2017

Analysis and Optimization of Scientific Applications through Set and Relation Abstractions

M. Tohid (Rastegar Tohid, Mohammed)
Louisiana State University and Agricultural and Mechanical College

ANALYSIS AND OPTIMIZATION OF SCIENTIFIC APPLICATIONS THROUGH SET AND RELATION ABSTRACTIONS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Electrical Engineering and Computer Science

by
Mohammad Rastegar Tohid
M.Sc., Newcastle University, 2008
B.Sc., Azad University (S. Tehran Campus), 2006
May 2017

Dedication

Dedicated to my parents who have dedicated their lives to their children.

Acknowledgments

I thank my advisor Dr. Ramanujam for valueing independent research and allowing me to work on topics that geniunely intrigued me while never lettig me go off the rail. His optimism and words of encouragement always aspired me. It has been a great pleasure to have known and worked with him.

I would like to thank Dr. Paul Kelly at Imperial College for supporting me during my visit. I respect his exemplary passion to create an environment for collaboration and sharing knowledge among peers. I also would like to thank Fabio Luporini for intersting discussions we had on many topics both in person and over the Skype.

Many thanks to Carlo (Dr. Bertolli at IBM) for introducing me to the research topic which has inspired the work presented in this dissertation.

I would like to thank Sahar who has never stopped believing in, supporting me and helping me survive difficult times. I thank Vahid, for I also thank my other colleagues and friends, Sameer, for providing insights into software development, Ye, for showing interest in my research and commenting on my work and Zahra, for the

I thank my siblings, Reza and Raana, and their spouses, Mahta and Pascal, for their words of encouragement and setting great examples for me to follow.

Finally, I thank my parents. Thank you mom for your never ending love and support, and being my best friend all throught my life. Thank you dad for all your sacrifices and for teaching me how to think critically.

Contents

List of Tables

List of Figures

Abstract

Writing high performance code has steadily become more challenging since the design of computing systems has moved toward parallel processors in forms of multi and many-core architectures. This trend has resulted in exceedingly more heterogeneous architectures and programming models. Moreover, the prevalence of distributed systems, especially in fields relying on supercomputers, has caused the programming of such diverse environment more difficulties.

To mitigate such challenges, an assortment of tools and programming models have been introduced in the past decade or so. Some efforts focused on the characteristics of the code, such as polyhedral compilers (e.g. Pluto, PPCG, etc.) while others took in consideration the aspects of the application domain and proposed domain specific languages (DSLs). DSLs are developed either in the form of a stand-alone language, like Halide for image processing, or as a part of a general purpose language (e.g., Firedrake- a DSL embedded in Python for solving PDEs using FEM.) called embedded. All these approaches attempt to provide the best input to the underlying common programming models like MPI and OpenMP for distributed and shared memory systems respectively.

This dissertation introduces Kaashi, a high-level run-time system, embedded in C++ language, designed to manage memory and execution order of programs with large input data and complex dependencies. Kaashi provides a uniform front-end to multiple back-ends focusing on distributed systems.

Kaashi abstractions allows the programmer to define the problem's data domain as a collection of sets and relations between pairs of such sets. The aforesaid level of abstraction could enable series of optimizations which, otherwise, are very expensive to detect or not feasible at all. Furthermore, Kaashi's API helps novice programmers to write their code more structurally without getting involved in details of data management and communication.

Chapter 1
Introduction

This chapter reviews the current state of the high-performance computing to demonstrate the motivations and influences behind the research presented in following chapters. In the first part, we discuss challenges of today's heterogeneous HPC systems. Next, an overview of pertinent ideas and solutions available to tackle such challenges is provided. Finally, the objectives and contribution of the research conducted for the dissertation in hand is presented.

1.1 Challenges

1.1.1 Hardware Architecture

Since the turn of the millennium the focus of micro-architecture design has shifted toward multi- and many-core processors. This trend started in response to power dissipation and fault tolerance issues arisen from higher clock rates in smaller fabrication scales. Meanwhile, the number of transistor per unit area has grown according to Moore's Law which has enabled the processor architects to pack more resources into a single die. There have been two major consequences as a result of these developments:

1. deploying new hardwares will no longer automatically translates to faster execution of the software. Because the information required for parallelism above instruction level is not available to hardware.

2. HPC systems have progressively become more and more heterogeneous and, therefore, more difficult to program.

The following reviews two primary processor design strategies, namely, multi-core and many-core systems, and highlights their pros and cons.

- **Multi-core Processors**

  Multi-core architectures prioritize the performance of each core on the chip over the aggregate throughput of the all. Therefore, such processors have a few number of complex cores tuned to deliver maximum performance for a single task- a task usually consists of series of operations. To alleviate memory accesses latencies, CPUs allocate a significant portion of the die to multiple levels of on-chip, hardware-managed, cache. The multi-level cache design enables the processor to exploit *spatial* and *temporal* localities. Multi-core processors employ many intricate techniques such as pre-fetching and out-of-order execution to maximize the ILP (instruction level parallelism). Meanwhile, modern processors have attempted to improve the parallel processing capabilities by introducing *single instruction, multiple data* (SIMD) extension (i.e., vectorization).

  Performance Analyses. Multi-core architectures perform better in the presence of coarse-grained parallel data. Larger cache sizes and agile hardware-managed caches enable the programmer to optimize the code by exploiting spatial and temporal localities. One can improve spatial locality by storing data consumed by a thread in contiguous memory locations while larger caches would help boost the temporal locality in such processors. Moreover, high degree of ILP in the workload leads to high utilization of *Arithmetic and Logic Units* (ALUs), hence, better computational performance. Also, it is necessary to recourse to vector extensions (SIMD instructions) for attaining near peak performance.

- **Many-core Processors**

  As the name suggests, a many-core processor is an assembly of many simple cores which run in parallel to accommodate the throughput-driven design. Many-core processors usually act as accelerators where the parallel workload is offloaded. The CPU (a multi-core processor) often manages the flow of data to and from the accelerators as well as launching kernels on such devices.

Performance Analyses. Since most of the die on many-core processors is dedicated to processing cores in order to boost the computational throughput of the device, there is not much room for other resources such as registers and shared memory, i.e., $L1$ cache. These resources are shared among the resident threads of each multiprocessor and therefore are scarce. It is the programmer's responsibility to meticulously marshal the data between the shared and on-device memory to maximize the memory bandwidth utilization. Although, to alleviate the memory latency problem, context switching is managed by the hardware with no overhead, programmer still needs to ensure that access to on-device memory is coalesced. Avoiding divergence among the threads of a warp is another essential factor in increasing the effective computational throughput of the device.

To summarize, The number of cores are considerably lower in multi-processors while the clock frequency and cache sizes are much higher. On the other hand, beside having many more cores, many-core architectures provide better memory bandwidth in order to overcome limitations of the smaller caches and also providing more data for their cores. Table 1.1 nicely captures the heterogeneity of specifications and configurations of four state-of-the-art processors employed in many of TOP500 supercomputers.

Table 1.1: Examples of processors commonly employed in supercomputers to illustrate their diversity.

	Processor	**#Cores**	**Freq.** (MHz)	**Cache L1/L2/L3** (MB)	**Mem. BW** (GB/s)	**Power** (Watts)
Multi-core	Intel Xeon	18	2300	0.576/4.608/45	102	150
	IBM Power8	12	2750	0.768/6.144/96	230	250
Many-core	Intel Xeon Phi	61	1238	1.952/31.232/-	351	300
	Nvidia K80	2496	560	0.064/1.572/-	480	300

1.1.2 Memory

Interestingly enough, despite of all aforementioned difficulties in the advancement of processing units, the performance gap between the processor and memory has never been higher than today (figure 1.1). Imbalance between the computation time and memory latency has caused the memory bandwidth to become the bottleneck of many scientific applications. Techniques such as overlapping of the communication and computation, and pre-fetching have been introduced and implemented to minimize the effective latency of memory access (the delay between the data request and execution). Intuitively, eliminating the need to move the data is the ultimate solution to this issue, alas, impossible to entirely achieve. However, avoiding data movement, whenever possible, is almost always a favorable optimization that can be adopted in conjunction with any other optimization technique.

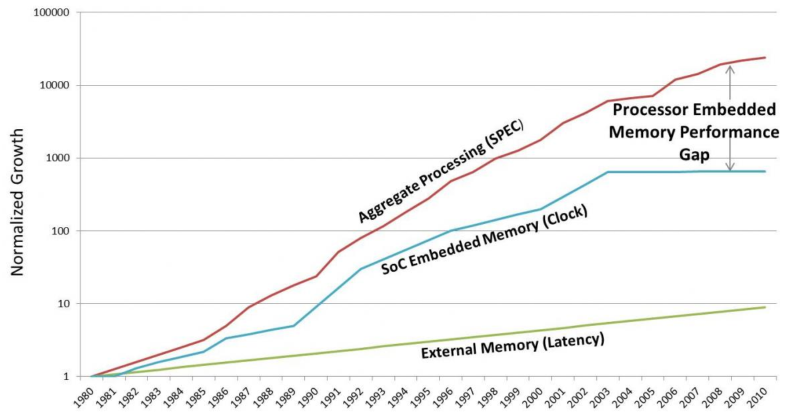


Figure 1.1: Performance gap between memory and processor. [Pat11]

1.2 Objectives and Outline

Work presented in this dissertation researches solutions for two classes of algorithms: (1) those aimed at optimizing memory management, mainly for distributed systems, and (2) optimizations geared toward maximizing data reuse and minimizing data movement via a run-time scheduler. These algorithms have been implemented in *Kaashi* run-time multi-stage scheduler with an embedded automatic data management system. Kaashi's API provides high-level abstractions to enable programmers conveniently define the domain of interest, usually a grid, through simple interface, and run a single code on range of processor architecture and configurations.

The rest of the dissertation is organized as follows. Chapter 2 provides the mathematical foundation as well as an overview of software Kaashi has upon which been built. Chapter 3 Introduces Kaashi and describes its memory management utilities in addition to the details of the scheduling algorithm used in clusters with NUMA configuration,i.e., inter-node techniques. In Chapter 4 we discuss the intra-core scheduling techniques and elaborate on benefits of hierarchical scheduling techniques used in Kaashi. Chapter 5 provides experimental results focusing on real world scientific problems through Kaashi. Finally, conclusions and future works are discussed in Chapter 6.

Chapter 2
Background

This chapter begins with a short review of binary relations, the mathematical foundation of this work, to familiarize the reader with the notations and terminologies used throughout the dissertation. Subsequently, we introduce the Zoltan library with a focus on functionalities which Kaashi relies upon.

