hardware, the distances range from 0 to 4. The outgoing arrows from processor p_0 to all memories indicate the cost of distance, which increases as the remote distance increases. The other three processors exhibit the same kind of behaviour concerning the cost of the distance. For parallel execution, we place one instance on each processor at the same time. The following table shows the placement of the instances on processors and program

locations access pattern of the instances of the running problem.

I(0) a[0] := a[1] on processor 0 fetch location 1 store location 0

I(1) a[1] := a[2] on processor 1 fetch location 2 store location 1

I(2) a[2] := a[3] on processor 2 fetch location 3 store location 2

I(3) a[3] := a[4] on processor 3 fetch location 4 store location 3

3.0.6.0 Representation

The algebraic representation of this problem sets (I, P, L, M) , instance placement to the processor's relation; $\Pi(i, p)$, and data store relation; $\Sigma(i, l)$ is given below respectively:

$$\iota \geq 0 \quad \text{and} \quad -\iota + 3 \geq 0$$

$$p \geq 0 \quad \text{and} \quad -p + 3 \geq 0$$

$$l \geq 0 \quad \text{and} \quad -l + 4 \geq 0$$

$$m \geq 0 \quad \text{and} \quad -m + 4 \geq 0$$

$$\iota - p \geq 0 \quad \text{and} \quad -\iota + p \geq 0$$

$$\iota - l \geq 0 \quad \text{and} \quad -\iota + l \geq 0$$

The data fetch relation for this example is different from the previous example, which is shown by the following inequalities.

$$\iota - l + 1 \geq 0 \quad \text{and} \quad -\iota + l - 1 \geq 0$$

By applying Farkas lemma on both problem constraints, we get a quantifier-free problem in the form of forty equality constraints. As this problem is the same in nature as the previous problem we are getting the same number of constraints. For a compound problem, we will get more constraints and consequently a greater number of equality constraints than forty. We will see it in the next example.

3.0.6.1 Implementation and Results

We have computed this quantifier-free problem on MATLAB with a SCIP solver. The output of the delta matrix is:

$$\Delta = \begin{bmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \end{bmatrix}$$

By substituting these delta values, we get

$$\begin{aligned} (-1)l + (1)m + 0 &\geq 0 \longrightarrow -l + m \geq 0 \\ (1)l + (-1)m + 0 &\geq 0 \longrightarrow l - m \geq 0 \end{aligned}$$

This shows that $l = m$, and each program location is placed in the corresponding memory node as shown in the Figure 3.2

It means each processor is storing data to its local memory and fetching data from very next memory. Due to this reason data has to travel a distance of 1 to reach the processor. The arrows going out and coming into the processors are annotated with the kind of access and the cost of the distance; the left side of the arrow is illustrating the kind of data access (store or fetch), and the right side of the arrows tells about the cost of the distance (0 or 1). For this example, the maximum distance a data has to travel is 1, and we got the same output from the solver/optimizer, $d = 1$. And this is the optimal output we can get.

3.0.7.0 Polyhedral Representation

There are eight instances and program locations in the problem. So from program text and available hardware, we can represent the set of instances, processors, program locations, and memories in algebraic form as follows

$$\begin{aligned} \iota &\geq 0 \quad \text{and} \quad -\iota + 7 \geq 0 \\ p &\geq 0 \quad \text{and} \quad -p + 7 \geq 0 \\ l &\geq 0 \quad \text{and} \quad -l + 7 \geq 0 \\ m &\geq 0 \quad \text{and} \quad -m + 7 \geq 0 \end{aligned}$$

As the example hardware has eight processors and the running problem has eight instances as well, so we can run each instance to the corresponding processors simultaneously. The following table shows the placement of the instances on processors and program locations access pattern of the instances of the running problem.

I(0)	$a[4] := a[4] + 1$	on processor 0	fetch location 4	store location 4
I(1)	$a[5] := a[5] + 1$	on processor 1	fetch location 5	store location 5
I(2)	$a[6] := a[6] + 1$	on processor 2	fetch location 6	store location 6
I(3)	$a[7] := a[7] + 1$	on processor 3	fetch location 7	store location 7
I(4)	$a[0] := a[0] + 1$	on processor 4	fetch location 0	store location 0
I(5)	$a[1] := a[1] + 1$	on processor 5	fetch location 1	store location 1
I(6)	$a[2] := a[2] + 1$	on processor 6	fetch location 2	store location 2
I(7)	$a[3] := a[3] + 1$	on processor 7	fetch location 3	store location 3

The $\Pi(\iota, p)$ relation between instances and processors is shown at the bottom part of Figure 3.3, and represented with the following inequalities:

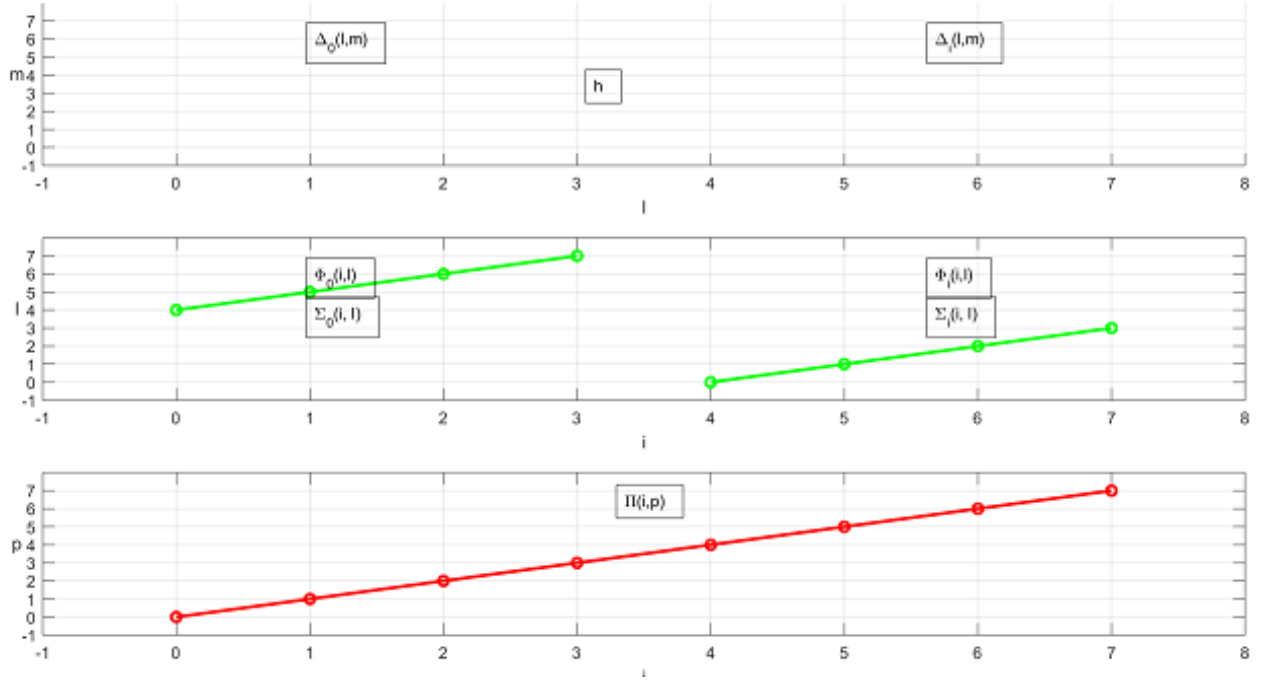


Figure 3.3: Rotation Problem

$$\iota - p \geq 0 \quad \text{and} \quad -\iota + p \geq 0$$

However, we can not represent the store and fetch behaviour of the running problem with one polyhedral relation as shown in the middle part of the Figure 3.3. The first four instances; ι_0 to ι_3 , are storing/fetching to/from program locations l_4 to l_7 . Whereas the next four instances; ι_4 to ι_7 , are fetching/storing from/to program locations l_0 to l_3 . Therefore, the $\Sigma(\iota, l)$ relation is decomposed into two polyhedral relations, $\Sigma(\iota, l) = \Sigma_0(\iota, l) \cup \Sigma_1(\iota, l)$. Likewise the $\Phi(\iota, l)$ relation is break down into two polyhedral relations, $\Phi(\iota, l) = \Phi_0(\iota, l) \cup \Phi_1(\iota, l)$.

