SPARSE MATRICES AND SUMMA MATRIX MULTIPLICATION

ALGORITHM IN STAPL MATRIX FRAMEWORK

A Thesis

by

DIELLI HOXHA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Nancy M. Amato
Co-Chair of Committee,	Lawrence Rauchwerger
Committee Members,	Jean Ragusa
Head of Department,	Dilma Da Silva

May 2016

Major Subject: Computer Engineering

ABSTRACT

Applications of matrices are found in most scientific fields, such as physics, computer graphics, numerical analysis, etc. The high applicability of matrix algorithms and representations make them an important component in any parallel programming language, therefore matrix frameworks are a continuous research effort in high performance computing. This work focuses on a generic matrix framework in the STAPL library. First, we extend the STAPL library by adding a sparse matrix container. Second we implement SUMMA, the parallel matrix-multiplication algorithm, for fine grained computations. Then, implement parallel matrix-matrix algorithms for the sparse matrix container. Finally, we conduct experimental studies for each of the components we have implemented and discuss the findings. Experiments are conducted on a Cray XE6m cluster. Experimental studies consist of multiple matrix and data inputs that showcase and stress the matrix models implemented. We find that the sparse matrix container outperforms its dense counterpart in sparse inputs, and vice versa. Both containers, and the matrix summa implementation show scalability up to 512 cores.

DEDICATION

To my family.

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Present day machines contain multiple processing elements that application developers utilize with parallel processing to divide problems and solve them faster. Parallel systems allow us to solve problems that ordinary sequential computers cannot solve or produce results in a more timely manner. However, parallelism introduces new challenges to the programmer, which do not arise in sequential programming. Designing parallel programming languages and libraries that are easy to use and deliver performance is a challenging and continuous research effort.

Parallel matrices are used in many modern industrial applications to provide high performance solutions, therefore they present an important component in any parallel programming system. Parallel matrices are a well researched topic, and work has focused on the type of the matrix (eg. sparse, dense), and the target environment (eg. shared or distributed memory). Matrix-matrix multiplication is one of the most researched operations.

The research conducted in this thesis is written in the STAPL library, which is a collection of parallel algorithms and datastructures. The STAPL components are shown in Figure 1.1 and described in Section 2 and 3.

1.1 Contributions

A parallel sparse matrix implementation was developed using the STAPL `pContainer` framework by providing specialized components to implement sparse matrix functionality. STAPL currently has a dense matrix container. Along with the implementation of the sparse matrix container, parallel operations for sparse matrices were also implemented.

Implementing the parallel SUMMA algorithm using algorithmic skeletons required

extending the STAPL skeleton framework to support the computations. The algorithm is implemented by providing new components that implement the operations and express the data dependencies of the matrix multiplication algorithm. The extensions and modifications of STAPL were implemented in a manner consistent with the STAPL programming model.

In short the contributions of this thesis can be summarized as follows:

1. Extend the STAPL skeleton framework by adding a matrix multiplication algorithm skeleton and specific skeleton components for the SUMMA algorithm.

2. Implement a parallel sparse matrix container and a set of arithmetic operations defined for matrices (e.g., subtraction, addition, etc.) using the STAPL's `pContainer` framework.

3. Conduct performance and scalability studies for the sparse matrix container and for SUMMA algorithm implementation.
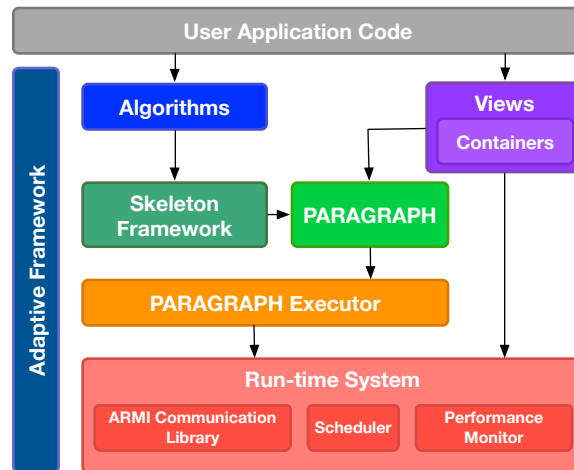
Figure 1.1: The STAPL components.

1.2   Outline

In Section 2 we first give a brief overview of STAPL, a parallel programming library for C++. Then we describe the related work focusing specifically on parallel matrix-matrix multiplication and parallel sparse containers. In Section 3 we give a brief outline of the STAPL `pContainer` framework and discuss our implementation of the STAPL sparse matrix. In Section 4 we discuss the STAPL algorithmic skeleton framework. Then, we explain in detail our implementation of the SUMMA algorithm using algorithmic skeletons. In Section 5, we discuss the performance results achieved by the STAPL sparse matrix, its operations, and the SUMMA implementation.. Finally, Section 6 gives conclusions and suggests future work.

2. PRELIMINARIES AND RELATED WORK

2.1 STAPL Overview

A major challenge of parallel programming is productivity, because programming in parallel is harder than sequential since it requires coordination of the work being performed by each processing element. The Standard Template Adaptive Parallel Library (STAPL) [2] attempts to simplify parallel programmability by providing global object programming model, ability for automatic tuning, and a uniform communication runtime system.

STAPL is a parallel programming library for C++ that provides a collection of distributed data structures (containers) and parallel algorithms, and a methodology for easy customization specific to different applications. STAPL is designed to provide an interface similar to that of the Standard Template Library (STL). STAPL allows users to express algorithms with a unified view of data independent of memory model while hiding the details of concurrent execution. STAPL is a programming language based on the data-flow paradigm and it uses a data-flow executor known as the PARAGRAPH to execute task-graphs of parallel programs. Taskgraph generation is accomplished through Algorithmic Skeletons.