2.1 Binary Relations

Augustus De Morgan, the prominent British mathematician, defines relations as follows:

> *"when two objects, qualities, classes, or attributes, viewed together by the mind, are seen under some connexion, that connexion is called a relation."*

2.1.1 Formal Definition

In general, a relation could be defined over any number of sets. However, in this work, we are solely interested in relations defined over two sets, i.e., binary relations. A binary relation between two sets A and B, denoted by R, is a collection of ordered pairs where the first element is an element of the set A and the second belongs to B. This collection is a subset of $A \times B$. Formally:

Definition 1. Binary Relation. A binary relation R between two arbitrary sets X and Y, denoted by xRy, is the graph G- a subset of the Cartesian product $X \times Y$. X is called the departure set and Y is the codomain.

It is worth mentioning that the relation is not commutative, that is:

$$xRy \not\Rightarrow yRx$$

This means elements' order in each pair of G must be respected, because, if $X \neq Y$, each of the relations xRy and yRx could hold, independent of the other.

Also, if two relations have the same graph, one cannot conclude that the relations are the same. As an example, let's consider $G = (1,3), (1,4), (2,3), (2,4)$ as a relation graph. We can define distinct relations $R1$ and $R2$, both with G, one from $\mathbb{N}$ to $\mathbb{Z}$ and the other from $\mathbb{Z}$ to $\mathbb{R}$.

2.1.2 Types of Relations

To function properly, Kaashi relies on certain properties which relations passed to it must carry. In this section we discuss such properties.

Definition 2. Injective Relation. A relation is injective if for all elements x and z in X, and y in Y, if a relation is defined between x and y as well as z and y, then x must be equal to z.

$$\forall x, y \in X \text{ and } y \in Y : \text{ if } xRy \text{ and } zRy \implies y = x$$

Definition 3. Functional Relation. A relation is functional if for all elements x in X, and y and z in Y, if a relation is defined between x and y as well as x and z, then y must be equal to z.

$$\forall x \in X \text{ and } y, z \in Y : \text{ if } xRy \text{ and } xRz \implies x = z$$

Definition 4. Left-Total Relation. Relation R is left-total if for any element x in X there exists y in Y such that xRy.

$$\forall x \in X, \exists\ y \in Y \ni xRy$$

Definition 5. Right-Total Relation. Relation R is right-total if for any element y in Y there exists x in X such that xRy.

$$\forall y \in Y,\ \exists\ x \in Y\ \ni\ xRy$$

2.2 Hypergraphs

We use hypergraphs to represent sets and relations among them. This section offers a brief introduction to this concept.

2.2.1 Definition

In mathematics, hypergraph structure is a generalization of the concept of graphs. While in graphs each edge could only have two adjacent vertices, there is no limit on the number of neighboring vertices' of an edge in a hypergraph. Formally, a hypergraph $\mathbf{H}$ is a pair $\mathbf{H} = (\mathbf{V}, \mathbf{N})$ where, $\mathbf{V}$ is the set of elements, i.e., vertices, and $\mathbf{N}$ *nets*, also know as *hyperedges*, between such vertices. Nets are collection of non-empty subsets of $\mathbf{V}$. Figure 2.1 illustrates an example of a hypergraph.

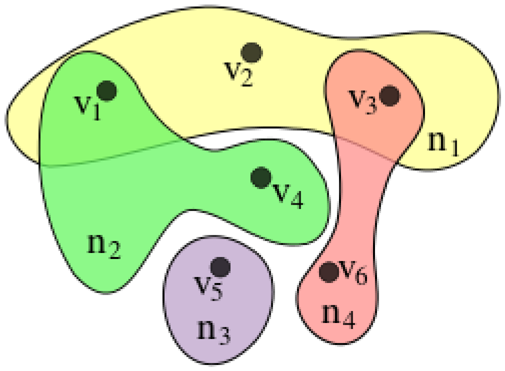


Figure 2.1: An example hypergraph.

In the example above, the set of vertices is $\mathbf{V} = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ and $\mathbf{N} = \{n_0, n_1, n_2, n_3\}$ is the hyperedges (nets) set. Where

$$n_0 = \{v_0, v_1, v_2\}$$

$$n_1 = \{v_0, v_3\}$$

$$n_2 = \{v_4\}$$

$$n_3 = \{v_2, v_5\}$$

One should note that a heyperedge may only be connected to one vertex, like n_2 in the last example.

2.3 Zoltan Library

The Zoltan library [DBR+09, DBH+02] is a toolkit of combinatorial algorithms. Zoltan provides a collection of highly optimized functionalities for load balancing and data management, e.g., distributed data directories, graph partitioning and graph coloring routines. We have decided to use Zoltan it is a powerful library with no dependency on any other library other than MPI. Also, Zoltan does not require any particular data structure. In lieu, Zoltan library employs callback function approach to acquire required data from the application. In this section we introduce a number of such utilities which have been employed by Kaashi.

Figure 2.2 shows a prototypical flow of an application built on top of the Zoltan library.

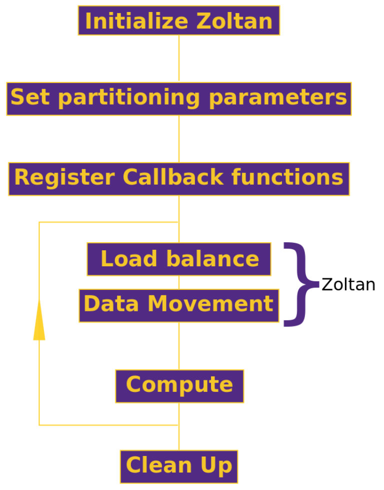


Figure 2.2: Program flow of a Zoltan application.

2.3.1 Load Balancing

Zoltan provides a wide range of partitioning algorithms, both native and third party, through a uniform interface. Well-established third party partitioning libraries, such as ParMETIS [Kar11] and SCOTCH [PR96], for graphs, and PaToH [ÇA11], for hypergraphs, are supported, however, these libraries should be installed independently. Listing 2.3 is the Zoltan's routine used for domain decomposition. In this routine, the argument *zz* is the pointer to the Zoltan structure invoked to manage the partitioning. Next is an output argument called *changes* which identifies whether or not the partitioning has changed by the invocation of the method. Arguments *num_gid_entries* and *num_lid_entries*, respectively, determine the number of entries describing a single global and local ID (These should be set to maximum value used among all the processes). Upon return, if the process needs to import objects from other processes, Zoltan sets variables *number_import* and *import_procs* accordingly and allocates arrays *import_golbal_ids* and *import_local_ids* with size of the number of imports and fills them with IDs of import objects. Similarly, the export information is provided by Zoltan.

```c
int Zoltan_LB_Balance(struct Zoltan_Struct *zz,
                      int *changes,
                      int *num_gid_entries,
                      int *num_lid_entries,
                      int *num_import,
                      unsigned *import_global_ids,
                      unsigned *import_local_ids,
                      int **import_procs,
                      int *num_export,
                      unsigned *export_global_ids,
                      unsigned *export_local_ids,
                      int **export_procs);
```

Figure 2.3: Zoltan's load balancing routine.

2.3.2 Distributed Directory

Zoltan's distributed directory is a distributed hash table with a pointer to the data held in the directory. A distributed design of this data management utility alleviates the memory and processing bottlenecks common in centralized designs. An object stored in the table is known by its global identification number (global ID), set by the application. Hashing provides very fast lookup for the information associated with a global ID in a two stage process using two hash algorithms. The first algorithm yields the rank of the owner of the ID. On the next step, a separate hash algorithm retrieves the associated data from the process' hash table. More information on Zoltan's hashing algorithms can be found in [HP01]

Listing 2.4 reviews some of the Zoltan's distributed directory routines. Before calling any other function, the routine *Zoltan_DD_Create* must be invoked to create an empty distributed directory. Subsequently, each object's identification code(s), namely, global ID, as well as other optional entries including, but not limited to, the local IDs, i.e., local indices, and associated data to the object are added to the directory through the invocation of *Zoltan_DD_Update*. In order to access the information already stored in the directory, one should call *Zoltan_DD_Find* which returns the directory information for the list of input global IDs. Finally, *Zoltan_DD_Destroy.* destroys the directory and returns the memory to the system.

2.3.3 Unstructured Communication

To manage data and also to schedule execution order in parallel, Kaashi performs myriad of communications across processes. Zoltan's unstructured communication functionality is one of the best candidates for conducting data movements as it has a straightforward interface for complicated point-to-point communication patterns. Moreover, the asynchronous versions of Zoltan_Comm_Do routine, provided as a part of the communication package, enables overlapping of the computation and communication.

In this section, we touch on functions used by Kaashi, listed in 2.5 - more information on this can be, found at [zol]

Similar to other Zoltan utilities, *Zoltan_Comm_Create* must first be called before any other routines in the package. Calling this function results in the allocation of a communication

```
int Zoltan_DD_Create(
    struct Zoltan_DD_Struct *
        *dd, // Structure maintains directory state and hash table.
    MPI_Comm
        comm, // MPI comm duplicated and stored specifying directory processors.
    int num_gid_entries, // Length (number of) of global IDs.
    int num_lid_entries, // Length of local IDs- zero.
    int user_length,     // Length (number of char) of user defined data field
                         // (optional, may be zero).
    int table_length,    // Length of hash table (0 for default, i.e., 100,000)
    int debug_level);    // [0,9].

int Zoltan_DD_Update(
    struct Zoltan_DD_Struct *
        *dd,        // Structure maintains directory state and hash table.
    unsigned *gid, // Input: List of global to update.
    unsigned *lid, // Input: List of corresponding local IDs (optional).
    char *data,    // Input: List of corresponding user data (optional).
    int *part,     // Input: List of corresponding parts (optional).
    int count);    // Number of global IDs in update list;

int Zoltan_DD_Find(
    struct Zoltan_DD_Struct
        *dd,        // Structure maintains directory state and hash table.
    unsigned *gid, // Input: List of global to update
    unsigned *lid, // Input: List of corresponding local IDs (optional).
    char *data,    // Input: List of corresponding user data (optional).
    int *part,     // Input: List of corresponding parts (optional).
    int count,     // Number of global IDs in update list;
    int *owner);   // Corresponding list of data owners (out).
```

Figure 2.4: Zoltan's distribured directory routines.

plan. In addition to the amount of data being transfered, the communication plan describes the processes to which data will be sent and / or received.

The Zoltan_Comm_Do function performs the communication described in the plan returned by Zoltan_Comm_Create. Based on the information extracted from the plan, Zoltan's communication copies data objects to a buffer and transfers data to the destination processors. Zoltan_Comm_Do also receives object data from other processors and copies it into the receive buffer.