$$\Sigma_0(\iota, l) : \iota - l + 4 \geq 0 \quad \text{and} \quad -\iota + l - 4 \geq 0$$

$$\Sigma_1(\iota, l) : \iota - l - 4 \geq 0 \quad \text{and} \quad -\iota + l + 4 \geq 0$$

$$\Phi_0(\iota, l) : \iota - l + 4 \geq 0 \quad \text{and} \quad -\iota + l - 4 \geq 0$$

$$\Phi_1(\iota, l) : \iota - l - 4 \geq 0 \quad \text{and} \quad -\iota + l + 4 \geq 0$$

Similarly we need to decompose the $\Delta(l, m)$ relation into two polyhedral relations, $\Delta(l, m) = \Delta 0(l, m) \cup \Delta 1(l, m)$ as shown at the top part of the Figure 3.3, and relation inequalities are given below

$$\Delta 0(l, m) :$$

$$\Delta 0_{00}l + \Delta 0_{01}m + \Delta 0_{02} \geq 0$$

$$\Delta 0_{10}l + \Delta 0_{11}m + \Delta 0_{12} \geq 0$$

$$\Delta 1(l, m) :$$

$$\Delta 1_{00}l + \Delta 1_{01}m + \Delta 1_{02} \geq 0$$

$$\Delta 1_{10}l + \Delta 1_{11}m + \Delta 1_{12} \geq 0$$

Hence, we need to find twelve unknown Δ integer variables for this problem. We also need two functions that demonstrate which memory should be used for fetches: f_0, f_1 .

$$f_0 : m = a_0\iota + b_0p + c_0l + e_0$$

$$f_1 : m = a_1\iota + b_1p + c_1l + e_1$$

Ultimately both of these functions will turn into two inequalities each for the sake of Farkas lemma implementation. Due to the nature of the problem, we also need a hyperplane h , which is unknown and decide where $\Delta_0(l, m)$ and f_0 should be used and where $\Delta_1(l, m)$ and f_1 should be used. The hyperplane h is represented by the following two inequalities:

$$-h_0\iota - h_1p - h_2l - h_3 \geq 0$$

$$h_0\iota + h_1p + h_2l + h_3 \geq 0$$

where $h_0, h_1, h_2,$ and h_3 are unknown real variables.

3.0.7.1 Problem Constraints

Distances of stores can not exceed d :

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge (\Sigma_0(\iota, l) \vee \Sigma_1(\iota, l)) \wedge (\Delta_0(l, m) \vee \Delta_1(l, m)) \Rightarrow |p-m| \leq d \quad (3.12)$$

For each read there is a memory no farther away than d , and f_0, f_1 indicate which memory that is:

$$\exists h. \forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. (-h_0\iota - h_1p - h_2l - h_3 \geq 0 \vee h_0\iota + h_1p + h_2l + h_3 \geq 0) \Rightarrow$$

$$(\forall m \in \mathbb{R}. (m = a_0\iota + b_0p + c_0l + e_0 \vee m = a_1\iota + b_1p + c_1l + e_1)) \wedge \Pi(\iota, p) \wedge$$

$$(\Phi_0(\iota, l) \vee \Phi_1(\iota, l)) \Rightarrow m \in \mathcal{M} \wedge |p - m| \leq d \wedge (\Delta_0(l, m) \vee \Delta_1(l, m)) \quad (3.13)$$

Subconstraints The running problem constraints are further divided into subconstraints, four for constraint 3.12 and four for constraint 3.13 as well. The subconstraints of constraint 3.12 are as follows:

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma_0(\iota, l) \wedge \Delta_0(l, m) \Rightarrow |p - m| \leq d \quad (3.14)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma_0(\iota, l) \wedge \Delta_1(l, m) \Rightarrow |p - m| \leq d \quad (3.15)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma_1(\iota, l) \wedge \Delta_0(l, m) \Rightarrow |p - m| \leq d \quad (3.16)$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma_1(\iota, l) \wedge \Delta_1(l, m) \Rightarrow |p - m| \leq d \quad (3.17)$$

And the subconstraints of constraint 3.13 are given below:

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. -h_0\iota - h_1p - h_2l - h_3 \geq 0 \Rightarrow$$

$$(\forall m \in \mathbb{R}. (m = a_0\iota + b_0p + c_0l + e_0 \wedge \Pi(\iota, p) \wedge \Phi_0(\iota, l) \Rightarrow$$

$$m \in \mathcal{M} \wedge |p - m| \leq d \wedge \Delta_0(l, m)) \tag{3.18}$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. -h_0\iota - h_1p - h_2l - h_3 \geq 0 \Rightarrow$$

$$(\forall m \in \mathbb{R}. (m = a_0\iota + b_0p + c_0l + e_0 \wedge \Pi(\iota, p) \wedge \Phi_1(\iota, l) \Rightarrow$$

$$m \in \mathcal{M} \wedge |p - m| \leq d \wedge \Delta_0(l, m)) \tag{3.19}$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. h_0\iota + h_1p + h_2l + h_3 \geq 0 \Rightarrow$$

$$(\forall m \in \mathbb{R}. (m = a_1\iota + b_1p + c_1l + e_1 \wedge \Pi(\iota, p) \wedge \Phi_0(\iota, l) \Rightarrow$$

$$m \in \mathcal{M} \wedge |p - m| \leq d \wedge \Delta_1(l, m)) \tag{3.20}$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. h_0 \iota + h_1 p + h_2 l + h_3 \geq 0 \Rightarrow$$

$$(\forall m \in \mathbb{R}. (m = a_0 \iota + b_0 p + c_0 l + e_0 \wedge \Pi(\iota, p) \wedge \Phi_1(\iota, l) \Rightarrow$$

$$m \in \mathcal{M} \wedge |p - m| \leq d \wedge \Delta_1(l, m)) \tag{3.21}$$

3.0.7.2 Farkas Lemma Implementation

As the running problem constraints contain universal and existential quantifiers, so we need to apply the Farkas lemma to get a quantifier-free problem. To use this lemma, the subconstraints of the problem will be further divided into subsubconstraints depending on the number of functions on the right side of the implication of the constraint. Constraints 3.14 to 3.17 are further divided into two sub-constraints for each. And constraints 3.18 to 3.21 will be further divided into six constraints for each. So, we get thirty-two (32) constraints in total. Let us apply Farkas lemma on constraint 3.14, which gives the following two subconstraints:

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma_0(\iota, l) \wedge \Delta_0(l, m) \Rightarrow p - m + d \geq 0 \tag{3.22}$$

$$\forall \iota \in \mathcal{I}. \forall p \in \mathcal{P}. \forall l \in \mathcal{L}. \forall m \in \mathcal{M}. \Pi(\iota, p) \wedge \Sigma_0(\iota, l) \wedge \Delta_0(l, m) \Rightarrow -p + m + d \geq 0 \tag{3.23}$$

The left-hand side of 3.14 can be represented by a polyhedron, let us call it D_{sa} .

$$\mathbf{A}_{sa} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & \Delta 0_{00} & \Delta 0_{01} \\ 0 & 0 & \Delta 0_{10} & \Delta 0_{11} \end{bmatrix} \quad \mathbf{b}_{sa} = \begin{pmatrix} 0 \\ 7 \\ 0 \\ 7 \\ 0 \\ 7 \\ 0 \\ 7 \\ 0 \\ 0 \\ 4 \\ -4 \\ \Delta 0_{01} \\ \Delta 0_{12} \end{pmatrix}$$

Now, by applying the Farkas lemma on constraint 3.22.

$$p + m + d = \lambda_0 + \sum_k \lambda_k ((\mathbf{A}_{sa})_k + (\mathbf{b}_{sa})_k)$$