The implementation of the sparse matrix container in STAPL is based on the pContainer framework [12]. The implementation of the SUMMA matrix multiplication algorithm is accomplished using algorithmic skeletons [15]. The implementation of the components is discussed in detail in Section 3 and 4. The rest of this section discusses related work in two areas. First in sparse matrix containers, and second, in parallel matrix-matrix multiplication algorithms.

2.2   Parallel Sparse Matrix Containers

Many applications in engineering, scientific computing, economics, etc, use very large matrices that have high dimensionality and prove to be too difccult and impracticable to use because of their large sizes. Reducing dimensionality in matrices in order to get a better understanding of the data is a common practice. In many cases, reducing dimensionality of large amounts of data results in matrices which are mostly empty (i.e. contain very small amount of nonzero elements). These matrices are known as sparse matrices. The goal is to find representations of sparse matrices in today's computers such that zero elements are disregarded, not stored, and computation can be orchestrated using nonzero elements only. This has multiple benefits such as considerable savings in storage space requirements as well as performance improvements. Many formats have been suggested to represent sparse matrices and they all focus on different objectives such as straightforwardness, simplicity, generalization, storage reduction, etc. The key objective in providing parallel sparse matrix containers is to provide matrix applications that are faster than any implementation that can be expressed using a single processor computer.

There are many formats that are used for sparse storage representation, they can be organized based on the architecture they are targeting, for example vector architectures, shared memory or distributed memory architectures. The BLAS [1] framework recommends many matrix formats such as Coordinate (COO), Block Compressed Row (BCRS), Compressed Row (CRS) etc [5]. The most straightforward sparse matrix scheme is the Coordinate (COO) scheme. This scheme stores the row and column indices of every nonzero element. The Coordinate scheme is not the most efficient because it involves a great deal of indexing, therefore slowing down the computations. Another format is the Compressed Row Format (CRS) which

puts nonzero elements of rows one after the other in consecutive positions. This is the most widely used format because its very straightforward and efficient. Further, this is also the most general sparse matrix representation, in other words, it does not focus on specific types of sparse matrices, such as diagonal, jagged, etc. The Compressed Column Storage (CCS) format is another form of representing matrices which is the transpose format of the CRS. Another format specific to the diagonal sparse matrices is the Jagged Diagonal Storage (JDS) format which is commonly used in GPU systems [7]. In this format, the strategy is to use the diagonals to represent the minimum number of nonzero elements in a row. The Transpose Jagged Diagonal Storage Format (TJDS) is based on the traditional JDS format, and has been shown to provide superior performance [14]. The performance edge is achieved by removing the permutation step required in the JDS format, a step which also decreases the storage space needed.

In the STAPL library we use the Compressed Row Storage (CRS) format to implement the storage component of our parallel sparse matrix container, because it brings out most of the benefits of having a sparse matrix container. The CRS Format is a good choice for our purpose because it is the most general format, and it can be applied to used for any of the sparse matrix algorithms available. More details on the implementation of the STAPL parallel matrix container are given in Section 3.

2.3 Parallel Matrix Multiplication Algorithms

Parallel matrix multiplication algorithms have been an active area of research for decades, and since they are very important in scientific studies, there is constant research for improvement. Parallel matrix multiplication algorithms are characterized with data distribution, underlying memory architecture, and the parallel environment where they will be executed. There are matrix multiplication algorithms that

are specific to shared memory, distributed memory, or GPU machines. In this thesis, we will focus on parallel matrix algorithms that are suitable for shared and distributed memory implementations. The rest of this section will give a brief introduction to the details of parallel matrix multiplication, highlighting design decisions made by different algorithms, and it will conclude with a detailed explanation of the core algorithm for parallel matrix multiplication which is used in state of the art algorithm [13].

Assume we have three matrices A, B, and C all of size n by n. The product $C = AB$ is defined as $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ where n is the number of columns of A and rows in B. The naive algorithm for implementing matrix multiplication requires n^3 multiply operations and $n^2(n-1)$ add operations, in the case of square matrices [13]. The algorithmic challenges of implementing efficient parallel matrix multiplication are task placement, and communication of partial results. These issues are complicated by the lower level details of data-distribution and communication aggregation.

In Figure 2.1 a diagram of naive matrix multiplication is shown, where each value of matrix C is computed in one step. In other words, the final value of each C index is calculated at one time. On the other hand, if we look at the SUMMA example in Figure 2.2, we see that the calculation is structured differently. First, an initial matrix C' is created and after each iteration it contains the preliminary values of each rank-one update. A rank-one update produces a non-final matrix C' whose values will converge to the final values after a set number of iterations. So, on each iteration, as the next set of values is created, they are added to the preliminary values, iteratively accumulating the final solution C.