For the cases where there are possibilities for overlapped communication and processing, Zoltan offers POST and WAIT variants of Zoltan_Comm_Do. In such instances communication is initiated by *Zoltan_Comm_Do_Post*; incoming messages are posted and outgoing messages are sent. Then the application can proceed with the computation. After the processing is complete, the corresponding *Zoltan_Comm_Do_Wait* must be invoked to wait for all incoming messages to complete receiving data. The Post and Wait routines take the same arguments as Zoltan_Comm_Do.

```
int Zoltan_Comm_Do(
    struct Zoltan_Comm_Obj
        *plan, // A pointer to a communication plan built by Zoltan_Comm_Create.
    int tag,   // An MPI message tag.
    char *send_data, // A buffer filled with object data to be sent to other
                     // processors.
    int nbytes, // The size (in bytes) of the data for one object, or the scale
                // factor if the objects have variable sizes.
    char *recvbuf // Upon return, a buffer filled with object data received from
                  // other processors.
    );

int Zoltan_Comm_Do_Post(
    struct Zoltan_Comm_Obj
        *plan, // A pointer to a communication plan built by Zoltan_Comm_Create.
    int tag,   // An MPI message tag.
    char *send_data, // A buffer filled with object data to be sent to other
                     // processors.
    int nbytes, // The size (in bytes) of the data for one object, or the scale
                // factor if the objects have variable sizes.
    char *recvbuf // Upon return, a buffer filled with object data received from
                  // other processors.
    );

int Zoltan_Comm_Do_Wait(
    struct Zoltan_Comm_Obj
        *plan, // A pointer to a communication plan built by Zoltan_Comm_Create.
    int tag,   // An MPI message tag.
    char *send_data, // A buffer filled with object data to be sent to other
                     // processors.
    int nbytes, // The size (in bytes) of the data for one object, or the scale
                // factor if the objects have variable sizes.
    char *recvbuf // Upon return, a buffer filled with object data received from
                  // other processors.
    );

int Zoltan_Comm_Invert_Map(
    int *lengths_to, // Input array with the number of values in each of the
                     // messages to be sent. Note that the actual size of each
                     // value is not specified until the Zoltan_Comm_Do routine
                     // is invoked.
    int *procs_to, // Input array with the destination processor for each of the
                   // messages to be sent.
    int nsends,    // Input argument with the number of messages to be sent.
                   // (Length of the lengths_to and procs_to arrays.)
    int self_msg,  // Input argument indicating whether a processor has data for
                   // itself (=1) or not (=0) within the procs_to and lengths_to
                   // arrays.
    int **
        lengths_from, // Returned array with lengths of messages to be received.
    int **procs_from, // Returned array of processors from which data will be
                      // received.
    int *nrecvs,      // Returned value with number of messages to be received
                      // (lengths of lengths_from and procs_from arrays).
    int my_proc,      // The ID of the processor in the comm communicator.
    int nprocs,       // Number of processors in the comm communicator.
    int out_of_mem,   // Since it has a barrier operation, this routine is a
                      // convenient time to tell all the processors that one of
                      // them is out of memory. This input argument is 0 if the
                      // processor is OK, and 1 if the processor has failed in a
                      // malloc call. All the processors will return with a code
                      // of COMM_MEMERR if any of them is out of memory.

    int tag,      // A message tag which can be used by this routine.
    MPI_Comm comm // MPI Communicator for the processor numbering in the procs
                  // arrays.
    );
```

Figure 2.5: Zoltan unstructured communication routines.

Supplementary to routines that carry the actual communication, Zoltan also provides the *Zoltan_Comm_Invert_Map*. This low level communication routine can be used when a process knows to whom it needs to send data, but not from who- and vice versa. To accomplish this, each process pass to this routine a set of lengths and destinations for the messages to be transferred. This routine enables Kaashi to pass information regarding the size and location of the data being transferred for the purpose of partitioning.

Chapter 3
Related Works

This chapter surveys the latest approaches to automatic code optimization. We explore tools for both regular (software with static control structure and affine memory accesses) and irregular programs. Some of these are developed as libraries while the others are *Domain-Specific Languages* (DSLs). First we study efforts for optimizations of applications over structured loops under the umbrella of polyhedral model. Subsequently, tools for boosting performance of codes over unstructured meshes are examined.

3.1 Regular Applications

Many tools [Gri04, LLL01, LCL99, GL96, HNS09] for optimizing applications with static control flow and affine memory access rely on polyhedral model. The polyhedral model provides a mathematical abstraction to represent dynamic instances of each statement as a lattice point within a polytope. To reason the correctness of loop transformations within the mathematical representation, dependence analysis of the code must also be present. Having this information, one can apply a collection of transformation, all within the polyhedral abstraction, to optimize a program.

There have been works employing polyhedral techniques in static and *just-in-time* (JIT) compilers as well as DSLs. The following are the widely used tools based on polyhedral model.

3.1.1 PLUTO

PLUTO [BHRS08] is a directive-based fully automatic source-to-source transformation framework which simultaneously optimizes regular programs for both parallelism and locality using polyhedral model. An optimization process is usually accomplished in three steps:

1. Static dependence analysis of the input program [Fea88, Fea91, Pug91].

2. Applying code transformations in the polyhedral abstraction.

3. Generation of the transformed code [GLW98, KPR95, Bas04].

Step 1 is implemented using Clan (*Chunky Loop ANalyzer*) [BCG+03] to generate polyhedral representation of *Static Control Parts* (SCoPs) of codes and Candl (*Chunky ANalyzer for Dependences in Loops*) [Bas07] for data dependency computations.
In step 2, PLUTO tries to find a transformation (classic transformations in code optimization sense is carried out by a single transformation function in polyhedral model) to generate coarse-grained parallel code which is communication- and locality-optimized. This is achieved through minimizing the following cost function:

$$\delta_e(\overrightarrow{s}, \overrightarrow{t}) = \Phi_{S_j}(\overrightarrow{t}) - \Phi_{S_i}(\overrightarrow{s}), \quad \langle \overrightarrow{s}, \overrightarrow{t} \rangle \in \rho_{e^{s_i \to s_j}} \tag{3.1}$$

where δ is the cost function, $\overrightarrow{s}$ and $\overrightarrow{t}$ are, respectively, source and target (transformed) iteration spaces, Φ_{S_i} is a one-dimensional affine transformation of statement S_i, ρ is the dependence polyhedron and $e^{s_i \to s_j}$ denotes a dependency edge from S_i to S_j.
PipLib (*The Parametric Integer Programming*) [Fea88] is used as the *Integer Linear Programming* (ILP) solver to find the lexicographic minimal solution to minimize equation 4.1. To minimize the cost function, PLUTO iteratively finds independent solutions for each statement. Solving the ILP finds the coefficients of the best mappings for a single statement (this means that the number of independent solutions must at least be as many as the dimensionality of the polytope associated with each statement).
Finally, ClooG [Bas04] is used to generate the transformed, optimized code. The affine functions, Φ , are called *scattering functions* in the specification of Cloog. Cloog can scan a union of polyhedra specified as scattering functions. Scattering functions are specified statement-wise, and the legality of scanning the polyhedron with these dimensions in the particular order should be guaranteed by the specifier – PLUTO in this case. The code

generator does not have any information on the dependences and hence, in the absence of any scattering functions would scan the union of the statement polyhedra in the global lexicographic order of the original iterators.

3.1.2 Polly

Polly [GZA^{+}11] is an optimization infrastructure for LLVM. Like PLUTO, Polly uses an abstract polyhedral representation to analyze and optimize loop and memory access pattern of a program. Currently classic loop transformations, such as, tiling and loop fusion as well as ones enabling OpenMP parallelism and SIMDization are provided by this framework. Since Polly is implemented within LLVM, it can support any LLVM frontend and more importantly generate code for wide range of backends including GPUs.
In contrast to PLUTO, Polly analyze the code for parallelism and tilability over the transformed AST [GVC15].

3.1.3 CHiLL

CHiLL [CCH08] is a compiler framework which provides rich set of composable transformations including loop permutation (interchange), tiling, unroll-and-jaand others. Developers of the framework argue that employing fixed transformation strategies as well as performing transformation in isolation have caused the codes generated by automatic optimizing compilers to fall short of manually optimized versions.
To optimize a code CHiLL reads a transformation script which describes the intended transformation sequence with minimal parameters. Afterwards, based on the provided script, CHiLL transforms the input program and generates high-quality code with minimum overhead (even for complex code constructs). Through uniform representation of iteration spaces and statements, CHiLL is able to compose multiple transformations without generating any intermediate code.

3.1.4 Halide

So far frameworks for optimizing regular applications have been discussed in this chapter. This section overviews Halide [RKBA+13] a DSL, with an optimizing compiler, embedded in C++. Halide systematically model the tradeoff space between locality, parallelism, and redundant recomputation in stencil pipelines. It also provides a scheduling representation that spans this space of choices. The DSL compiler then combines Halide programs and schedule descriptions to synthesize points anywhere in this space. Finally, the code generator produces high quality vector code for image processing pipelines, using a machinery much simpler than the polyhedral model. Halide also includes an auto-tuner that can infer high performance schedules for complex image processing pipelines using stochastic search.

3.2 Irregular Applications

Unlike code optimizers for regular application most of the effort in the area of optimizing applications with indirect memory accesses have been through DSLs. In this section three of successful DSLs as well OP2, an industry-quality library, are studied.

3.2.1 FEniCS

FEniCS [LMW+12] is an open source toolchain targeting automated solution of differential equations. Figure 3.1 illustrates the core components of FEniCS toolchain and the interaction among these components. Following paragraphs offer more details the building blocks of FEniCS.

DOLFIN. DOLFIN *(Dynamic Object-oriented Library for FINite element computation)* [LW10] is a C++/Python library which implements data structures and algorithms for computational meshes and finite element assembly. In other word, DOLFIN is a problem solving environment for models based on partial differential equations. DOLFIN wraps functionalities of other FEniCS components, as well as a few external softwares, and handles the communication among all these components.

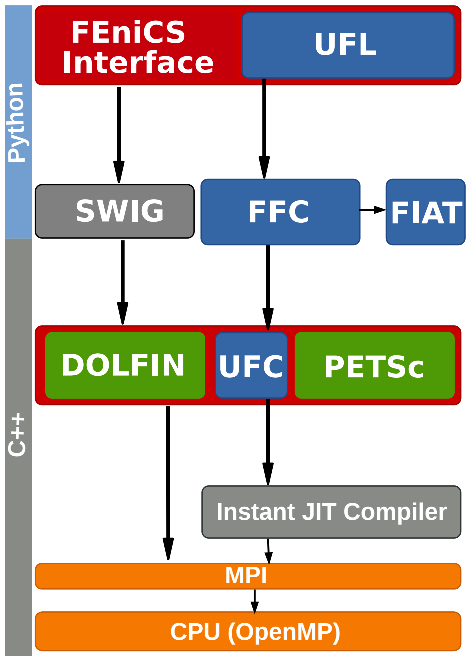


Figure 3.1: FEniCS toolchain- adapted form [Rat14a]

UFL. *The Unified Form Language* [ALn12] is a DSL, embedded in Python programming language, for describing finite element variational forms and functionals. Through this front-end weak forms and finite element spaces can be expressed very similar to mathematical notations. UFL acts as the front-end of the FEniCS Form Compiler [LoRW12] (discussed in next section). Moreover, it analyzes and transforms the expressions unaware of the mesh and function spaces.