The polyhedron D_{sa} has fourteen affine inequalities, so $p = 14$. By expanding the above formula we get

$$\begin{aligned} -p + m + d = & \lambda_0 + \lambda_1(\iota) + \lambda_2(-\iota + 7) + \lambda_3(p) + \lambda_5(-p + 7) + \lambda_6(l) + \lambda_7(-l + 7) + \\ & \lambda_7(m) + \lambda_8(-m + 7) + \lambda_9(\iota - p) + \lambda_{10}(-\iota + p) + \lambda_{11}(\iota - l + 4) + \\ & \lambda_{12}(-\iota + l - 4) + \lambda_{13}(\Delta 0_{00}l + \Delta 0_{01}m + \Delta 0_{02}) + \end{aligned}$$

$$\lambda_{14}(\Delta 0_{10}l + \Delta 0_{11}m + \Delta 0_{12})$$

By applying distributive property

$$\begin{aligned} -p + m + d = & \lambda_0 + \lambda_1\iota - \lambda_2\iota + 7\lambda_2 + \lambda_3p - \lambda_5p + 7\lambda_5 + \lambda_6l - \lambda_7l + 7\lambda_7 + \lambda_7m - \\ & \lambda_8m + 7\lambda_8 + \lambda_9\iota - \lambda_9p - \lambda_{10}\iota + \lambda_{10}p + \lambda_{11}\iota - \lambda_{11}l + 4\lambda_{11} - \lambda_{12}i + \lambda_{12}l - 4\lambda_{12} \\ & \lambda_{13}\Delta 0_{00}l + \lambda_{13}\Delta 0_{01}m + \lambda_{13}\Delta 0_{02} + \lambda_{14}\Delta 0_{10}l + \lambda_{14}\Delta 0_{11}m + \lambda_{14}\Delta 0_{12} \end{aligned}$$

By combining the like terms

$$\begin{aligned} -p + m + d = & \lambda_1\iota - \lambda_2\iota + \lambda_9\iota - \lambda_{10}\iota + \lambda_{11}\iota - \lambda_{12}\iota + \\ & \lambda_3p - \lambda_4p - \lambda_9p + \lambda_{10}p + \\ & \lambda_5l - \lambda_6l - \lambda_{11}l + \lambda_{12}l + \lambda_{13}l\Delta 0_{00} + \lambda_{14}l\Delta 0_{10} + \\ & \lambda_7m - \lambda_8m + \lambda_{13}m\Delta 0_{01} + \lambda_{14}m\Delta 0_{11} + \\ & \lambda_0 + 7\lambda_2 + 7\lambda_4 + 7\lambda_6 + 7\lambda_8 + 4\lambda_{11} - 4\lambda_{12} + \lambda_{13}\Delta 0_{02} + \lambda_{14}\Delta 0_{12} \end{aligned}$$

By factoring out i , p , l , and m respectively

$$\begin{aligned} -p + m + d = & \iota(\lambda_1 - \lambda_2 + \lambda_9 - \lambda_{10} + \lambda_{11} - \lambda_{12}) + \\ & p(\lambda_3 - \lambda_4 - \lambda_9 + \lambda_{10}) + \\ & l(\lambda_5 - \lambda_6 - \lambda_{11} + \lambda_{12} + \lambda_{13}\Delta 0_{00} + \lambda_{14}\Delta 0_{10}) + \\ & m(\lambda_7 - \lambda_8 + \lambda_{13}\Delta 0_{01} + \lambda_{14}\Delta 0_{11}) + \\ & \lambda_0 + 7\lambda_2 + 7\lambda_4 + 7\lambda_6 + 7\lambda_8 + 4\lambda_{11} - 4\lambda_{12} + \lambda_{13}\Delta 0_{02} + \lambda_{14}\Delta 0_{12} \end{aligned}$$

Now, by equating coefficients of i , p , l , m , and constant terms of both sides of the formula, we will get five following equations as the output of Farkas lemma implementation.

1. $\lambda_1 - \lambda_2 + \lambda_9 - \lambda_{10} + \lambda_{11} - \lambda_{12} = 0$
2. $\lambda_3 - \lambda_4 - \lambda_9 + \lambda_{10} = -1$
3. $\lambda_5 - \lambda_6 - \lambda_{11} + \lambda_{12} + \lambda_{13}\Delta 0_{00} + \lambda_{14}\Delta 0_{10} = 0$
4. $\lambda_7 - \lambda_8 + \lambda_{13}\Delta 0_{01} + \lambda_{14}\Delta 0_{11} = 1$

$$5. \quad \lambda_0 + 3\lambda_2 + 3\lambda_4 + 3\lambda_6 + 3\lambda_8 + 4\lambda_{11} - 4\lambda_{12} + \lambda_{13}\Delta_{02} + \lambda_{14}\Delta_{12} = d$$

By applying this same procedure of the Farkas lemma to all of the remaining thirty-one (31) constraints of the problem, we get one hundred and sixty (160) equality constraints, five from each constraint. There are four hundred and fifty-six (456) lambda variables from λ_0 to λ_{455} with lower bound and upper bound 0 to 1 respectively. Twelve Δ variables are integers and range from negative infinity to positive infinity. And the variables $a_0, b_0, c_0, e_0, a_1, b_1, c_1, e_1, h_0, h_1, h_2,$ and h_3 are all real with bounds from negative infinity to positive infinity. The objective function d is an integer variable and ranges from 0 to 7. So, in total we have four hundred and eighty-one (481) unknown variables in the running problem.

We have tried the running rotation problem at SCIP solver, however, this MINLIP solver did not give us an optimal output. However, we have optimally solved the rotation problem for minimizing the maximum (running time) cost with a heuristic technique, simulated annealing, which is explained in the next chapter.

Chapter 4

Minimize the Maximum Cost

One of the main reasons for compiler optimization is to minimize a program's execution time. At the time of computation, a processor needs data to perform its task, and if the required data is readily available for the processor to use without any delay, it will take the minimum execution time. We are looking to provide in time or before time data to the execution units of NUMA machine and distributed computers. Data access time varies from node to node in NUMA machines, access from a local storage node is significantly less than from a remote memory node. And for distributed multicomputers [Norvell *et al.*, 2017], some storage nodes are simply not reachable by some processing nodes. So, the goal is to place data into the best memories according to the time-dependent need of the computations.

4.0 Temporal Data Placement (TDP)

In this regard, TDP [Norvell *et al.*, 2017][Sindhu *et al.*, 2019] contributes in two ways. First of all, TDP is a compile-time approach, and in the compiler-domain normally a

function is used to place data into memory. Whereas, in TDP, we use a relation rather than a function, which allows data replication. This data placement relation; known as Δ , permits program locations to be mapped to more (or less) than one memory. The implementation and outcomes of this change are described clearly in minimizing the maximum distance chapter. Secondly, we include time in the Δ relation along with program locations and memories. The addition of time in this relation is the novel research contribution that helps the optimizer to make a time-dependent data placement. Moreover, the mapping of locations to memories can change over time, which is the data migration.

Let us take an example to see the working of TDP. Suppose we have the following program code taking place on a NUMA machine with four nodes.

I(0): a[0] := a[0] + 1	I(4): a[4] := a[4] + 1
I(1): a[1] := a[1] + 1	I(5): a[5] := a[5] + 1
I(2): a[2] := a[2] + 1	I(6): a[6] := a[6] + 1
I(3): a[3] := a[3] + 1	I(7): a[7] := a[7] + 1

Suppose we want to execute at most one instance at each processor at each time. So, we can schedule the first four instances at the first time and the next four for the next time, as we have eight computations and four processors.