There are two well known strategies for partitioning matrices in distributed memory machines. The most straightforward approach is to arrange P processors in a two-dimensional grid sized $\sqrt{P} - by - \sqrt{P}$. The basic idea here is to assign proces-

$$A \qquad B$$

1	2		5	6
3	4	.	7	8

$$C_{11} = 1 \times 5 + 2 \times 7 = 19 \qquad C_{12} = 1 \times 6 + 2 \times 8 = 22$$

19	0		19	22
0	0		0	0

$$C_{21} = 3 \times 5 + 4 \times 7 = 43 \qquad C_{22} = 3 \times 6 + 4 \times 8 = 50$$

19	22		19	22
43	0		43	50
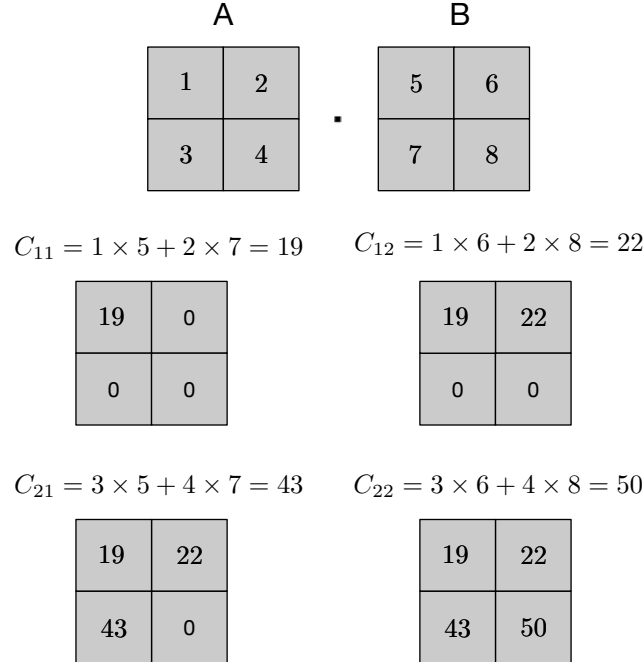
Figure 2.1: Naive matrix multiplication example.

sor (i, j) to its corresponding chunk of computations $C_{ij} = \sum_{k=1}^{\sqrt{P}} A_{ik} B_{kj}$. Another method of distributing the data can be done in a 3 dimensional setting ($\sqrt[3]{p}$-by-$\sqrt[3]{p}$-by-$\sqrt[3]{p}$). In this case a processor (i, j, k) is assigned the values of matrix A_{ik} and B_{kj} and is responsible for calculating the values of C_{ij}. The 3 dimensional algorithm proves to be more balanced and produce fewer communications than the 2 dimensional algorithm [10].

Naive implementations and focus on distributions does not always yield the best performance. The number of times values are copied from one location to another, remote reads, and synchronization, all add to the complexity and cost of perfecting a parallel matrix multiplication algorithm. Two of the most well known dense-matrix multiplication algorithms are Cannon's [3] and Fox's [4]. These algorithms suffer from some disadvantages. The first problem with these algorithms is that they
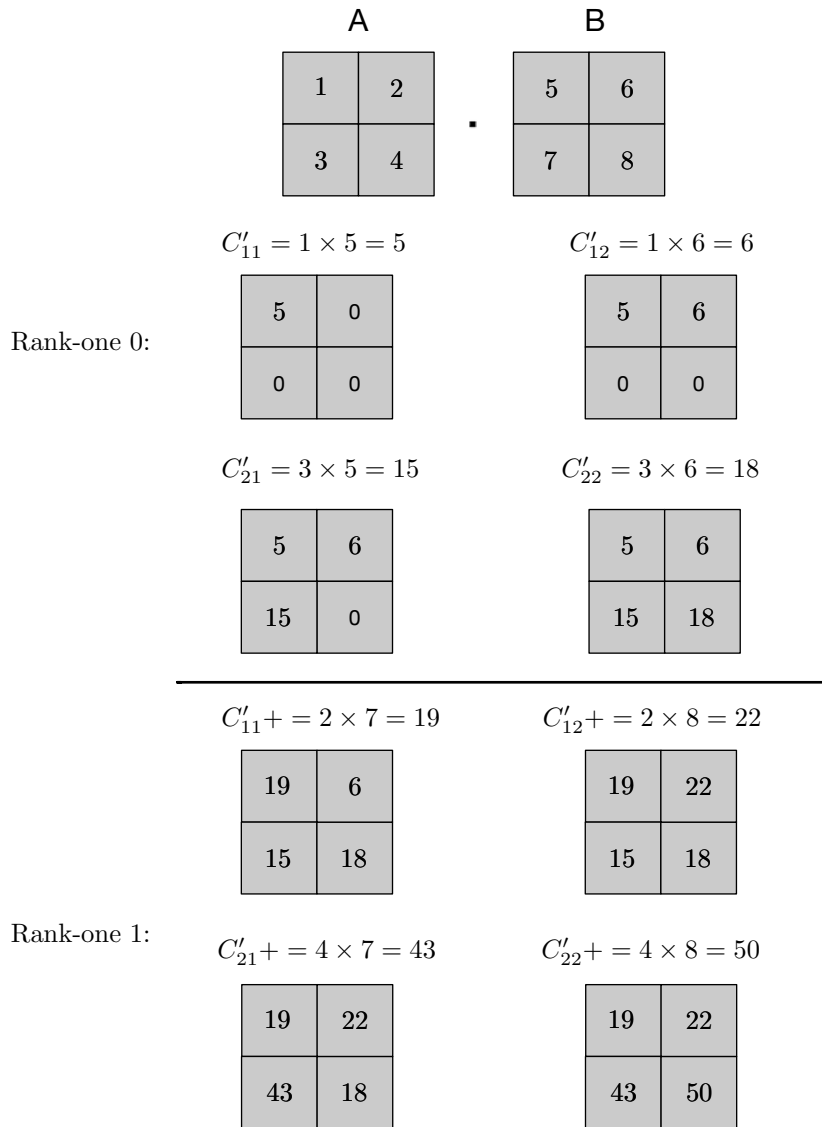
Figure 2.2: SUMMA rank-one illustration.

assume matrices with perfect square dimensions ($row = col = \sqrt{p}$). Removing such a constraint from these algorithms is nontrivial. Another issue with these algorithms is when one of the dimensions of the matrix becomes too small. In this case, a large amount of remote-reading occurs, degrading overall algorithm performance.