FFC. FEniCS Form Compiler generates efficient C++ code from a high-level mathematical description (defined at UFL layer) of a finite element variational problem.

UFC. *Unified Form-assembly Code* [ALM12]is a unified framework for finite element assembly. UFC is the interface between the FFC and DOLFIN. More precisely, the UFC interface defines the structure and signature of the code that is generated by the form compilers for DOLFIN.

FIAT. *FInite element Automatic Tabulator* [KIr04] is implemented as Python module which automatically tabulates finite element basis functions over polynomial function spaces.

Instant. Instant [WMA12] is a Python module for just-in-time compilation C/C++ codes. Instant is built on top of the SWIG[Bea03] and Distutils which enables dynamic inlining of DOLFIN or FFC code.

3.2.2 Liszt

Liszt [DJP$^+$11] is a domain-specific language embedded in Scala for optimizing unstructured mesh applications for wide range of backends including CUDA, pthread and MPI.
Liszt extends the Scala language through abstract data types for mesh elements (nodes, edges, ...) which are put together in sets. Built-in function used to access to neighboring elements of mesh are used to discover the mapping among the mesh elements instead of the explicitly defining such relationships. In other words, since map values do not exist in Liszt, program analysis techniques are used to compute the stencil of expressions within a phase.
Code parallelization in Liszt is done at two levels: First, the coarse-grain partitioning for the MPI backend with automatic determination of halo elements which must be communicated between MPI processes. Second, the coloring strategy to avoid race conditions in pthreads and CUDA programming models.
Elegant high-level abstraction of the Liszt enables effortless implementation of mesh based application such as PDE solvers. Nonetheless, the inconvenience of interfacing Scala with commonly used programming languages in scientific community like Fortran and C, and therefore, powerful 3rd party libraries implemented in these language, puts Liszt at a disadvantage

3.2.3 OP2

OP2 [MGR+12] is an *active library*[1] [VG98] to parallelize execution of kernels, in loops with no intra-dependencies, over unstructured meshes. Through an unified API, OP2 generates optimized code for OpenMP, MPI and CUDA backends by utilizing source-to-source translation and compilation of transformed code.

Since the main contribution of this dissertation, Kaashi optimization framework, is inspired by this library, this section continues with detailed expositions of OP2 design, concepts and programming model.

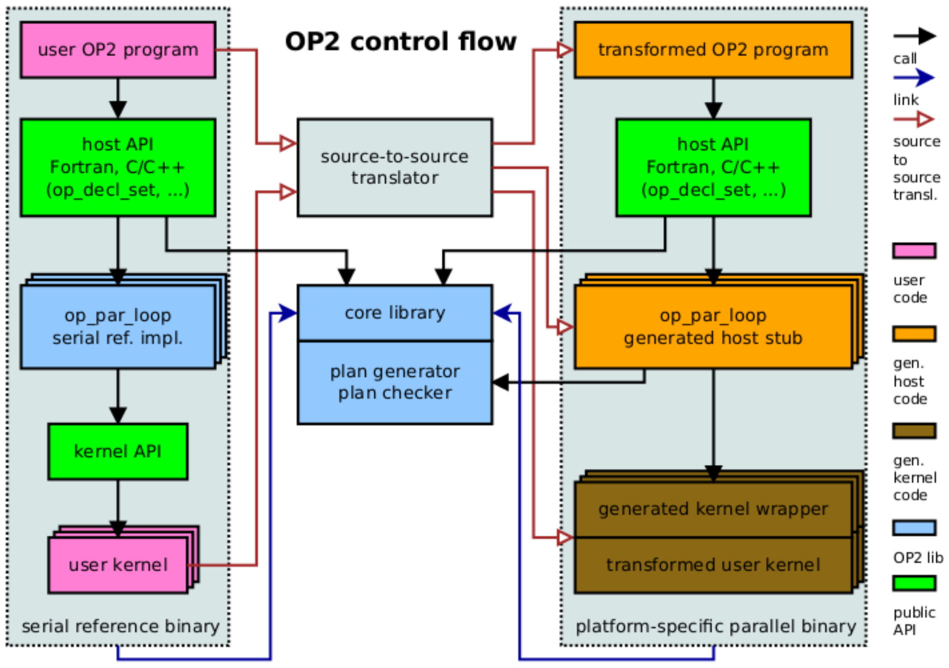


Figure 3.2: OP2 architecture. [Rat14c]

Design Strategies. OP2 components as well as the control flow of the program within the library is illustrated in figure 3.2. The application declares OP2 data structures as well as kernels complying with the library's restrictions, through the OP2 API. Since OP2 in an extension of C (and FORTRAN) language, unlike Liszt, exploiting full-fledged features

[1]Active libraries take an active role in compilation and may generate components, specialize algorithms, optimize code, configure and tune themselves for a target machine, and describe themselves to tools (such as profilers and debuggers) in an intelligible way.

of these language is fairly straightforward.

The serial reference implementation can be viewed as an unoptimized DSL, with no dynamic optimization, suitable for fast prototyping of applications over meshes. In this mode, shown on the left box of the figure 3.2 (within the gray box), OP2 acts as an static library where the source-to-source translation of the input code is bypassed and user kernels are left intact. Since no optimization is applied in this mode, the program execution is expected to be slow. To access optimization capabilities of the OP2 for each target architecture, the code translator must be utilized to generate high performance code (as shown on right side of the figure 3.2). The source-to-source translator analyzes each parallel loop invocation and, based of the characteristics of the loop, generates a code with a parallel execution plan and platform-specific kernel, a wrapper over the user kernel routine. The execution plan defines the partitioning of the datasets using a coloring strategy to avoid racing condition. Finally, for optimal execution, the automatically generated, platform-specific kernel is invoked based on the plan.

The OP2 Data Model.

The OP2 library is developed on the foundationod three basic abstraction:

Sets Mesh elements, such as nodes, edges, etc., are abstracted as *sets*. OP2 only keeps track of the size and name of the sets (*op_set*) and therefore is not able to distinguished them in any other way.

Data. This data structure (*op_dat*) associates data items to each element of a particular set. It is important to note that any number of *op_dat* may be associated to a single *op_set*.

Map *Maps* are access descriptors used to describe the mesh structure. Through maps the connectivities between elements of two op_sets are defined.

Maps also used to determine if the elements of a certain set are accessed *directly* or *indirectly* within a parallel loop. Indirect accesses imply that the elements of the set, in a particular

loop, will be accessed through the mappings to elements of neighboring sets. On the other hand, direct access means the set elements independently accessed.
In summary, OP2 is an abstraction framework, in a form of an active library, for parallel execution of the solution of applications over unstructured grids. OP2 parallelizes code for wide range of target architectures by creating optimized execution plan with respect to data dependencies for indirectly accessed data and reordering the data (AoS vs SoA) based on the backend characteristics.

3.2.4 PyOP2

PyOP2 [RMM$^+$12] started as a Python implementation of the OP2 library, however, it has since grown out of the initial goal and is by itself has become a powerful tool for applying parallel operations over unstructured meshes. PyOP2 shares the same basic abstractions (sets, maps, etc.) with OP2 and defines the topology of an unstructured grid and parallel loops over such meshes in a very similar manner.

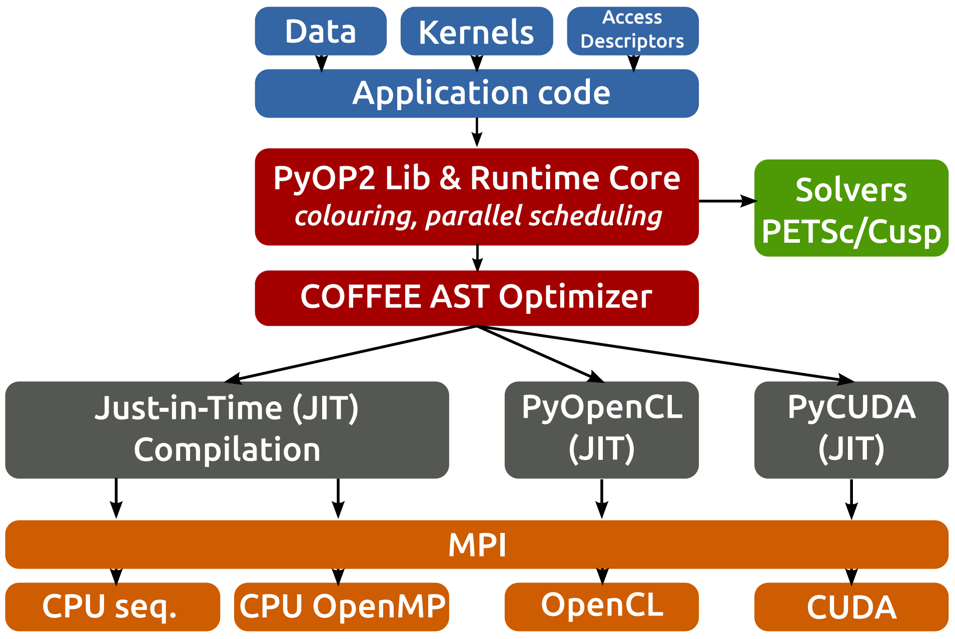


Figure 3.3: PyOP2 architecture.[Rat14b]

PyOP2 distinguishes itself from OP2 by providing easy to use interface to wildly used linear algebra solvers and custom designed tools, such as COFFEE [LVR+14], for optimized execution of codes used for solving PDEs using *finite element method* (FEM). In fact, PyOP2 is used as the parallel execution layer of Firedrake [RHM+15], an automated system for solving finite element problems.

Chapter 4
Harmonious Decomposition of Data and Computation through the Relation Abstraction

Due to the discrete nature of computers, most scientific programs express the domain of their application in the form of grids. This approach has led many codes to spend most of their time and resources on applying an operation (i.e., an instance of a *kernel*), to a small portion of a domain, usually, construed as a mesh element. One can view this as the local effect of a global phenomenon, i.e., the kernel, at a given moment in time. This implies that the reordering of the kernel execution, during a single time-step, is legal. Moreover, in the absence of reduction operations and dynamic global variables kernel instances can run in parallel. Such prevailing characteristics of scientific codes are prime motivations behind the research presented in this dissertaion.

In the first section of this chapter, we introduce the core concepts which serve as the cornerstones of the research conducted for this dissertation. Next, we elaborate our model to describe data and computaion spaces and continue by introducing our optimization approach based on such model.

4.1 Underlying Notions

To afford high-level abstractions, we place reliance on a handful of fundamental concepts. In this section, we expound these underlying notions.

Set. The founder of the *set theory*, Georg Cantor, has defined the set as [Can95]:

"A set is a gathering together into a whole of definite, distinct objects of our perception or of our thought—which are called elements of the set."