We represent the schedule with Θ relation that is between times and instances. This relation tells us that which instance will start its execution at which time, and we can write it as follows:

$$\Theta(\tau, \iota) : \quad \tau = \lfloor \frac{\iota}{4} \rfloor$$

As $\Theta(\tau, \iota)$, relation includes a floor function, so we need to transform it into a simple algebraic form, and that is

$$4\tau - \iota + 3 \geq 0 \quad \text{and} \quad -4\tau + \iota \geq 0$$

The $\Pi(\iota, p)$ relation places instances on processors, and we want to place one instance on each processor at each time. We can do it by using modulo operation

$$p = \iota \bmod 4$$

Since we have two times in the running problem, and so, we would need following two relations to deal with

$$\text{For time 0:} \quad \iota - p \geq 0 \quad \text{and} \quad -\iota + p \geq 0$$

$$\text{For time 1:} \quad \iota - p - 4 \geq 0 \quad \text{and} \quad -\iota + p + 4 \geq 0$$

However, representing this $\Pi(\iota, p)$ polyhedrally is awkward, so instead we are going to represent time-dependent placement of instances to the processors with the join of $\Theta(\tau, \iota)$ and $\Pi(\iota, p)$ relations. By combining the $\Theta(\tau, \iota)$ and $\Pi(\iota, p)$, we get a composite relation $(\Theta \bowtie \Pi)(\tau, \iota, p)$, which has a concise polygonal representation, and that is:

$$4\tau - \iota + p \geq 0 \quad \text{and} \quad -4\tau + \iota - p \geq 0$$

Each instance is fetching from and storing to the same program location. The fetch relation $\Phi(i, l)$ and store relation $\Sigma(i, l)$ are

$$\Phi(i, l) : i = l \equiv \quad i - l \geq 0 \quad \text{and} \quad -i + l \geq 0$$

$$\Sigma(i, l) : i = l \equiv \quad i - l \geq 0 \quad \text{and} \quad -i + l \geq 0$$

Recall that a data placement relation is between program locations, times, and memories. We represent it as a matrix of two rows at least. The number of rows can be increased depending on the problem.

$\Delta(\tau, l, m) \equiv M_\Delta[\tau, l, m, 1]^T \geq 0$, where M_Δ is

$$\begin{bmatrix} \Delta_{00} & \Delta_{01} & \Delta_{02} & \Delta_{03} \\ \Delta_{10} & \Delta_{11} & \Delta_{12} & \Delta_{13} \end{bmatrix}$$

which give us the following two inequalities:

$$\Delta_{00} \times \tau + \Delta_{01} \times l + \Delta_{02} \times m + \Delta_{03} \geq 0$$

$$\Delta_{10} \times \tau + \Delta_{11} \times l + \Delta_{12} \times m + \Delta_{13} \geq 0$$

4.0.0 Minimize the Maximum Cost Procedure

We want to minimize the total time of execution by placing data in the proximity of the computations. The total time is the summation of time to fetch data, time to do computations, and time to store the results. We represent the data access times of a given processor p from a given memory m with $\delta_{fetch}(p, m)$ for fetch, and $\delta_{store}(p, m)$ for the store. We assume $\delta_{fetch}(p, m) = \delta_{store}(p, m) = 1$, if p and m have the same index. Otherwise the cost will be 2. We can calculate the total cost of stores on processor p at time τ by:

$$s(\tau, p) = \sum_{\substack{\iota \in \mathcal{I} \\ l \in \mathcal{L} \\ m \in \mathcal{M}}} \delta_{store}(p, m) \mid \Theta(\tau, \iota) \wedge \Pi(l, p) \wedge \Sigma(l, l) \wedge \Delta(l, \tau, m)$$

It is possible to place more than one instance at a processor in a single time period, that is why we are summing over all the instances in the above given formula. Similarly, we are summing over all locations that are stored during that particular time period. And if a given program location is mapped to more than one memory, then we must store to all memories in which that location is represented. Therefore, we are summing over all memories as well. However, for the running problem each processor is computing only one instance in one time period.

The total cost of fetches on processor p at time τ is:

$$f(\tau, p) = \sum_{\substack{\iota \in \mathcal{I} \\ l \in \mathcal{L}}} \min_{m \in \mathcal{M}} \delta_{store}(p, m) \mid \Theta(\tau, \iota) \wedge \Pi(l, p) \wedge \Sigma(l, l) \wedge \Delta(l, \tau', m)$$

where τ' is the time after τ . In case of data replication, we can fetch a given location from any memory in which that is placed. However, the optimizer should fetch from nearest memory to get a minimum cost. For the computation time, $\delta_{comp}(l, p)$ says how much computation time an instance ι takes on processor p . The total cost of computations on processor p at time τ is:

$$c(\tau, p) = \sum_{\iota \in \mathcal{I}} \delta_{comp}(\iota, p) \mid \Theta(\tau, \iota) \wedge \Pi(l, p)$$

In one time period, a processor can execute more than one instance, therefore, we are summing over all instances if there exist. Now, we can calculate the total running time of a given problem by summing over all the times with the following formula:

$$\sum_{\tau \in \mathcal{T}} \max_{p \in \mathcal{P}} (f(\tau, p) + c(\tau, p) + s(\tau, p))$$

For the running problem, there are four instances executing at one time on four different processors. Each processor may have a different total cost (fetch cost + compute cost + store cost), however, for each time we will pick the cost of that processor, which consumes maximum time among all four processors.

4.0.1 Problem Formalization

We can now formalize minimum maximum cost problem as:

Find Δ to minimize $\text{time}(\Delta)$ where:

$$\text{time}(\Delta) = \sum_{\tau \in \mathcal{T}} \text{time}(\Delta, \tau)$$

$$\text{time}(\Delta, \tau) = \max_{p \in \mathcal{P}} \text{time}(\Delta, \tau, p)$$

$$\text{time}(\Delta, \tau, p) = \sum_{\iota \in \mathcal{I} \times (\iota, \tau) \wedge \circ(\iota, p)} \text{fetchTime}(\Delta, \tau, p, \iota) + \text{computeTime}(\iota, p) + \text{storeTime}(\Delta, \tau, p, \iota)$$

$$\text{fetchTime}(\Delta, \tau, p, \iota) = \sum_{\iota | \Phi(\iota, \iota)} \min_{m | \Delta(\iota, \tau, m)} \delta_{\text{fetch}}(p, m)$$

$$\text{computeTime}(\iota, p) = \delta_{\text{comp}}(\iota, p)$$

$$\text{storeTime}(\Delta, \tau, p, \iota) = \sum_{\iota | \Sigma(\iota, \iota)} \sum_{m | \Delta(\iota, \tau+1, m)} \delta_{\text{store}}(p, m)$$

We are assuming that the computeTime is constant over all ι and p and therefore we can simplify by assuming that it is 0.

4.0.2 Optimization Problem Formalization

We formulate the minimum maximum cost problem as an optimization problem with the following known and unknown variables:

- Known: The set of times, instances, processors, program locations, and memories are respectively:

$$\mathcal{T} = \{0, 1\}$$

$$\mathcal{I} = \{0, 1, \dots, 7\}$$

$$\mathcal{P} = \{0, 1, 2, 3\}$$

$$\mathcal{L} = \{0, 1, \dots, 7\}$$

$$\mathcal{M} = \{0, 1, 2, 3\}$$

The schedule Θ , computation placement Π , fetch Φ and store Σ relations. The fetch $\delta_{fetch}(p, m)$ and store $\delta_{store}(p, m)$ access times.

- Unknown: $\Delta : \mathcal{L} \leftrightarrow \mathcal{T} \leftrightarrow \mathcal{M}$ relation: a matrix of size 2×4 (at least) comprises eight unknown delta variables.
- Cost Determination Formulae: The total costs of fetch, compute and store on processor p at time τ are:

$$f(\tau, p) = \sum_{\substack{\iota \in \mathcal{I} \\ l \in \mathcal{L}}} \min_{m \in \mathcal{M}} \delta_{store}(p, m) \mid \Theta(\tau, \iota) \wedge \Pi(\iota, p) \wedge \Sigma(\iota, l) \wedge \Delta(l, \tau', m)$$

$$c(\tau, p) = \sum_{\iota \in \mathcal{I}} \delta_{comp}(\iota, p) \mid \Theta(\tau, \iota) \wedge \Pi(\iota, p)$$

$$s(\tau, p) = \sum_{\substack{\iota \in \mathcal{I} \\ l \in \mathcal{L} \\ m \in \mathcal{M}}} \delta_{store}(p, m) \mid \Theta(\tau, \iota) \wedge \Pi(\iota, p) \wedge \Sigma(\iota, l) \wedge \Delta(l, \tau, m)$$

- Objective Function: We want to minimize the total running time, that is

$$\sum_{\tau \in \mathcal{T}} \max_{p \in \mathcal{P}} (f(\tau, p) + c(\tau, p) + s(\tau, p))$$

4.1 Optimization Methods

There are two types of optimization methods: exact methods and approximate methods. For many problems, exact methods require (or are thought to require) exponential time. The well-known examples of exact methods are enumeration methods using

cutting plane, branch and bound or dynamic programming techniques. We used SCIP (Solving Constraint Integer Programs) for solving minimize the maximum distance problems. SCIP is a branch-cut-and-price framework, and for the cost minimization problems, it is not yet possible to represent \sum , min, and max operation with the algebraic inequalities. Moreover, to optimally solve large-scale engineering and scientific problems instance by instance, an exact technique will take an unacceptable amount of time. For example, we tried to solve the rotation problem for minimizing the maximum distance between processors and memories (see section 3.0.7 on page 67) with an exact method (SCIP) but was not successful.