An algorithm that overcomes the shortcomings of Fox and Cannon algorithms is SUMMA - A Scalable Universal Matrix Multiplication Algorithm [13]. The SUMMA algorithm performs better than its counterparts because it is more general, simpler, and more efficient. This algorithm assumes that the parallel computer processors are arranged in a $r \times c$ mesh where r is the number of rows, and c is the number of columns, therefore, the number of processors $p = rc$. The chunk of data and tasks that correspond to a processor P_{ij} are indexed by row and column data (i, j). The SUMMA algorithm proceeds as a sequence of rank one updates and through multiple iterations accumulates partial results to find the final values.

Consider the matrices A, B, and their product C, of size r by c. Calculating the initial value of C'_{ij} requires multiplying $C_{ij} = \sum_{l=0}^{k-1} a_i^l b_l^j$, where $a_i^l = A_{i,0}...A_{i,k-1}$ and $b_l^j = B_{0,j}...B_{0,k-1}$, and k is the number of rows in A and columns of B. Therefore, we can see that the matrix-matrix multiplication can be achieved as an iterative solution by computing C' values, until the final C values have been accumulated.

The SUMMA technique is to parallelize the rank-one updates, allowing us to calculate C' values in parallel, and eventually converging to the final solution. This approach of SUMMA allows us to have a minimal amount of remote-reading and copying, because the data decomposition is performed in a way such that the needed data is always local assuming blocks of A and B are rotated between processors. In the rank-one updates all the values of A_i are assigned to the row i and all the values of B_j will reside in node column j. The SUMMA discusion is continued in Section 4.2.

3. PARALLEL SPARSE CONTAINERS IN STAPL

3.1 STAPL `pContainer` Framework Overview

The goal of STAPL is to simplify the process of developing parallel programs. Libraries such as STL, BGL, and MTL provide sequential containers that make programming easier for developers [12]. Similarly, STAPL provides parallel containers, in the literature also known as `pContainers`, which are objects shared between multiple processors and provide parallel methods that can be called in concurrent fashion [12].

STAPL containers are data structures that are globally addressable, independent of the distribution, which means that the programmer does not have to manage and control the distribution of data in a parallel system if they do not wish to do so. Furthermore, they are easily extendable and composable to create intricate data-structures which support nested parallelism and that allow better exploitation of data locality [12].

STAPL containers hold a finite number of elements and use the runtime of STAPL to invoke asynchronous calls to remote locations to implement their operations. There are various types of parallel containers in STAPL categorized into static containers, including arrays and matrices, dynamic containers, including vectors and lists, and associative containers, including sets and maps, and relational containers, including graphs.

In Figure 3.1 we give a visual overview of the parallel container framework in STAPL by showing the framework components used in a container instance. On the top level, we have the container directory, which is a globally addressable distributed register that provides information on the mapping of which processor stores an ele-
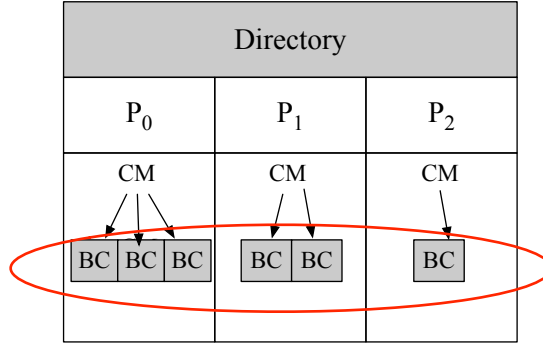
Figure 3.1: Organization of the `pContainer` framework.

ment with a given id. Every processor has its own container manager (noted CM in Figure 3.1), which is responsible for storing the base containers (noted BC in Figure 3.1), and a registry for mapping element id to base container id. The base containers can be sequential or parallel containers and they provide interfaces and methods for interacting with the data. The STAPL container framework provides a collection of base class implementations that can be utilized and extended to develop new containers by using inheritance, specialization and composition. Therefore, the STAPL parallel container framework can be used to generate wrappers for any sequential base container that provides enough information to use it in a parallel environment [12].

Our work implements a parallel sparse matrix container using the STAPL container framework.. This container provides an interface that includes element access and update operations, along with methods that provide metadata of the container (e.g., size, number of nonzeros etc.). It also provides templated parameters that can be used to specify the data distribution and traversal of the matrix. Additionally, we provide matrix-matrix algorithms and operations that include parallel matrix multiplication, addition, assignment, and subtraction that can be used to process these matrices. These algorithms are implemented using the STAPL skeleton framework.

3.2 STAPL Sparse Matrix

When designing a parallel sparse container one must consider the machine environment, load balancing, minimizing communication, and maximizing performance of each processor. The rest of this section will describe the approach used to design a parallel sparse matrix container, the details of implementation, and sparse matrix operations..

Sparse matrices differ greatly in terms of space efficiency and algorithms from dense matrices. Therefore, there are benefits to having a sparse matrix container in the STAPL library. The main difference between the two types of the matrices is the size of the storage required to store nonzero elements. With sparse matrices we can save a great amount of memory in cases with few nonzeros, and thus, be able to run much larger problems. Further, having sparse matrices in STAPL means that users can actually develop algorithms which are specific to sparse computations.