The *Set* in this research represents the same concept. This means a set could denote any ensemble of entities, for example, a collection of objects (like edges and cells), indices, or even iteration spaces. A set is defined by three attributes: (1) set name, (2) *cardinality*, i.e., the number of elements in the set and (3) list of items in the set. The latter is represented as a list of identification numbers which we call *global IDs*. Keeping track of set elements by their global IDs, instead of the actual objects, grants the opportunity to work with a set whilst being oblivious of actual object held in the set. For the cases where the access to the actual object is necessary we introduce the concept of the *data space*, discussed later in section 4.1.

Relation. Associations between a pair of sets are described by relations. This is analogue to the concept of binary relations in mathematics (see 2.1). The generality of the relation concept provides the foundation to express any range of mathematical objects such as matrices, grids, and so forth. Three components required to construct a relation are (1) the *departure* set, (2) the *codomain* set, and (3) the list of associations between each element of the departure set and a collection of elements in the codomain. One may view the relation as a characteristic function which indicates whether an element of the departure set corresponds to a particular element of the codomain. In consideration of keeping generality, we do not put any restrictions on arity of the relations, meaning, the relation arity does not have to be constant across all associations.

Data Space. A *data space* is a one-to-one relation between set's objects and their global IDs- one can view the global IDs as the indices of elements in the set. Keeping the set and data as two separate entities allows us to perform the data decomposition analyses with no knowledge on the actual data associated with the set.

We treat the data object as a map (in programming languages terms) where the global

IDs are keys and application's data objects are values of the map. In addition to this information, we also collect meta-data on some data qualities such as its type. These information are used for both automatic memory allocations, and the proper invocation of kernels by matching the types. As soon as the application data is attached to a data object, Kaashi takes over the management of the data for all purposes including storage, communication, and layout.

Computation Space. We define the *computation space* as a collection of relations between data sets and iteration sets. The computation space relates a series of *kernels* to a collection of data sets. To increase the accuracy of our data dependency analyses, in addition to relations between the iteration and data spaces, we also track data access modes. We recognize four modes of access: (1) only read, (2) just write, (3) read and then write, causing *WAR* dependency, or (4) write and then read (i.e., *RAW*). Furthermore, we distinctly recognize *reduction*, albeit a RAW dependency. Reductions happen when an object is read and written in one statement. A dependency imposed by a reduction prevents some data locality optimizations such as *interleaving*. We elaborate on such optimizations later in this chapter in section 4.4.

4.2 Load Balance Objectives

Orderly distribution of data across the resources is one of the biggest challenges of parallel programming. The ultimate goal of the domain decomposition is to achieve minimum execution time by efficiently dividing data among resources. There are two principals that are usually considered for optimal decomposition of the data domain:

1. Even distribution of workload.

2. Minimum communication between resources.

Unfortunately, there is no single algorithm capable of optimally partitioning all classes of applications for both these objectives. Applications like molecular dynamics require partitioners to preserve geometric locality of the data, while others, like FEM, benefit more from exploiting data interdependencies. Usually, applications with dynamic structure (e.g., adaptive meshes) are better-off with faster partitioning algorithms (at the expense of lower quality). On the other hand, codes with static domains perform better utilizing more expensive approaches. For example, partitioning algorithms based on hypergraphs increase the computation cost in early execution stages to, later, minimize the time spent on the remaining, iterative, parts of the application. The load balance approach introduced here is tuned to work best with partitionings based on hypergraphs since the primary targets of the our research are parallel applications with static domains. In the rest of this section, we elaborate on the data modeling of our approach based on the notions introduced in the last section. We also define the *cost-function* to be minimize for optimal decomposition of data.

4.2.1 Memory Analysis

Once the data and computation spaces are defined in terms of sets and relations, we can adapt existing partitioning algorithms, including graphs and hypergraphs, to partition the data space. In this work, we model relations as hypergraphs since our cost-function is based on the communication cost and hypergraphs best represent such models. We define the relation R_i between a pair of sets as the hypergraph H_i:

$$H_i = R_i(D_i, C_i) \tag{4.1}$$

where D_i and C_i are, respectively, departure (computation or iteration), and codomain (data) sets. Therefore, we can describe all the N relations of an application as:

$$H = \bigcup_{i=0}^{N-1} R_i(D_i, C_i) \tag{4.2}$$

subsequently, the memory requirement of an application can be written as:

$$M = \sum_{i=0}^{N_r-1} M_{R_i} + \sum_{j=0}^{N_s-1} M_{S_j} \tag{4.3}$$

where M_R and M_S are, respectively, memory requirements of relations and sets, while N_r and N_s correspond to the number of such relations and sets in the application. The objective is to maximize the load balance among the P processes while maintaining minimum number of communications for the computation.

4.2.2 Cost-Function

Once we formulated the memory requirements, we need to define the cost-function of our parallel partitioning algorithm. The objective is to distribute the data across resources with minimum load imbalance, and communication among processes.
Although it is more common to model the partitioning problem in terms of graphs we have decided to work with hypergraphs for two reasons:

1. hypergraphs model the communication more accurately.

2. hypergraphs allow modeling of non-symmetric dependencies among departure sets.

We should note that such benefits come at the cost of a more computationally expensive algorithm. However, the parallel implementation of our data decomposition approach and, also, the speed-up gained in iterative fragments of the application will compensate the overheads endured for partitioning.
Analogues to Williams' definition of the partitioning problem in [Wil91], we define the load balancing problem as a hypergraph-coloring problem: given a finite number of elements (vertices) and relation between them (nets), color (associate) them with K colors (processes), to minimize a cost-function F for a given coloring. By defining F as a *minimum-cut*

problem, we can write:

$$F(N, P) = \sum_{i=0}^{|E|-1} (\lambda_i(N, P) - 1) \tag{4.4}$$

where N is the set of nets of the hypergraph, P is the particular coloring we are evaluating and $\lambda_i(N, P)$ is the number of partitions (colors) spanned by the i^{th} net in the partitioning P. By adding the constraint $\lambda_i(N, P) \leq K$ we get the (K-1)-cut metric which can accurately reflect the cost of the communications [DBH+06]– We assume all the nodes in the system have similar throughputs for a given task.

4.3 Data Decomposition

Modeling the application, at the abstract level of sets and relations, allows us to reason with the minimal information on characteristics of the data objects. Additionally, sets and relations, as defined in this dissertation, imply that:

1. the order of applying kernels to the set elements does not affect the final result.

2. the sets and relations are static.

Taking advantage of these relaxed constraints, our data decomposition approach targets to optimize temporal and spatial localities, although in two separate stages. We target spatial locality at *inter-node level* —distributed memory— and temporal locality at *intra-node* —shared memory— layers. Following sections expound on how we achieve these objectives.

4.3.1 Partitioning Sets vs Data Structures

Considering the memory and processing performance gap in addition to the added cost of the communication network, leads us to, when possible, favor paying penalty in terms of computation in lieu of the communication. In other words, we aim to divide the workload equally among the resources and avoid the communication when possible. In other words, the priority in our data and computation reorganization approach is the minimization of the communication rather than the imbalance.

However, one should differentiate the partitioning of the sets and the data structures. The former only considers the number of memory receptacles required for a space described in the framework of sets and relations while the latter requires the actual size of the data memory space. The identical memory requirements of elements within a particular set allows our load balancing technique to minimize the cost function oblivious of the actual physical memory requirements of data objects. In fact, it is only the executer which needs to know the actual physical memory requirements.

4.3.2 Data Decomposition for Message Passing Systems

We defining the data space as the union of all data sets (Please note that D notation refers to data not departure):

$$S_{data} = D_1 \cup D_2 \ldots \cup D_n \tag{4.5}$$

In a *K-way* partitioning, the goal of our data decomposition is to divide each of the data sets into K subsets

$$D_i = s_1 \cup s_2 \ldots \cup s_K, \quad where \quad \forall i, j \quad if \quad i \neq j \quad then \quad s_i \cap s_j = \emptyset \tag{4.6}$$

where each subset has the same *cardinality*, i.e., number of members:

$$|s_1| = |s_2| \ldots = |s_K| \tag{4.7}$$

Recalling the memory requirements of an application from equation 4.3:

$$M = \textstyle\sum_{i=0}^{N_r-1} M_{R_i} + \sum_{j=0}^{N_s-1} M_{S_j}$$

and recognizing the uniformity of nodes' throughputs, we can calculate the optimal load for each node in a *K-way* partitioning:

$$\begin{aligned} M &= (\sum_{i=0}^{N_r-1} M_{R_i} + \sum_{j=0}^{N_s-1} M_{S_j})/K \\ &= (\sum_{i=0}^{N_r-1} M_{R_i})/K + (\sum_{j=0}^{N_s-1} M_{S_j})/K \end{aligned} \tag{4.8}$$

The above optimal load balance may not be achievable especially as we defined the objective of cost-function (equations 4.4) to be minimum communications. Also, such cost-function only considers the communication cost a single relation. One solution to over come the later is to impose the cost-function constraints to each individual relation, but there are two caveats with this approach:

1. the parallel hypergraph partitioning algorithm is expensive.
2. the partitioning of each relation must be done independent of the others, even though they may have shared sets.

In response to such challenges, we define one relation as the seed and impose the cost function on it. Once the codomain of the seed relation is partitioned, we color the remaining sets based on their adjacency, directly or indirectly, to the seed set, hence we will be able to:

1. minimize the number of required, expensive, hypergraph partitionings.
2. take the data dependency between sets of various relations into the consideration.

To partition the departure set of the relation R, we define the hypergraph $H = R(D, C)$, and impose cost-function F to partition the codomain set C:

$$F(N, P) = \sum_{i=0}^{|E|-1} (\lambda_i(N, P) - 1)$$

As soon as the *seed set*, i.e., the codomain of the seed relation, is decomposed, we can also partition the departure set based on the codomain following the algorithm 1. First we lookup the partitions containing each element of the codomain. Then, for each element of the departure set, d we lookup the partition rank of members of the codomain which are related to d and assign d to the minimum rank.

Algorithm 1: Partition the departure set based on the codomain partitioning.

```
Input  : Departure set $D$; codomain set $C$; relation $R$; partition hash table
         $HT_{codomain}$
1 Function $PartitionDeparture$ $(D, C, R, HT)$
2     for $c \in C$ do
3         $pins\_owner[c] = HT_{codomain}[c]$
4     end
5     for $d \in D$ do
6         $HT_{departure}[d] = min(pins\_owner[a]), \quad \forall\ a \in C$ adjacent to $d$.
7     end
Output: $HT_{departure}$
```

We may encounter two scenarios once the seed relation is partitioned:

1. there are sets which are not reachable from either of the seed relation's set.

2. all the sets within the data space are, directly or indirectly, related to the two sets of the seed relation.

The first case implies there are data sets that do not have any dependency to remaining sets of the data sets. Therefore, their decomposition does not affect the rest of the space as long as they are also partitioned evenly among the resources. Hence, we can use the same approach as before to partition them.