4.2 Approximate/Heuristic Methods

An approximate or a heuristic method is any problem-solving approach that uses a practical method and various shortcuts to get the output quickly. The output may not be optimal but can be satisfactory enough. Heuristic methods are flexible and used for such complex problems, for which it is not possible to get an optimal result or not suitable to use any exact method due to time constraints. Local search and randomization algorithms like Simulated Annealing (SA) are good examples of heuristic techniques. We are using SA for solving the minimum maximum cost problems.

4.2.0 Simulated Annealing (SA)

SA is a random search heuristic method, which is very effective for approximating global optimization problems. The source of SA is the physical annealing of solids. In the annealing process, a metallic solid is heated to a high temperature, and then

gradually cooled on a specific schedule. As the metal cools its atoms settle into an optimal crystalline structure. Kirkpatrick et al.[Kirkpatrick *et al.*, 1987] and V. Cerny [Černý, 1985] identify that the concept of annealing of solids can also work for the optimization of combinatorial problems. The different states of the solid correspond to the different solutions of an optimization problem. The energy of the system corresponds to the objective function which needs to be minimized. SA is derived from the Monto Carlo Method proposed by Metropolis et al.[Metropolis *et al.*, 1953][pjm van laarhoven, 1988].

- Given a current state of solid, characterized by the position of its particles, a small, randomly generated, perturbation is applied by a small displacement of a randomly chosen particle.
- If Energy (E): current state fitness value – new state fitness value < 0 , then the process is continued with new state.
- If $E \geq 0$, then the probability of acceptance of new state depends on $\exp(-E/kT)$, where $\exp(-E/kT)$ is also known as Metropolis Criterion, T is the control parameter and k is the Boltzmann Constant.

SA has the following properties: iterative improvement, local random search, exploration, and exploitation.

4.2.0.0 Iterative improvement

Suppose we are given a finite solution space of the problem, the cost function and a mechanism to generate the neighbours. We will start with a random initial solution

and consider it as the current solution. Run a fixed loop and during each iteration of the loop generate a neighbour, and compare it with the current solution. If we find that the neighbour solution is better than the current, replace it with the neighbour, otherwise, keep the current solution. The disadvantage of the iterative improvement mechanism is that it can be stuck into a local optimum and has no way to escape from it.

4.2.0.1 Local random search

SA is a local search-based approach, trying to find a better solution in the immediate neighbours of the current solution rather than sampling throughout the whole solution space.

4.2.0.2 Exploration

Recall in iterative improvement the algorithm can be stuck into local optimum; however, SA can escape from it by accepting some worse solutions based on the following probabilistic acceptance function.

$$p(x', x, T) = \begin{cases} e^{-(f(x')-f(x))/T} & \text{if } f(x') - f(x) > 0 \\ 1 & \text{otherwise} \end{cases}$$

where x : current solution, x' : neighbour solution, T : Temperature.

At very high temperatures, SA accepts almost every worst solution which allows exploring the entire solution space. In other words, the algorithm behaves like a random walk. However, the frequency of accepting bad solutions gradually decreases as the temperature decreases.

4.2.0.3 Exploitation

At the later stages, when the temperature approaches 0, the algorithm does not accept any worst solution and starts behaving like a Hill-Climbing algorithm.

4.2.0.4 Algorithm

Generally, SA contains two loops. The outer loop controls the temperature by gradually decreasing it with a proportional function. Kirkpatrick et al proposed such a proportional function; $T(t + 1) = \alpha T(t)$, α can range from 0.8 to 0.99. The inner loop generates the neighbour solutions, evaluating the cost difference between two solutions, and decides either to move to the neighbour or not. Following is the SA algorithm based on [Kirkpatrick *et al.*, 1987][Černý, 1985][Eglese, 1990][Luke, 2012].

- $x =$ Initial Solution
- T assigns with a very high temperature
- Outer Loop (Fixed number of times or Depends on criteria like $T < 0.01$)
- Inner Loop (Normally fixed number of times)

x' = generate random neighbor of x

Evaluate Cost Difference: $f(x') - f(x)$

Move with the Probability Acceptance Function

$$p(x', x, T) = \begin{cases} e^{-(f(x')-f(x))/T} & \text{if } f(x') - f(x) > 0 \\ 1 & \text{otherwise} \end{cases}$$

- $T = \alpha T$

4.2.1 SA Implementation

For the running problem, according to $\Theta(\tau, \iota)$ and $\Pi(\iota, p)$ relation, each processor is executing its same index number instance during the first time period:

$$I(0): \quad a[0] := a[0] + 1$$

$$I(1): \quad a[1] := a[1] + 1$$

$$I(2): \quad a[2] := a[2] + 1$$

$$I(3): \quad a[3] := a[3] + 1$$

Processor p_0 executes instance ι_0 . Instance ι_0 fetches and stores to location l_0 . Memory m_0 is local for processor p_0 , and if Δ relation can place l_0 at m_0 , then p_0 will get all necessary data from its local memory and complete its execution in 2: 1 for fetching the required program location and 1 to store data back to the memory after computation. Therefore, the total processing cost of processor p_0 will be 2 for first time period.

To explain the working of functions $s(\tau, p)$ and $f(\tau, p)$, suppose for time 0, Δ relation mapped program location l_0 at memory m_0 , l_1 at m_1 , l_3 at m_3 , and l_2 at two memories: m_2, m_3 . The execution cost of processors p_0, p_1 , and p_3 will be 2 as shown in the Figure 4.0. Since all of these processors are getting and storing data from/to their local memories, moreover, l_0, l_1 , and l_3 program locations are represented in only one memory. However, location l_2 is represented in two memories and needed for processor p_2 . Processor p_2 can get l_2 from any of the memories where it is represented. Suppose, p_2 gets data from its local memory m_2 , then the cost of fetch will be 1 as shown in the Figure 4.0. However, p_2 needs to store l_2 in both

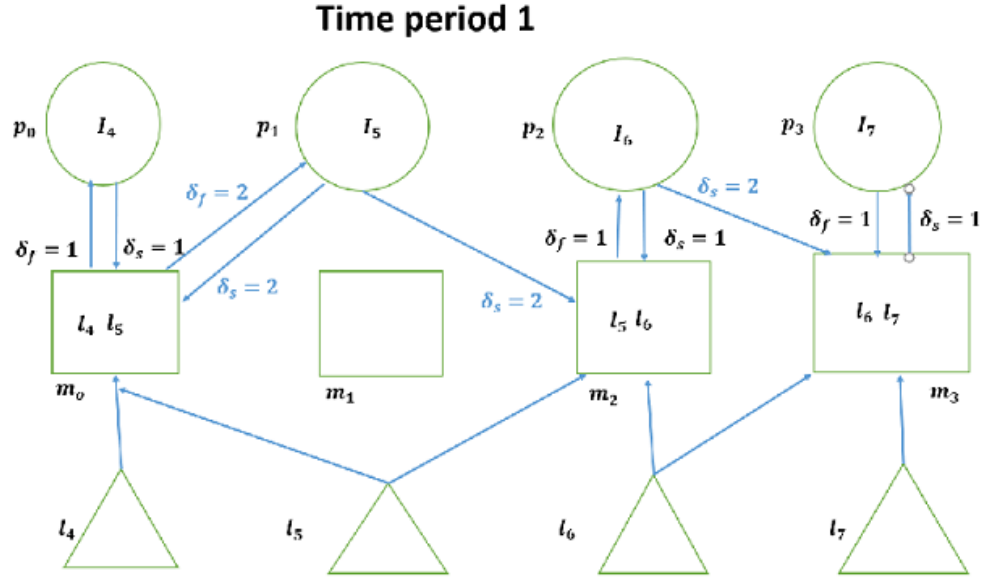


Figure 4.1: Data Placement for Time 1

$$I(6): \quad a[6] = a[6] + 1$$

$$I(7): \quad a[7] = a[7] + 1$$

are going to be executed in time period 1. $\Pi(l, p)$ relation maps instance i_4 at processor p_0 , i_5 at p_2 , i_6 at p_3 , and i_7 at p_3 . Suppose data placement for time 1 is as follows:

Locations l_4 and l_5 at memory m_0 .