Using the views in STAPL, specifically the use of packed dense domains to describe sparse containers, we were able to run dense-specified algorithms on sparse matrices and vice versa. More detail for this is given in the experimental results section.

When designing the STAPL sparse matrix, we had the following goals in mind.

Scalable Performance. We want the container to show scalable performance for shared and distributed memory systems in parallel execution. Further, the container methods that are implemented using runtime facilities whose performance has been demonstrated.

Thread Safety. At no point in time, the programmers that will use the STAPL sparse matrix will have to worry about data locality or consistency issues. The STAPL sparse matrix is integrated in the library so that the STAPL runtime system takes care of the parallel environment challenges of concurrent element access.

Unified View. The `pContainers` are globally addressable. In other words, the programmer does not have to worry whether the element in question is local (i.e. in shared memory) or remote (i.e. in distributed memory) on another node of the system. All the internal computations required to handle these cases are already taken care of by the library so the programmer can focus on algorithm implementation. However, if the programmer is interested in such details, they have the ability to customize those components of the container.

Composition. The sparse matrix may have elements that are containers, and itself may be used as the element of another container. For example, a user might use an array, for which the data type is a sparse matrix, and vice versa. This is fairly straightforward, and all the programmer has to do is define the types.

As mentioned in the previous section, the STAPL `pContainer` framework allows integrating any type of external containers to serve as the base containers. This is the approach we took to implement the sparse matrix. We used the Matrix Template Library [11] sparse container in the Compressed Row Format (CRS) to implement the STAPL sparse matrix. Integrating this in STAPL requires adding a new class wrapping the MTL instance to meet the API required of the base container. Then, through this wrapper, we are able to write STAPL methods that will reflect the methods of the underlying container.

In Table 3.1 we have expressed the associated types common to all of the STAPL `pContainers`. Each `pContainer` has an associated distribution, mapper, and partitioner. The distribution is responsible for distributing the elements of a container, and it interacts with the directory which is globally addressable. The mapper and the partitioner, are the components which are used to specify the distribution in question.

14

Table 3.1: `pContainer` associated types.

Value type	C::value_type	Object type stored in the container.
Iterator type	C::iterator	Iterator type used to iterate through a container's elements.
Reference type	C::reference	Reference to the value type of the container.
Partition type	C::partition_type	The type of the container partitioner that maps elements to partitions.
Domain type	C::domain_type	The type of the domain of GIDs
GID type	C::gid_type	The type of the GID
CID type	C::cid_type	The type of the CID (base container ID)
Mapper type	C::mapper_type	The type of the mapper that maps partitions to locations.
Base container type	C::component_type	The type of the underlying base container

Table 3.2: `pContainer` valid expressions.

c.size()	Returns the global size of the container
c.get_element(i)	Returns the element at GID i
c.set_element(i, x)	Sets the element at GID i to the value x
c[i]	Returns a reference to the value at GID i
c.begin()	Iterator to the first element
c.end()	Iterator to one past the last element

To implement the STAPL sparse matrix we combine the dense domain with a sparse base container storage and were able to re-use many of the elements of the STAPL dense matrix. The distribution of the STAPL matrix interacts with the compressed-row-format container and gets access to only the non-zero values. In other words, the loosely-coupled and generic design approach in the `pContainer` framework allows us to easily integrate the compressed row format matrix, and at the

same time get its benefits. Most of the interfaces that are used for the dense matrix can be reused in the sparse version without any need to change. The distributions and partitioning algorithms behave the same, only that in the sparse case, they have to deal with a smaller number of elements.

In Table 3.2 we present the methods common to all STAPL `pContainers`. The methods provide a unified view for the programmer to access and modify elements in the container, as well as other information such as size and iterators. Internally, these methods interact with the STAPL runtime system [8], to fulfill the requests.

```
--------------------------------------------------------------------------
Command:          ./dense_mtl_test_100000_10000
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.15921
--------------------------------------------------------------------------

    GB
3.730^                         ####################################
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
    |                       #
  0 +------------------------------------------------------------->GB
    0                                       7.459
```
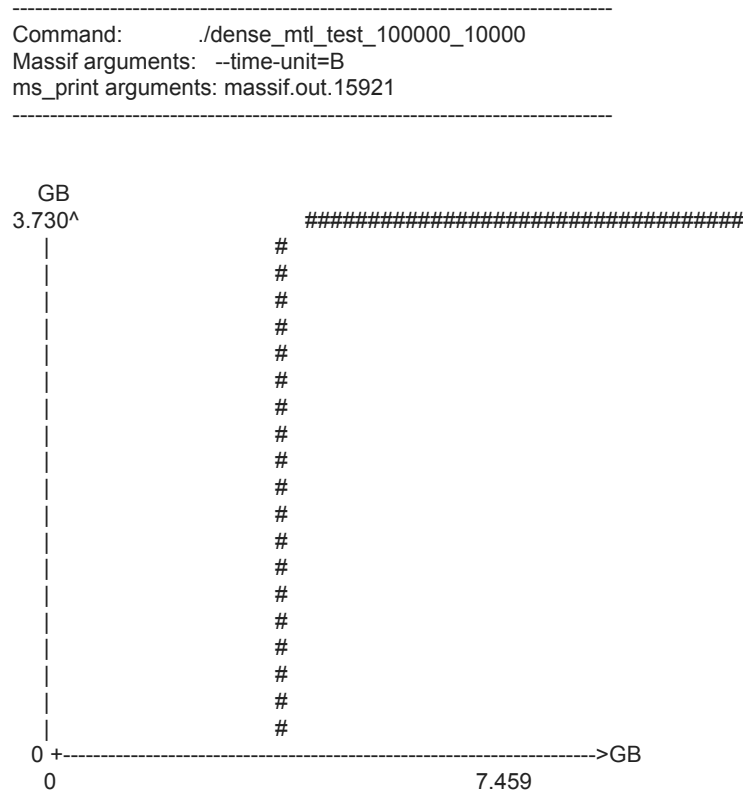
Figure 3.2: Memory utilization for STAPL dense matrix of size 100,000×10,000.