For the second scenario, suppose set x is partitioned and is related to an unpartitioned set u through R. In this case, s could be the departure or codomain of R. If it is the codomain set, we can apply algorithm 1 to partition u. We follow a similar approach for the opposite case, as detailed in algorithm 2.

Algorithm 2: Partition the codomain set based on the departure partitioning.

```
Input : Departure set $D$; codomain set $C$; relation $R$; partition hash table
        $HT_{departure}$
1 Function $PartitionCodomain\ (D, C, R, HT)$
2     for $d \in D$ do
3         for $p \in Pins[d]$ do
4             $HT_{codomain}[p] = min(HT_{codomain}[p], HT_{departure}[d]$
5         end
6     end
Output: $HT_{codomain}$
```

To partition a codomain set based on the departure set, each member of the codomain is assigned to the same partition as the net it is belonged to and has the minimum partition rank.

4.4 Computation Reorganization

Since the parallel processing, both in micro and macro scales, has emerged to be the dominant design, throughput has become the primary criterion of software optimizations. The maximum theoretical throughput of a particular system is only gained if:

1. there are enough chunks of data to keep all the processors active at all time.

2. parallel execution of such chunks (at least the same number of chunks as the number of processors at any point in time) is legal.

3. the distance between the processor and the data it operates on is zero.

Considering the performance gap between the memory and processors, and also sequential nature of many applications (or at least section of it) make it impossible to attain the ideal throughput. However, optimization for such prerequisites puts us on the right path to achieve higher effective throughput. The data partitioning approach discussed in the last section 4.3 is the backbone of our solution for management of the data chunks. The data decomposition distributes chunks of data almost equally among the resources.

Once the data space is evenly decomposed among the resources, it is time to reorganize the computation with the goal of minimizing execution time by minimizing the distance between the computation and the corresponding data. We pursue this strategy by improving data locality in two ways:

1. **Temporally.** Once a data is available in the processor, do all the possible computations on it.

2. **Spatially.** Keep the data to be processed as close as possible to the resource which will access it.

The rest of this section expounds our strategies for such optimizations.

4.4.1 Data and Computation Bloc

So far, we have profited from the relation abstraction for the data decomposition. In this section, we study strategies developed to benefit from such abstraction to schedule the computation.

Our goal is to form a *bloc* of data and computation with the purpose of eliminating data movements whenever possible. For this we define both the data and computation spaces (we call the combination of the two the *application domain*) in terms of sets, describe an association between a pair of them in terms of relations. Meanwhile, as discussed in 4.1, we also collect information on data qualities and access modes. The resulting framework enables us to systematically examine dependencies and efficiently arrange the data and computation together.

Similar to data decomposition, the computation set is broken down into K subsets. Our aim is to find the optimal relation between each subset of data and computation which ensures lowest cost of communication.

Since the associations between computations and their corresponding data are modeled based on relations, algorithm 1 can readily be used to partition the computations. In this approach, the computation space is represented as the departure set and the data space serves as the codomain of the relation.

In short, we divide the data and computation domain, and consequently the relation between the two, into a series of subsets. Each of these subsets could be viewed as a tile, in compiler terms, and the overall decomposition as the interleaving of the relations.

4.4.2 Avoiding Data Movement

Hitherto we have been able to evenly partition the application domain among the resources and minimize the distance between computation and the data. In other words, we have improved the spatial and temporal locality for a single iteration. Nevertheless, most of the scientific applications involve iterations over the same domain, either till convergence or for a fix number of iterations. In such cases, at the end of each iteration, data dependencies across partitions force some communication on boundaries of partitions. In this section we introduce an approach which enables us to identify dependencies that could be respected without the communication. A dependency is not carried by the outer iteration if:

1. it is first written into at the beginning of the iteration.

2. it is only read throughout the iteration.

If a data is not owned by a process but it is consumed by it, a copy of the data is transferred to such process before the first iteration begins. Now, if the first operation on the data is a write, this means that the process will have access to the latest version of the data for the following iterations and there is no need for communication in such cases. Also, if a data object is only read within the application domain, once a data is provided to a resource, it is guaranteed that the data is up to date for all iteration. Hence, communication for such objects can also be avoided. Collecting the access mode information in addition to the relation between the data and computation, enables us to straightforwardly recognize data sets that do not carry dependencies across the iterations. We will avoid communication for such data sets.

4.5 Conclusion

In this chapter we introduced a high-level framework based on the set and relation abstractions to efficiently describe the application domain. This model is accompanied by a cost-function which accurately models the data movement cost. Also, we introduced a new loop interleaving technique for for harmonious decomposition of data and computation to improve spatial and temporal locality within sequence of loops. Finally, we discussed how our introduced abstraction for describing the relation between computation and data can be exploited to help us avoid unnecessary communication.

Chapter 5
The Kaashi Parallelization Framework

Many numerical algorithms and scientific computations such as particle simulations, and FEMs can be viewed as the independent application of kernels on local elements of an unstructured grid. Despite their parallel nature, the inherited indirect memory accesses in such applications makes them difficult to parallelize and optimize during the compilation. Many [SGO12, ATD04, MSS+88], pioneered by [SMC91], have proposed a run-time system based on the inspector/executor strategy. At run-time, the inspector traverses the application domain and extracts access pattern information to plan an optimized computation scheme. However, inferring data and computation dependencies, especially in the absence of user directives, is costly and not always feasible in practice. In this chapter, we introduce *Kaashi*, a run-time framework based on the relation abstraction implemented in C++. Kaashi is designed to facilitate the describing of the application domain and entailed dependencies, and to automatically handle the cumbersome tasks of memory management and data movement (communication) across parallel resources.

Kaashi primarily targets *irregular applications* (those with indirect memory accesses, usually over unstructured meshes). The user friendly interface of the Kaashi enables the developers to describe the domain of their application in terms of a collection of sets and series of relations among them. On the other hand, the relational model built by Kaashi's translator, provides a robost yet easy to use framework to interpret dependencies and optimize execution order.

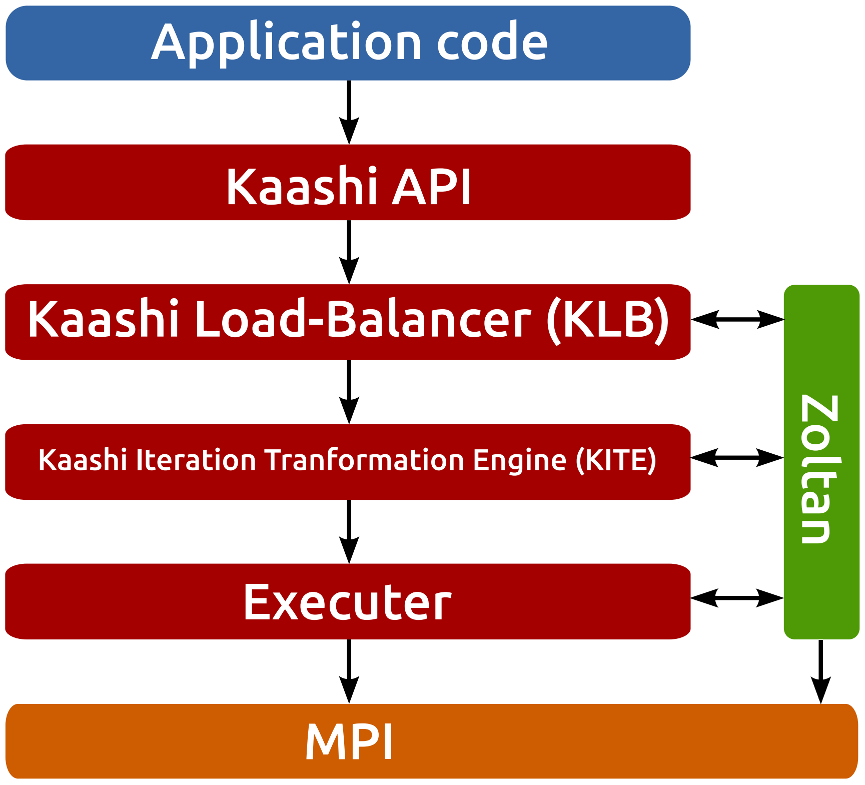

Application code
Kaashi API
Kaashi Load-Balancer (KLB)
Kaashi Iteration Tranformation Engine (KITE)
Executer
Zoltan
MPI

Figure 5.1: Schematic overview of the Kaashi.

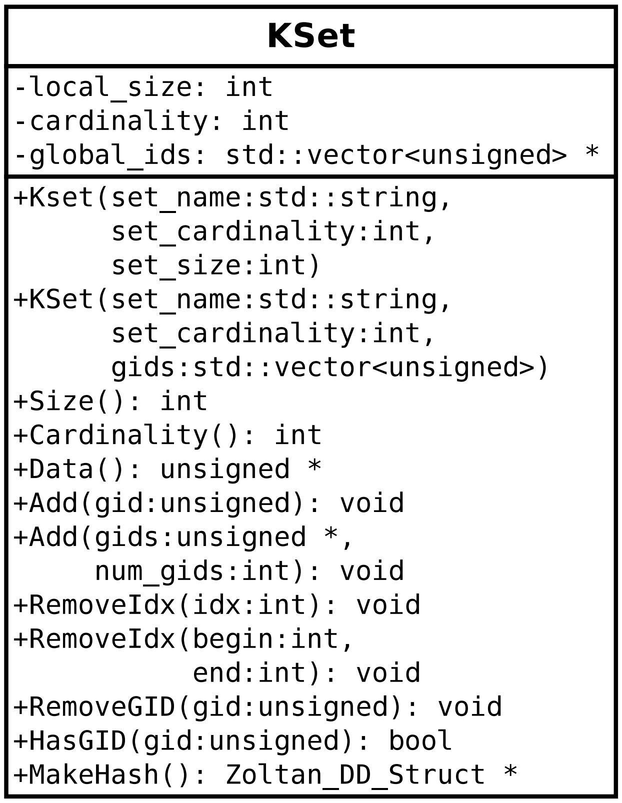
Figure 5.1 shows the modular view of the Kaashi library. As illustrated in this diagram, conforming with our streamline design, Kaashi solely depends on Zoltan and no other third party library. The rest of this chapter is dedicated to explain designs of the building blocks of the Kaashi and exhibit its capabilities.

5.1 Core Concepts

The design of the Kaashi API allows the programmer to describe sets and relations abstractly. KDT translates such descriptions into a distributed relational model and allocates the objects. The application domain's data and computation (iteration) spaces are represented as sets. Meanwhile, there are two separate classes to represent relations: (1) KRelation to describe a relation between a pair of data sets, and (2) KER to define a relation between a single computation set and one or more data sets. KER also records memory access modes. Distinguishing the two types of relations enables more complex optimizations, like such as loop interleaving, discussed later in this chapter. The rest of this section provides implementation details of the set and relations concepts.