No data placement at memory m_1 .

Locations l_5 and l_6 at memory m_2 .

Locations l_6 and l_7 at memory m_3 .

This data placement is shown in the Figure 4.1.

From the figure we can see that the processors p_0 and p_3 execution cost will be 2 due to the availability of program locations l_4 , and l_7 in their local memories, and these

locations are not represented in multiple memories as well. However, the program locations l_5 and l_6 are replicated in multiple memories. Processor p_2 needs program location l_6 that is represented in memories m_2 and m_3 . Its cost can be varied from 4 to 5 depending on the fetch access from memory m_2 or m_3 . The processor p_1 needed program location l_5 is not available in its local memory, rather it is represented in remote memories m_0 and m_2 . So, the processor p_1 cost will be 6: 4 for storing into two remote memories, and 2 for fetching data from a remote memory. The impact of cost on processor p_1 can be viewed clearly from the Figure 4.1, in which remote costs are annotated with blue colour. Hence, for time period 1, the objective function will choose 6 as it is maximum cost among all four processors. Therefore, the total execution cost of the running problem will be 10: 4 for time period 0, and 6 for time period 1.

From the above discussion, we can easily understand the impact of good data placement for improving the performance of computing machines. What would be the optimal data placement for the running problem? Of course, there should be such data placement that every processor gets its required data from its local memory. We have run this problem with the Simulated Annealing algorithm and get the following output for the $\Delta(\tau, l, m)$ relation:

$$\begin{bmatrix} 4 & -1 & 1 & 0 \\ -4 & 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} \tau \\ l \\ m \\ 1 \end{bmatrix} \geq 0$$

According to this relation, the data placement for time 0 is

Location l_0 is placed at memory m_0 .

Location l_1 is placed at memory m_1 .

Location l_2 is placed at memory m_2 .

Location l_3 is placed at memory m_3 .

and for time 1:

Location l_4 is placed at memory m_0 .

Location l_5 is placed at memory m_1 .

Location l_6 is placed at memory m_2 .

Location l_7 is placed at memory m_3 .

And this is the optimal data placement we can have for the running problem. Every processor gets data from its local memory, and each time maximum processor cost is 2. Consequently, the total execution cost of the running problem is 4 which is optimal.

Let us consider another example in which multiple processors need the same data at the same time.

I(0): $a[0] = a[0] + a[1]$

I(1): $a[1] = a[1] + a[2]$

I(2): $a[2] = a[2] + a[3]$

I(3): $a[3] = a[3] + a[4]$

Suppose we want to compute this program code on a NUMA machine of four nodes. There are four instances and suppose we assign one corresponding instance at each processor simultaneously. Every processor needs two consecutive program locations, and each two adjacent processors require one identical program location. For example, both processors p_0 and p_1 need program location l_1 at the same time. Processor p_0 will fetch l_1 while processor p_1 will store the same program location l_1 .

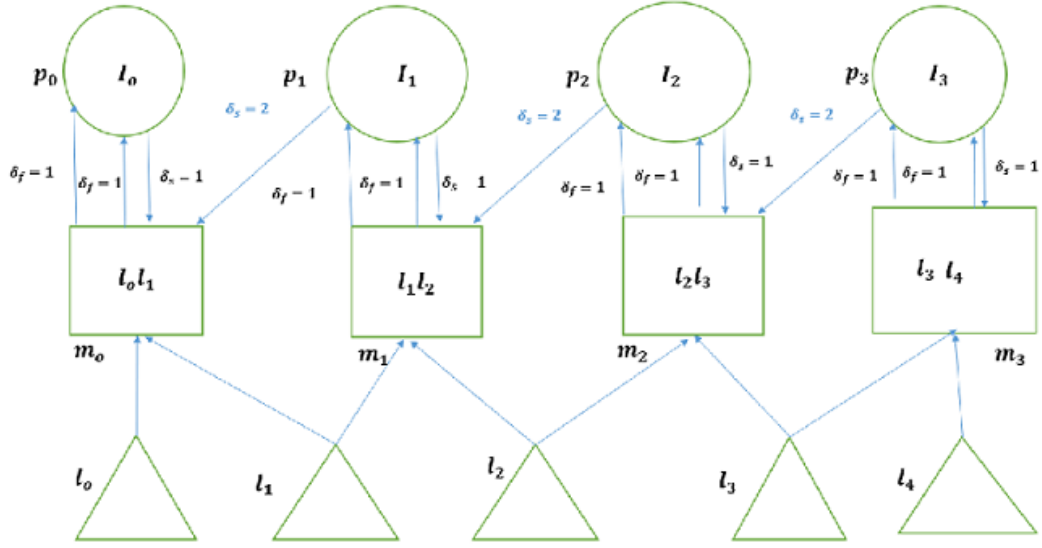


Figure 4.2: Presumed Data Placement

Suppose the Δ relation maps each processor's needed data at its local memory. As a result each memory will hold two consecutive program locations as shown in the Figure 4.2.

With this data placement each processor will store its output in two memories except p_0 , and total running time cost of this problem will be 5 as shown in the Figure 4.2. But the question that arises is this the optimal total execution time? We have computed this problem with SA algorithm and obtained the following output Δ matrix:

$$\Delta(\tau, l, m) : \begin{bmatrix} 5 & 2 & -2 & 1 \\ 2 & -7 & 8 & 4 \end{bmatrix} \begin{bmatrix} \tau \\ l \\ m \\ 1 \end{bmatrix} \geq 0$$

According to this result the first three program locations $l_0, l_1,$ and l_2 will be

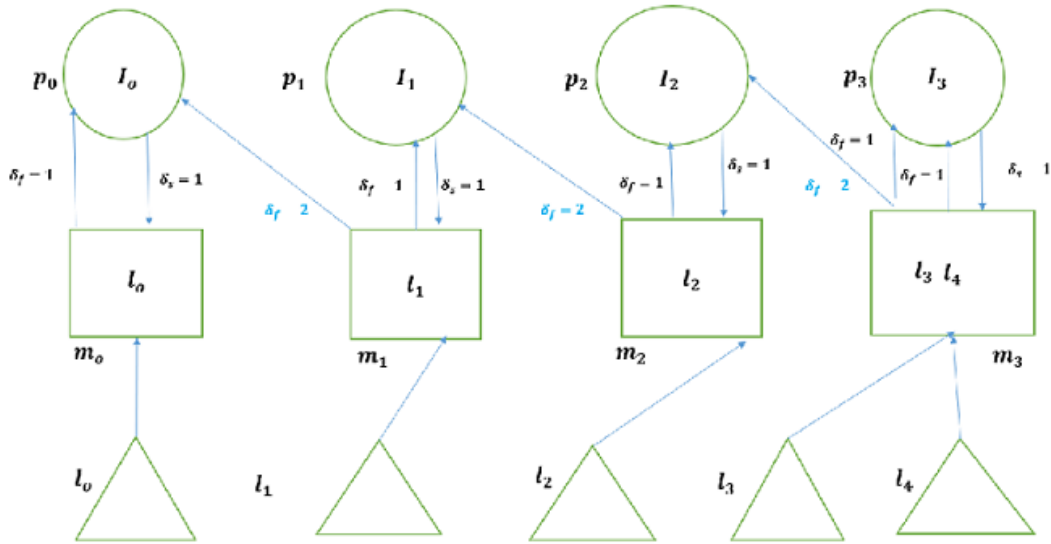


Figure 4.3: Optimal Data Placement

placed in the corresponding memories. Therefore, memories m_0 , m_1 , and m_2 will contain only one program location. And the last two program locations l_3 and l_4 will be mapped at memory m_3 . Hence no processor needs to store its result in multiple memories. Consequently, we get an optimal total running time for this problem and that is 4 instead of 5 as illustrated in the Figure 4.3.