16

We used Massif, a memory profiler, to show the significance of memory usage improvement achieved by the STAPL sparse container. Massif is a tool that graphs the memory usage of a program during its execution. In our experiment, we created a program that initializes a STAPL dense matrix container of ints and a STAPL sparse matrix container of ints, of size 100,000x10,000 with only 5 nonzero elements.
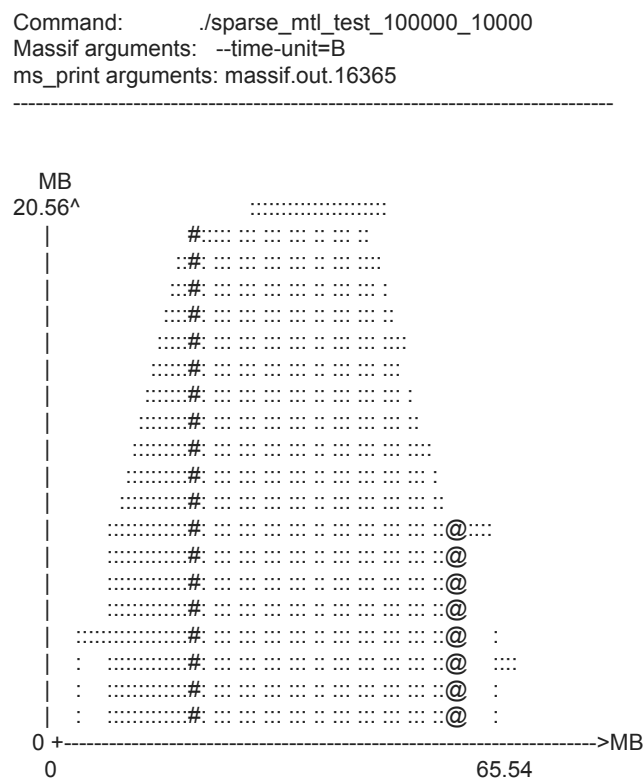
```
--------------------------------------------------------------------------
Command:            ./sparse_mtl_test_100000_10000
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.16365
--------------------------------------------------------------------------

    MB
20.56^                         ::::::::::::::::::::
     |              #:::: ::: ::: ::: :: ::: ::
     |             :#: ::: ::: ::: :: ::: ::::
     |            :::#: ::: ::: ::: :: ::: ::: :
     |           ::::#: ::: ::: ::: :: ::: ::: ::
     |          :::::#: ::: ::: ::: :: ::: ::: ::::
     |         ::::::#: ::: ::: ::: :: ::: ::: :::
     |        :::::::#: ::: ::: ::: :: ::: ::: ::: :
     |       ::::::::#: ::: ::: ::: :: ::: ::: ::
     |      :::::::::#: ::: ::: ::: :: ::: ::: ::::
     |     ::::::::::#: ::: ::: ::: :: ::: ::: ::: :
     |     ::::::::::#: ::: ::: ::: :: ::: ::: :: ::
     |    :::::::::::#: ::: ::: ::: :: ::: ::: ::: ::@::::
     |    :::::::::::#: ::: ::: ::: :: ::: ::: ::: ::@
     |    :::::::::::#: ::: ::: ::: :: ::: ::: ::: ::@
     |    :::::::::::#: ::: ::: ::: :: ::: ::: ::: ::@
     |  ::::::::::::::#: ::: ::: ::: :: ::: ::: ::: ::@   :
     | : ::::::::::::#: ::: ::: ::: :: ::: ::: ::: ::@   ::::
     | : ::::::::::::#: ::: ::: ::: :: ::: ::: ::: ::@   :
     | : ::::::::::::#: ::: ::: ::: :: ::: ::: ::: ::@   :
   0 +----------------------------------------------------------------->MB
     0                                             65.54
```

Figure 3.3: Memory utilization for STAPL sparse matrix of size 100,000×10,000.

In Figure 3.2 we can see the output of Massif for the STAPL dense matrix where the peak memory usage is over 3.730 GB. The same experiment when ran with a STAPL sparse matrix in Figure 3.3 utilizes only 20.56 MB. This enables us to run

17

programs of sparse matrices with data sizes that are not possible run with dense matrices. More details for this result are provided in Section 5.

In addition to the methods described in Table 3.2 we implemented parallel matrix operations for addition, subtraction, assignment and scale. These operations are performed by modifying the resulting matrix C with the input matrices. For example, if we are adding matrix A and B, then we first perform the operation $C = A$, and finally $C+ = B$. The same methodology is used in implementing subtract, and assign. Another operation we have implemented is the scaling of a matrix. The approach taken on the scale implementation is $C = A * k$ where k is a scalar.

In Section 5 we present experimental results of the matrix operations implemented for the STAPL sparse matrix, as well as a comparison with the dense matrix.