5.1.1 Kaashi Set (KSet)

The set abstraction is implemented as the class KSet. Sets are abstract representations of both data and iteration (computation) spaces. The KSet diagram 5.2 illustrates how Kaashi's design closely mimics the set abstraction in the mathematical term. Besides the standard set functionalities, we have implemented the *MakeHash* member function which creates a distributed directory of set element locations. The globally accessible distributed directory is in fact a distributed hash table with the objects' global IDs as the keys.



```
KSet
-local_size: int
-cardinality: int
-global_ids: std::vector<unsigned> *
+Kset(set_name:std::string,
      set_cardinality:int,
      set_size:int)
+KSet(set_name:std::string,
      set_cardinality:int,
      gids:std::vector<unsigned>)
+Size(): int
+Cardinality(): int
+Data(): unsigned *
+Add(gid:unsigned): void
+Add(gids:unsigned *,
     num_gids:int): void
+RemoveIdx(idx:int): void
+RemoveIdx(begin:int,
           end:int): void
+RemoveGID(gid:unsigned): void
+HasGID(gid:unsigned): bool
+MakeHash(): Zoltan_DD_Struct *
```

Figure 5.2: Class diagram of KSet- the set abstraction within Kaashi.

Sets, as shown in figure 5.3, are defined by their name (which also acts as the identifier) and the cardinilaity, i.e., the global size of the set. Since sets are distributed objects, Kaashi also stores the *local size* of the set (the part of the set within a particular process). Kaashi numbers the set elements, meaning, assign global IDs, sequentially starting from zero. However, if the application provides the global IDs, Kaashi follows the user's numbering. Please note that at this point we do not distinguish between the data and computation sets since the KSet abstracts away the type of its elements.

```
KSet(std::string set_name,
     int set_cardinality,
     int set_size);

KSet(std::string set_name,
     int set_cardinality,
     std::vector<unsigned> gids);
```

Figure 5.3: Kaashi API for defining sets.

5.1.2 Kaashi Data (KData)

KData is the encapsulation of the user data within the Kaashi framework. KData associates the actual data object values to their corresponding index system defined as KSets. Once a KData is defined, Kaashi takes over the management of its data objects, in other words, KData abstractly represents the distributed array which holds data values. Encapsulating the user data in KData and also separating the actual values from their indices allows Kaashi's *load balancer* **??** to efficiently partition and manage the data objects among all participating processes for optimized distributed parallel computations.

```
KData(KSet &data_set,
      KType data_type,
      void *dat);
```

Figure 5.4: KData constructor.

5.1.3 Kaashi Relation (KRelation)

A relation between two sets, where both represent data, is defined through the KRelation (figure 5.5) by determining the departure and codomain sets as well as the relation name and the associativity list between the sets.

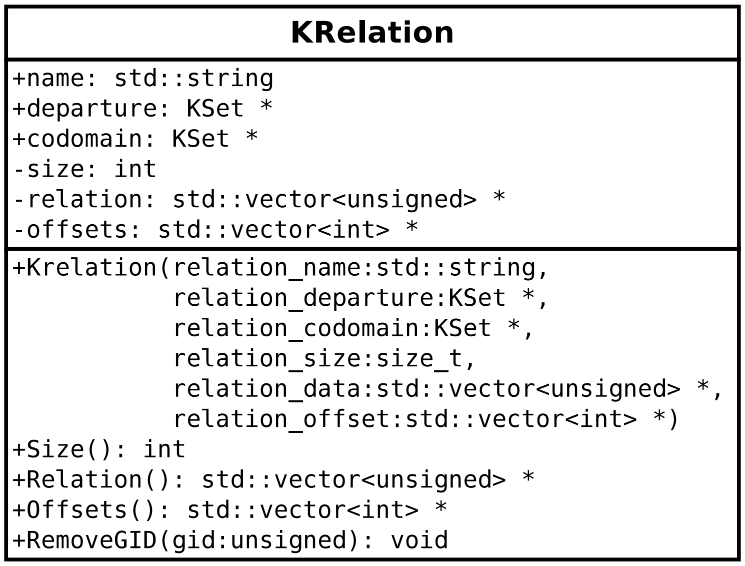
The associativity is represented as an adjacency matrix between the codomain and departure sets. Considering the sparse nature of the relations, the adjacency (associativity) information of the relations are implemented as sparse matrices- represented in Compressed Sparse Row (CSR) format. We choose to work with sparse matrices as they guarantee minimal memory footprint and, also, comply by our load balancing algorithm **??**.

```
  KRelation(std::string relation_name,
            KSet *relation_departure,
            KSet *relation_codomain,
            size_t relation_size,
            std::vector<unsigned> *relation_data,
            std::vector<int> *relation_offset);
```

Figure 5.5: Kaashi API for defining relations.

The actual values of the relations, i.e., the sparse matrix, just like sets, are distributed across the processes. The keys of a relation's hash table is the global ID's of its departure set. Figure 5.6 shows the class diagram of the KRelation.

Figure 5.5 shows the signature of the KRelation constructor. The *relation_data* and the *relation_offset* are pointers to vectors which keep track of the segment of the adjacency matrix that is stored within the process- note the *relation_size* is the local size of the relation.



```
KRelation
+name: std::string
+departure: KSet *
+codomain: KSet *
-size: int
-relation: std::vector<unsigned> *
-offsets: std::vector<int> *
+Krelation(relation_name:std::string,
           relation_departure:KSet *,
           relation_codomain:KSet *,
           relation_size:size_t,
           relation_data:std::vector<unsigned> *,
           relation_offset:std::vector<int> *)
+Size(): int
+Relation(): std::vector<unsigned> *
+Offsets(): std::vector<int> *
+RemoveGID(gid:unsigned): void
```

Figure 5.6: Class diagram of KRelation- the relation abstraction within Kaashi.

5.1.4 Class Kaashi

The class Kaashi acts as a wrapper over all the other Kaashi's functionalities. Kaashi acts as a glue which manages the interaction between different components of the framework. Each functionality of the Kaashi framework has a corresponding data structure which maintains all the responsibilities for one instance it. Each class is defined in a header file and encapsulates a Kaashi data structure and the functions that operate on that structure. The Kaashi class allows the user to access all these functionalities through one streamlined class with minimal efforts. Figure 5.7 demostrates how one can define the data space with the Kaashi's framework.

```
Kaashi(MPI_Comm mpi_communicator);

void Kaashi::AddSeedRelation(std::string departure_name,
                             int departure_cardinality,
                             std::string codomain_name,
                             int codomain_cardinality,
                             std::string relation_name,
                             size_t relation_size,
                             std::vector<unsigned> *relation_data,
                             std::vector<int> *relation_offset);

void Kaashi::AddRelation(std::string departure_name,
                         int departure_cardinality,
                         std::string codomain_name,
                         int codomain_cardinality,
                         std::string relation_name,
                         size_t relation_size,
                         std::vector<unsigned> *relation_data,
                         std::vector<int> *relation_offset);// As many needed.
```

Figure 5.7: Describing the application's data space in Kaashi.

Kaashi distributes data and relations and generates the corresponding directory of addresses as soon as they are added to the framework. The data management is done through *Kaashi Load Balancer (KLB)*, introduced in next section.

5.2 Kaashi Load Balancer (KLB)

Kaashi provides two separate routines to define relations. If non of the departure and codomain sets are defined, the relation is defined by *Kaashi::AddSeedRelation*. KLB treats such relations as a parallel hypergraph and distributes all three departure set, codomain set, and the relation. The load is balanced across the resources based the cost-function introduced in 4.4 , i.e., for minimum communication cost:

$$F(N,P) = \sum_{i=0}^{|E|-1} (\lambda_i(N,P) - 1)$$

KLB carries out the data movement according to such partitioning. Also, to keep track of data locations, the load-balancer generates the *Kaashi index database* (KID)– implemented as a distributed hash table. Meanwhile if only the departure or codomain set is not partitioned KLB follows 1 or 2 respectively.

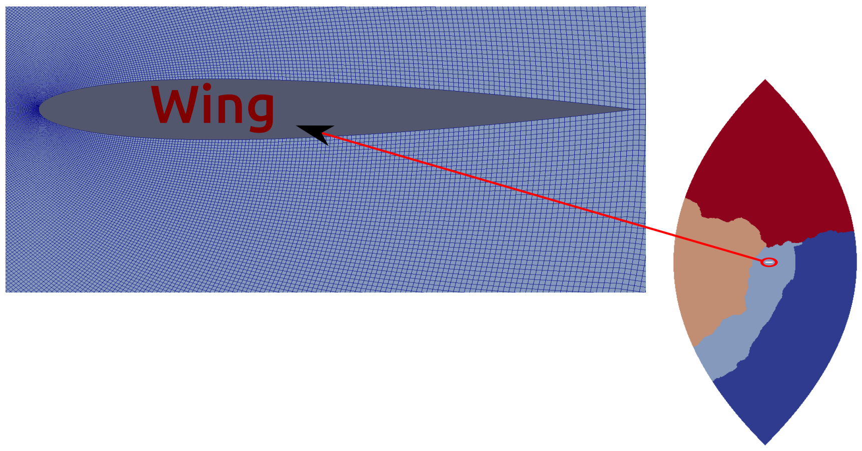


Figure 5.8: Simulation of the airflow around the airplane wing.

5.3 Kaashi's Iteration Transformation Engine (KITE)

The goal of the Kaashi framework is to reorganize the data and computation space with the aim of achieving maximum performance. *Kaashi's iteration transformation engine (KITE)* is implemented to carry out the analyses of finding the best iteration permutation matching the data distribution done and indexed by KLB.

Before we can explain the KITE approach for scheduling the computation, we introduce two essential Kaashi classes in the following sections.

5.3.1 Kaashi Extended Relation (KER)

Kaashi Extended Relation represents the relations defined between a computation set, namely, the *iteration set* and one or more data sets. In addition, KER object constructor also takes *access descriptors* which describe how each data set is accessed by the kernel.

```
KER(KData data,
    KRelation relation,
    KAccess access_mode);
```

Figure 5.9: KER constructor.

5.3.2 Kaashi Parallel Iteration (KPI)

Parallel iterations imitate for loops in other ordinary programming languages, however, with one important characteristic: The order of applying the kernel on data objects does not affect the final result. In other words, any permutation of the iteration space is legal. KPI, as shown in 5.10 encapsulates a computation set, a list of KERs based on such set and a kernel to be applied to each elements of the data set.

```
KPI(std::string kernel_name,
    KSet *iter_set;
    std::vector<KER> kers);
```

Figure 5.10: KPI constructor.

5.4 Experiment

In this section we provide an example application to examine Kaashi's capabilities. As shown in figure 5.11, our experiment [GGD05] is a two dimensional inviscid airfoil application over an unstructured grid. This irregular application uses a cell-centred discretization of the domain. The domain described in this example includes 72000 cells, 721801 nodes and 1438600 edges.