4.2.1.0 Compound Polyhedra Example

Let us consider the following rotation problem for minimizing the maximum running cost.

$$I(0): \quad a[4] := a[4] + 1$$

$$I(1): \quad a[5] := a[5] + 1$$

$$I(2): \quad a[6] := a[6] + 1$$

$$I(3): \quad a[7] := a[7] + 1$$

$$I(4): \quad a[0] := a[0] + 1$$

$$I(5): \quad a[1] := a[1] + 1$$

$$I(6): \quad a[2] := a[2] + 1$$

$$I(7): \quad a[3] := a[3] + 1$$

The problem has eight instances and eight program locations. Suppose we have a linear NUMA architecture of four processors and memories pairs, and we want to execute one instance at each processor at each time. Thus, we need two times to complete the execution of this problem. The following table depicts the scheduling and placement of the instances on processors with respect to times.

I(0)	$a[4] := a[4] + 1$	on processor 0	at time 0
I(1)	$a[5] := a[5] + 1$	on processor 1	at time 0
I(2)	$a[6] := a[6] + 1$	on processor 2	at time 0
I(3)	$a[7] := a[7] + 1$	on processor 3	at time 0
I(4)	$a[0] := a[0] + 1$	on processor 0	at time 1
I(5)	$a[1] := a[1] + 1$	on processor 1	at time 1
I(6)	$a[2] := a[2] + 1$	on processor 2	at time 1
I(7)	$a[3] := a[3] + 1$	on processor 3	at time 1

Polyhedral Representation The scheduling and placement of the instances on processors of example hardware can be made with a combined relation $(\Theta \bowtie \Pi)(\tau, \iota, p)$, and represented by the following inequalities:

$$4\tau - i + p \geq 0 \quad \text{and} \quad -4\tau + i - p \geq 0$$

Processor p_0 executes instance ι_0 for the first time and ι_4 for the second time. Instances ι_0 and ι_4 fetch and store to locations l_0 and l_4 , respectively. Similarly, all other instances access their corresponding index locations for fetch and store. The fetch relation $\Phi(i, l)$ and store relation $\Sigma(i, l)$ are

$$\Phi(i, l) : (i < 4 \text{ and } l = i + 4) \text{ or } (i \geq 4 \text{ and } l = i - 4)$$

$$\Sigma(i, l) : (i < 4 \text{ and } l = i + 4) \text{ or } (i \geq 4 \text{ and } l = i - 4)$$

The unknown $\Delta(\tau, l, m)$ relation places program locations to the memory banks of the example hardware. Due to the nature of the problem, this relation decomposes into two polyhedral relations:

$$\Delta(\tau, l, m) = \Delta 0(\tau, l, m) \cup \Delta 1(\tau, l, m)$$

For the first four instances, ι_0 to ι_3 , $\Delta 0(\tau, l, m)$ gives the information about which locations map in which memories, whereas $\Delta 1(\tau, l, m)$ tells where the program locations will be placed in memory banks accessed by the instances ι_4 to ι_7 .

$$\Delta 0(\tau, l, m) :$$

$$\begin{bmatrix} \Delta 0_{00} & \Delta 0_{01} & \Delta 0_{02} & \Delta 0_{03} \\ \Delta 0_{10} & \Delta 0_{11} & \Delta 0_{12} & \Delta 0_{13} \end{bmatrix} \begin{bmatrix} \tau \\ l \\ m \\ 1 \end{bmatrix} \geq 0$$

$$\Delta 1(\tau, l, m) :$$

$$\begin{bmatrix} \Delta 1_{00} & \Delta 1_{01} & \Delta 1_{02} & \Delta 1_{03} \\ \Delta 1_{10} & \Delta 1_{11} & \Delta 1_{12} & \Delta 1_{13} \end{bmatrix} \begin{bmatrix} \tau \\ l \\ m \\ 1 \end{bmatrix} \geq 0$$

Results The output of the simulated annealing algorithm for $\Delta 0(\tau, l, m)$ is

$$\Delta 0(\tau, l, m) : \begin{bmatrix} 2 & 1 & -1 & -4 \\ -4 & -1 & 1 & 4 \end{bmatrix} \begin{bmatrix} \tau \\ l \\ m \\ 1 \end{bmatrix} \geq 0$$

According to this result, the program locations accessed by the instances ι_0 to ι_3 for the first time period are placed in memories as follows:

Location l_4 is placed at memory m_0 .

Location l_5 is placed at memory m_1 .

Location l_6 is placed at memory m_2 .

Location l_7 is placed at memory m_3 .

And, the output of $\Delta 1(\tau, l, m)$ relation by the algorithm is

$$\Delta 1(\tau, l, m) : \begin{bmatrix} 0 & 2 & -2 & 1 \\ 4 & -3 & 4 & -4 \end{bmatrix} \begin{bmatrix} \tau \\ l \\ m \\ 1 \end{bmatrix} \geq 0$$

For time period 1, $\Delta 1(\tau, l, m)$ relation placed relevant program locations for instances ι_4 to ι_7 into the memories as follows:

Location l_0 is placed at memory m_0 .

Location l_1 is placed at memory m_1 .

Location l_2 is placed at memory m_2 .

Location l_3 is placed at memory m_3 .

Due to the optimal data placement by the optimizer, every processor of example hardware get required data from its local memory. Furthermore, we get an optimal total running time of this problem, which is 4: 2 for the first time and 2 for the second time.

The optimized results for all example problems show that the TDP is an efficient approach for modern parallel hardware to optimize the total execution time of the problems with the help of $\Delta(\tau, l, m)$ relation, that is capable to map program locations into the nearest memories of the processors.

We believe that for problems whose all sets, and relations can be represented by polygons, an exact method, Mixed-Integer Nonlinear Programming (MINLP) can be used to determine the optimal data placement. However, the problems which are based on the union of polyhedral, or contains \sum , max, and min operations, a heuristic method Simulated Annealing can be used to find an optimal (closer to optimal) data placement to minimize the maximum running time for a subset of problems in a considerable amount of time.

Chapter 5

Conclusion and Future Work

To get the best performance from any computing system, we need to supply data to the execution units in the minimum possible time. An intelligent data placement from secondary memory to the main memory can help us in this regard. The decision of data mapping to the memory places can be made at three different times: compile-time, load-time and run-time. The compile-time data placement to memory locations is the first step to enhance the performance of a computing system. However, present-day parallel hardware poses a challenge for a compiler: how to make an efficient data placement in the presence of multiple memory units in a single machine like a NUMA system, or to the several memory units of a distributed computer?

To utilize the power of parallel hardware a problem should be divided and mapped onto multiple processors. Once the computations are mapped then we need to place the data in the proximity to optimize the total running time. The polyhedral framework provides an algebraic procedure to schedule and place computations at multiple processors. However, parallel execution of a program on parallel hardware creates

more challenges for efficient data placement. Because multiple processors may need the same data at the same time, we need to replicate that data in multiple memories depending on the needs of the processing units. Moreover, data may need to migrate from one memory to another memory (memories) during execution. A natural intuition arises, why not place whole data of a program in all memories? A simple answer is that unnecessary placement of data at various memory nodes will increase computation time. So, how can we schedule and place data where it is needed and when it is needed? The Temporal Data Placement approach provides a solution for this problem.

TDP provides a systematic way to place data where it is needed and when it is needed. The use of a relation instead of a function for data placement provides a way to replicate data in multiple memories if needed. Furthermore, the inclusion of the time dimension in $\Delta : \mathcal{L} \leftrightarrow \mathcal{T} \leftrightarrow \mathcal{M}$ relation allows the mapping of data to memories to change over time. The data can migrate from one memory to another memory (memories) according to time dependent needs of the execution units. Based on the results after applying TDP for a couple of examples, we can conclude that for a subset of data placement optimization problems our approach could minimize the maximum distance between processors and memories, and also minimize the maximum cost (running time) of computation for NUMA systems and distributed computes.