4. SUMMA IN STAPL

4.1 Algorithmic Skeletons in STAPL

Writing parallel programs proves to be much more difficult than sequential programs due to the intricacies of concurrency and data access, therefore for many years now, easy programmability of parallel applications has been a continuous research topic [15]. Data-flow programming provides an intuitive way of visualizing parallel and collective operations. A data-flow graph of a specific program, also known as a task-graph, represents the flows of data and the operators invoked on that data when that particular program is run. In a data-flow graph, the edges are flows of data, and the vertices are tasks or functions to be performed on that data. In Figure 4.1 we can see an example of a data-flow graph which is a program that executes a numerical operation.

The STAPL programming model is based on the data-flow concept and it uses algorithmic skeletons to define task-graphs of parallel programs. Algorithmic skeletons are presented to the programmer as higher-order functions and operators which can
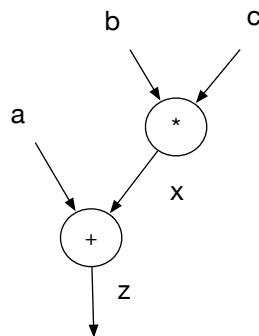
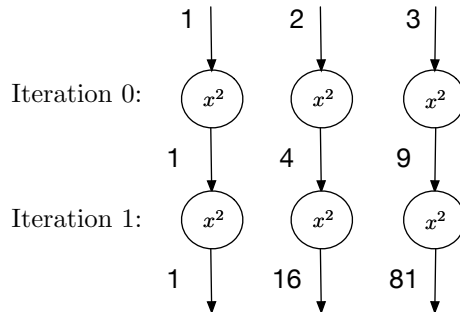Figure 4.1: Data-flow example of numerical operation.

Figure 4.2: Example of $map(x^2)$ skeleton with two iterations.

be used to specify data dependencies and steps to solve a particular problem [15]. After the task-graph representation is generated using algorithmic skeletons, it is executed using the STAPL data-flow engine knowns as the PARAGRAPH and the STAPL Runtime System [8]. The rest of this section gives an introduction to STAPL Algorithmic Skeleton framework and the specific tools that are used to define algorithmic skeletons. In Section 4.2 we provide a detailed description of the parallel implementation of SUMMA using algorithmic skeletons.

In Figure 4.2 we present the *map* skeleton, which takes a collection of data (e.g. array) and performs a unary operation on each of the elements. In this particular example, the operator invoked is the square operator. The figure also illustrates the iteration in the skeleton, which is discussed in the following paragraphs.

Parametric Dependency. Represents the finest grain of a task where a simple operation is executed in a function vertex of the task graph. In the $map(x^2)$ example in Figure 4.2 we have three instances of the parametric dependences on each iteration starting from each index of the collection. In symbolic notation, the parametric dependence of the *map* skeleton can be defined as $map\text{-}pd(x^2) \equiv \{< i >\rightarrow< i >, x^2\}$ [15].

Elem. The parametric dependency is defined over an index. In order to execute the parametric depencence operators on all indices of a collection, we have to use the elem operator. The elem operator uses a *span* defined over the domain of the collection to expand the computation. The symbolic notation of the example in Figure 4.2 for elem is $map(x^2) = elem(map\text{-}pd(x^2))$ [15].

Repeat. Some algorithmic skeletons will have multiple iterations of the same computations, very similar to loops. These situations can be expressed using the repeat operator. The symbolic notation for the Figure 4.2 is $map(x^2) = repeat(elem(map\text{-}pd(x^2)), 2)$. Figure 4.2 shows the use of repeat to apply the square operation to each element of the input twice.

Compose. The compose operator is used to combine multiple algorithmic skeletons together. For example, if we had a $map(x^2)$ and a $map(x--)$ we could compose them in one skeleton to first find the square of each element, and then decrement one from each element. The compose skeleton notation can be expressed as $compose(map(x^2), map(x--))(x) = map(x^2) \circ map(x--)(x)$.

Flows. When the composition of skeletons occurs, often times we want to relate different outputs together, or give the output of one skeleton as the input of one or more skeletons. In these cases, we use flows to define the producers and the consumers of data. The flows have to be specified explicitly for the iterations in which we have complex cases.

4.2   SUMMA Matrix Multiplication

This section provides a detailed description of how the SUMMA algorithm is expressed using algorithmic skeletons. Generating the taskgraph for this algorithm requires combinations of multiple skeletons, as well as definitions of each specific SUMMA skeleton component.
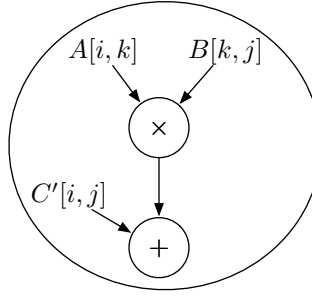
Figure 4.3: SUMMA parametric dependence.

SUMMA **Parametric Dependence.** The parametric dependence creates a simple SUMMA task by extracting the input indices from matrices A, B, and adds the result of the product in the resulting indices of C'. The indices used depend on the iteration number. Specifically, the parametric dependence specifies the finest-grain task in SUMMA which is $C[i,j] = A[i,k] \times B[k,j] + C'[i,j]$.

In Figure 4.3 we can see an illustration of the fine-grain task performed in SUMMA. This is invoked for each index, at each rank-one update, executing one more step toward the final solution. Now that we have defined the elementary task used to calculate each value of the resulting matrix, we need to expand the calculation over the two-dimensional domain. The calculation is expanded to the extent of the computation domain by using the Elem operator.