Figure 5.12 shows how the load among the resources has been balanced for all resources across variety of configurations. Figure 5.13, demonstrates how utilizing Kaashi not only simplifies the programming of the application, but also how high performance could be acheived through this library.

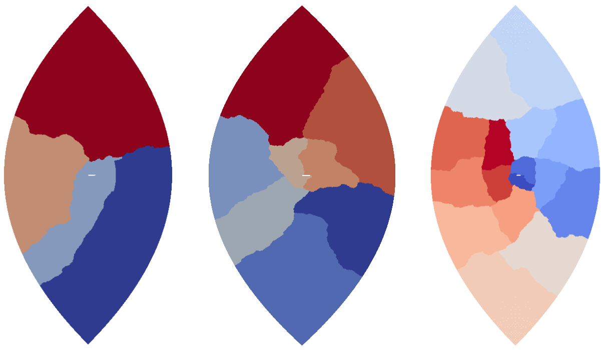


Figure 5.11: Load balance for various number of processes.

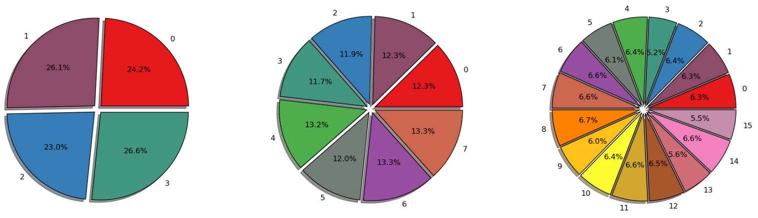


Figure 5.12: Balance distribution of the load among the processes.

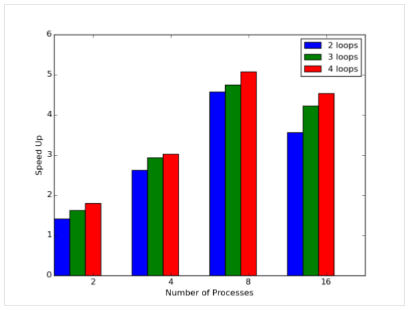


Figure 5.13: Speed up against sequential execution.

5.5 Conclusion

In this chapter we introduced Kaashi, a run-time parallel memory manager and scheduler written in C++. Kaashi provides a high-level interface based on set and relation abstractions which can efficiently model many scientific programs. Throughout the design of the

Kaashi, we have strived to preserve the generality and stick to abstract concepts when possible. The chief target of the Kaashi is irregular applications, namely, applications with indirect memory access. Nevertheless, performance improvement is expected for any program definable in terms of Kaashi.

Chapter 6
Conclusions

The complexity of today's computing systems, due to the heterogeneity of hardware architectures as well as the programming models, has steadily made the programming of such systems more challenging. In this thesis we have provided solutions to alleviate such difficulties through new approaches and tools. In the next section, we briefly review our contribution and conclude with ideas for future works.

6.1 Contribution

In this work we have tackled the challenges risen from heterogeniety of HPC in two ways:

1. introducing efficient algorithms to analyze and optimize irregular applications
2. providing a framework with user-friendly interface to manage error-prone and cumbersome tasks of memory management and data movement.

Our scheduling algorithm focuses on irregular applications and aims at avoiding communication to improve performance. We introduce overlapped tiling of the domain during each iteration step. Our algorithm minimizes the communication requirements by:

1. efficiently distributing the data and computation with the goal of minimizing the distance between the two.
2. avoiding the communication by tracking memory access patterns.

Our algorithm utilizes a parallel hypergraph partitioning algorithm with a cost-function based on the communication cost to minimize the distance between the computation and the corresponding data. Furthermore, when possible, our algorithm redundantly schedule a single task on multiple resources in order to avoid expensive communication. Finally, the memory accesses are tracked to determine the scope of the changes for each memory location. The communication is again avoided if the changes are contained within a single resource.

In addition to these algorithm we introduced Kaashi, a run-time parallel memory manager and scheduler written in C++. Kaashi provide a high-level easy to use interface which allows developer to describe their applications in terms of sets and relations. Throughout the design of the Kaashi, we have strived to preserve the generality and stick to abstract concepts when possible. The chief target of the Kaashi is irregular applications, namely, applications with indirect memory access. Nevertheless, performance improvement is expected for any program definable in terms of Kaashi.

The modular design of the library allows the user to rely on the library according to their needs. More experienced programmers may want to utilize the lower level functionalities of the library without losing the control, while others may allow Kaashi to entirely take over the management of data and computation.

6.2 Future Work

The Kaashi library has been designed with future expansions in mind, meaning, third party developers can access Kaashi's data structures transparently and extend the libraries functionalities. In the following, we discuss a number of possible areas to improve Kaashi's functionalities.

Power. Since the energy consumption of new processors hitting the power wall, power consumption analyses is no more limited to embedded system and must be considered as an optimization criterion.

Artificial Intelligence. As mentioned before, the primary target of the Kaashi is the irregular applications. The lack of formal parametric abstractions to describe such applications makes them a suitable candidates to use heuristics and machine learning techniques for optimizations. Such techniques can be utilize for determining the best partitioning size, best performing backend, or different optimization techniques according to features detected by Kaashi.

Bibliography

[ALM12] Martin S. ALnæs, Anders Logg, and Kent-Andre Mardal. *UFC: a Finite Element Code Generation Interface*, chapter 16. Springer, 2012.

[ALn12] Martin S. ALnæs. *UFL: a Finite Element Form Language*, chapter 17. Springer, 2012.

[ATD04] Manuel Arenaz, Juan Touriño, and Ramón Doallo. An inspector-executor algorithm for irregular assignment parallelization. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 4–15. Springer, 2004.

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

[Bas07] Cédric Bastoul. Chunky analyzer for dependencies in loops. `http://goo.gl/4TK3p1`, 2007.

[BCG$^+$03] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, Texas, october 2003.

[Bea03] David M Beazley. Automated scientific software scripting with swig. *Future Generation Computer Systems*, 19(5):599–609, 2003.

[BHRS08] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[ÇA11] Ümit Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011.

[Can95] Georg Cantor. Beiträge zur begründung der transfiniten mengenlehre. *Mathematische Annalen*, 46(4):481–512, 1895.

[CCH08] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[DBH$^+$02] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science & Engineering*, 4(2):90–97, 2002.

[DBH$^+$06] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, and Umit V Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[DBR$^+$09] Karen D Devine, Erik G Boman, Lee Ann Riesen, Umit V Catalyurek, and Cédric Chevalier. Getting started with zoltan: A short tutorial. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[DJP$^+$11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2011.

[Fea88] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[Fea91] P Feautrier. Data flow analysis of scalar end array references. *Intern. Journal of Parallel Programming*, 20(1), 1991.

[GGD05] MB Giles, D Ghate, and MC Duta. Using automatic differentiation for adjoint cfd code development. 2005.

[GL96] Martin Griebl and Christian Lengauer. The loop parallelizer loopo. In *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21, pages 311–320. Citeseer, 1996.

[GLW98] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 106–111. IEEE, 1998.

[Gri04] Martin Griebl. *Automatic parallelization of loop programs for distributed memory architectures*. Univ. Passau, 2004.

[GVC15] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems*, 2015.

[GZA$^+$11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[HNS09] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. Annotation-based empirical performance tuning using orio. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[HP01] B Hendrickson and A Pinar. Communication support for adaptive computation. In *Proc. 10th SIAM Conf. Parallel Processing for Scientific Computing*, 2001.

[Kar11] George Karypis. Metis and parmetis. In *Encyclopedia of Parallel Computing*, pages 1117–1124. Springer, 2011.

[KIr04] Robert C. KIrby. Algorithm 839: Fiat, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software*, 30(4):502–516, 2004.

[KPR95] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*, pages 332–341. IEEE, 1995.

[LCL99] Amy W Lim, Gerald I Cheong, and Monica S Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, pages 228–237. ACM, 1999.

[LLL01] Amy W Lim, Shih-Wei Liao, and Monica S Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. *ACM SIGPLAN Notices*, 36(7):103–112, 2001.

[LMW$^+$12] Anders Logg, Kent-Andre Mardal, Garth N. WElls, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[LoRW12] Anders Logg, Kristian B. ølgaard, Marie E. ROgnes, and Garth N. WElls. *FFC: the FEniCS Form Compiler*, chapter 11. Springer, 2012.

[LVR$^+$14] Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J Ramanujam, David A Ham, and Paul HJ Kelly. Coffee: an optimizing compiler for finite element local assembly. *arXiv preprint arXiv:1407.0904*, 2014.

[LW10] Anders Logg and Garth N. WElls. Dolfin: Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2), 2010.

[MGR$^+$12] G Mudalige, MB Giles, I Reguly, C Bertolli, PHJ Kelly, et al. Op2: An active library framework for solving unstructured mesh-based applications on multicore and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

[MSS$^+$88] Ravi Mirchandaney, Joel H Saltz, Roger M Smith, DM Nico, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 2nd international conference on Supercomputing*, pages 140–152. ACM, 1988.

[Pat11] David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[PR96] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference on High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[Rat14a] Florian Rathgeber. Firedrake: Re-imagining fenics by composing domain-specific abstractions. `https://goo.gl/mfvvij`, 2014.

[Rat14b] Florian Rathgeber. Firedrake: Re-imagining fenics by composing domain-specific abstractions. `https://goo.gl/zprrmQ`, 2014.

[Rat14c] Florian Rathgeber. Productive and efficient computational science through domain-specific abstractions. 2014.

[RHM$^+$15] Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. Firedrake: automating the finite element method by composing abstractions. *arXiv preprint arXiv:1501.01809*, 2015.

[RKBA$^+$13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[RMM$^+$12] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David Ham, Carlo Bertolli, Paul HJ Kelly, et al. Pyop2: A high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1116–1123. IEEE, 2012.

[SGO12] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 61–75. Springer, 2012.

[SMC91] Joel H Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on computers*, 40(5):603–612, 1991.

[VG98] T Veldhuizen and E Gannon. *Active libraries: Rethinking the roles of compilers and libraries*, volume 5. Oct, 1998.

[Wil91] Roy D Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and experience*, 3(5):457–481, 1991.

[WMA12] Ilmar M Wilbers, Kent-Andre Mardal, and Martin S Alnæs. Instant: just-in-time compilation of c/c++ in python. In *Automated Solution of Differential Equations by the Finite Element Method*, pages 257–272. Springer, 2012.

[zol] Zoltan user's guide: Communication utilities.

Vita

Mohammad Rastegar Tohid was born in Tehran, Iran, in 1983. In 2006 he finished his BSc in Computer Engineering at Azad University. The same year he moved to England and got his MSc in Communication and Signal Processing from Newcastle University in 2008. After spendig a year as a guest researcher, Mohammad moved to United States and began his PhD in Electrical and Computer Engineering at Louisiana State University. Mohammad will graduate in May 2017.