The key new concept here is to schedule data in addition to placing it. The Delta relation establishes a relationship between program-level data items and physical places, as well as time. This allows data to be replicated and to move around within a system. TDP considers the network as one of the places of storage. Allowing a network to be one of the places where data is replicated, we can easily transform

a shared memory algorithm into distributed algorithms. Cache prefetching, NUMA architecture optimization and distributed architecture optimization are some of the applications.

To optimize the execution times of computer programs we are using a mathematical representation of programs instead of a syntactic representation. By representing programs algebraically, we can work instance-wise rather than statement-wise which significantly help to expedite the overall computational performance. Our approach TDP is based on the polyhedral model, and applicable for both shared-memory multiprocessor systems: NUMA architectures, and distributed memory multiprocessors computers. In the polyhedral framework, reasonable research has been carried out to minimize the maximum distance between processors and shared last level cache. Whereas, the TDP objective is to target wide categories of modern parallel hardware because there exist many such computing machines which do not have cache in their architecture. To minimize the maximum distance between processors and memories of parallel hardware, we used a mathematical optimization method Mixed-Integer Nonlinear Programming (MINLP). The problem constraints (see (3.0.0) on page 49) contains universal (\forall) and existential (\exists) quantifiers. To eliminate these quantifiers we used the Farkas lemma [Feautrier, 1992], which transforms a system of inequalities with quantifiers into a system of affine equalities by adding Farkas multipliers; non-negative variables that range from 0 to 1. After transforming the minimize the maximum distance problems into affine equalities we computed these problems on an MINLP solver SCIP [Gamrath *et al.*, 2020] and obtained the optimized distance between processors and memories. The minimize the maximum cost (running time) problem (see (4.0.1) on page 83) contains \sum , max, and min operation. However,

until now it is not possible to represent these operations algebraically. Therefore, we used a heuristic technique Simulated Annealing (SA) to solve minimize the maximum total running times problems and achieved the optimized total running time.

In this thesis, we have proposed a viable nonparametric approach to minimize the maximum distance between processors and memories and minimize the maximum cost (total running time) of programs for modern parallel hardware. In the future, we want to expand our research for parametric case of said problems. We believe that temporal data placement may be automated, allowing optimizing compilers to modify programs automatically. Additionally in the future, we will combine multiple actions of our optimization process into one step. For example, in this thesis, the scheduling of the computation, the computation placement, and the temporal placement of data are performed in separate steps. The objective is to perform the scheduling of computations and placement of the computations in one step rather than two. Moreover, the temporal placement of data could be done in the same step as the placement of computations.

References

- [Allen and Kennedy, 2001] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, 2001.
- [Baillie, 1988] Clive F. Baillie. Comparing shared and distributed memory computers. *Parallel Computing*, 8(1):101–110, 1988. Proceedings of the International Conference on Vector and Parallel Processors in Computational Science III.
- [Bastoul, 2004a] Bastoul. Improving data locality in static control programs. 2004.
- [Bastoul, 2004b] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 7–16. IEEE, 2004.
- [Bastoul, 2012] Cédric Bastoul. *Contributions to high-level program optimization*. PhD thesis, Habilitation Thesis. Paris-Sud University, France, 2012.
- [Benabderrahmane *et al.*, 2010] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

- [Blagodurov *et al.*, 2010] Sergey Blagodurov, Alexandra Fedorova, Sergey Zhuravlev, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 557–558. IEEE, 2010.
- [Bondhugula *et al.*, 2008] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.
- [Bunday, 1984] B.D. Bunday. *Basic Linear Programming*. E. Arnold, 1984.
- [Černý, 1985] Vladimír Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.
- [Chen *et al.*, 2011] D.S. Chen, R.G. Batson, and Y. Dang. *Applied Integer Programming: Modeling and Solution*. Wiley, 2011.
- [Currie and Wilson, 2012] Jonathan Currie and David I. Wilson. OPTI: Lowering the Barrier Between Open Source Optimizers and the Industrial MATLAB User. In Nick Sahinidis and Jose Pinto, editors, *Foundations of Computer-Aided Process Operations*, Savannah, Georgia, USA, 8–11 January 2012.
- [Daellenbach and Hans, 1970] Earl R. Daellenbach and G. Hans. *User’s Guide to Linear Programming*. Prentice-Hall, 1970.

- [Dashti *et al.*, 2013] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.
- [Eglese, 1990] Richard W Eglese. Simulated annealing: a tool for operational research. *European journal of operational research*, 46(3):271–281, 1990.
- [Eskicioglu and Marsland, 1996] M. Rasit Eskicioglu and T. Anthony Marsland. Distributed shared memory: A review. techreport 96-22, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, September 1996.
- [Feautrier, 1988] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988.
- [Feautrier, 1992] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.
- [Feautrier, 2013] Paul Feautrier. Mathematical bases of the polyhedral model, 2013. On the web at <http://perso.ens-lyon.fr/paul.feautrier/cartoon.pdf>.
- [Gamrath *et al.*, 2020] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Weikun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter

- Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020.
- [Gaud *et al.*, 2015] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern numa systems. *Communications of the ACM*, 58(12):59–66, 2015.
- [Kirkpatrick *et al.*, 1987] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. In *Readings in Computer Vision*, pages 606–615. Elsevier, 1987.
- [Lu *et al.*, 2009] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, Ponnuswamy Sadayappan, Yongjian Chen, Haibo Lin, et al. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 348–357. IEEE, 2009.
- [Luke, 2012] S. Luke. *Essentials of Metaheuristics (Second Edition)*. Lulu.com, 2012.
- [Majo and Gross, 2011] Zoltan Majo and Thomas R Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, pages 1–10, 2011.
- [Majo and Gross, 2015] Zoltan Majo and Thomas R Gross. A library for portable and composable data locality optimizations for numa systems. *ACM SIGPLAN Notices*, 50(8):227–238, 2015.

- [Metropolis *et al.*, 1953] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.
- [Norvell *et al.*, 2017] Theodore S Norvell, Umair Sindhu, and Adrian Fiech. Temporal data placement: a first look. In *Newfoundland Electrical and Computer Engineering Conference (NECEC)*, 2017.
- [pjm van laarhoven, 1988] pjv van laarhoven. *Simulated Annealing:theory & Application*. kluwer academic, 1988.
- [Pouchet, 2010] Louis-Noël Pouchet. Polyhedral compilation foundations, 2010. From the web at <http://web.cs.ucla.edu/~pouchet/lectures>.
- [Sindhu *et al.*, 2019] Umair Sindhu, Theodore S Norvell, and Adrian Fiech. An efficient data placement approach for parallel systems. In *Newfoundland Electrical and Computer Engineering Conference (NECEC)*, 2019.
- [Ste, 2018] Symmetric multiprocessor architecture. In Thomas Sterling, Matthew Anderson, and Maciej Brodowicz, editors, *High Performance Computing*, pages 623–638. Morgan Kaufmann, Boston, 2018.
- [Susungi *et al.*, 2017] Adilla Susungi, Albert Cohen, and Claude Tadonki. More data locality for static control programs on numa architectures. In *Proceedings of the 7th International Workshop on Polyhedral Compilation Techniques (IMPACT’17)*, 2017.

- [Weisstein, a] Eric W Weisstein. Affine functions. From MathWorld—A Wolfram Web Resource. <https://mathworld.wolfram.com/AffineFunctions.html>.
- [Weisstein, b] Eric W Weisstein. Vector space. From MathWorld—A Wolfram Web Resource. <https://mathworld.wolfram.com/VectorSpace.html>.
- [Zhang *et al.*, 2011] Yuanrui Zhang, Wei Ding, Jun Liu, and Mahmut Kandemir. Optimizing data layouts for parallel computation on multicores. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 143–154. IEEE, 2011.
- [Zima and Chapman, 1993] Hans P Zima and Barbara M Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, 1993.