SUMMA **Two-dimensional Span.** When we distribute the tasks among processors we want it to be divided as evenly as possible. This makes our programs scale better and perform uniformly. The Span is used to allocate a balanced load to each processor. It uses the number of iterations, and sizes of matrices being multiplied to calculate each processor load. This serves us when the task-graph is generated, to partition each in its corresponding location. The formula derived in this step is also used to expand the calculation throughout the two-dimensional matrix domain.
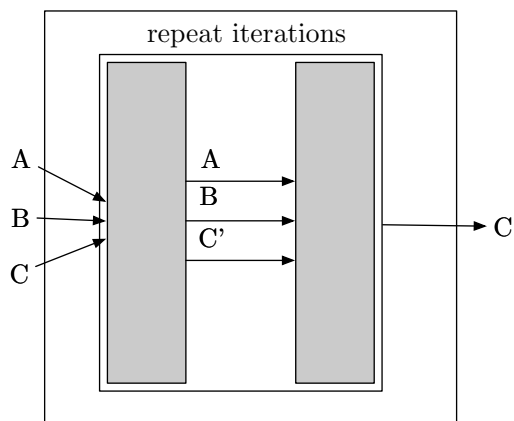
22

Figure 4.4: Skeleton composition for SUMMA.

SUMMA **Flows.** Are used to connect the different skeleton components that are combined to perform the SUMMA calculation. They need to be defined for three corner cases of the algorithm. The first iteration, the intermediate iterations, and the final iteration, as well as the special case where matrix multiplication requires only one iteration. In Figure 4.4 we show a high-level representation of the skeleton compositions involved to compute SUMMA. First we provide the inputs to the skeleton. Then, we invoke the *repeat* skeleton for a specific number of iterations, passing the C' matrix in between the iterations. When the iterations are completed, the output is copied to the final matrix C. Then, we use the STAPL skeleton framework to handle the I/O.

When multiplying two matrices of size $n \times k$ and $k \times m$ the inner dimension k defines the number of iterations needed to calculate the final result. We use k to provide the number of iterations of our skeletons. The repeat and elem operators are integrated in the SUMMA implementation. Repeat provides specification for the number of iterations, also known as rank-one updates, to calculate the final values of each index in the matrix C. Elem is used to invoke the parametric dependence
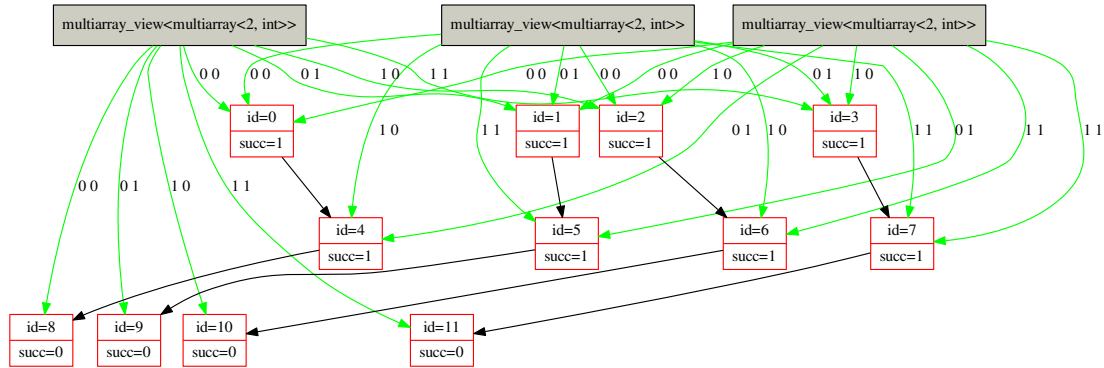
Figure 4.5: SUMMA taskgraph for multiplication of 2x2 matrices.

throughout the whole domain.

In Figure 4.5 we can see an illustration of the data-flow graph of SUMMA for multiplying two 2x2 matrices. The parts on the gray, are the inputs, the green lines, represent the flows between the tasks, and the boxes with red, are the actual tasks and their task ids. The Figure 4.5 shows the data-flow after it has been generated by the skeleton framework and the STAPL PARAGRAPH.

We specified multiple skeleton components from scratch to express the SUMMA algorithm. We also used some of the available components used for I/O such as sink. In Section 5, we provide experimental results and discuss the performance of SUMMA.

5. EXPERIMENTAL RESULTS

5.1 Machine Specifications

The experiments were run on Rain, a Cray XE6m cluster. In Table 5.1 we have illustrated hardware configurations of the cluster, it has 576 processors connected in a 2-D torus.

Table 5.1: Cray XE6m hardware specifications.

Board count	6
Nodes per board	4
Node count	24
Cores per node	32 on 12 nodes
	64 on 12 nodes
Total number of cores	576
Processor Type	64-bit AMD Opteron (Interlagos) 6272, 2.1GHz
Cache	8x61 KB L1 I-cache, 16x16 KB L1 D-cache, 8x2 MB L2 cache per core, 2x8 MB shared L3 cache
Memory	32 or 64 GB registered ECC DDR3 SDRAM per compute node
Memory per core	2 GB
Interconnect	1 Gemini routing and communication ASIC per two compute nodes.
	48 switch ports per Gemini chip (160GB/s switching capacity per chip).
	2-D torus organization

Table 5.2 displays the software specifications on rain, we used g++4.9.2 as a C++ compiler. STAPL uses many components of the C++ Boost library, for our results Boost 1.56 was used.

Table 5.2: Cray XE6m software specifications.

OS	Cray Linux Environment
Compilers	Cray g++ version 4.9.2
MPI	version 2.0
Libraries	Boost version 1.56

We provide experimental results for sparse matrices and their operations, the SUMMA computations, and the sparse SUMMA computation. Further, we compare the sparse and dense matrices in different types of inputs to test our claims for the different implementations.
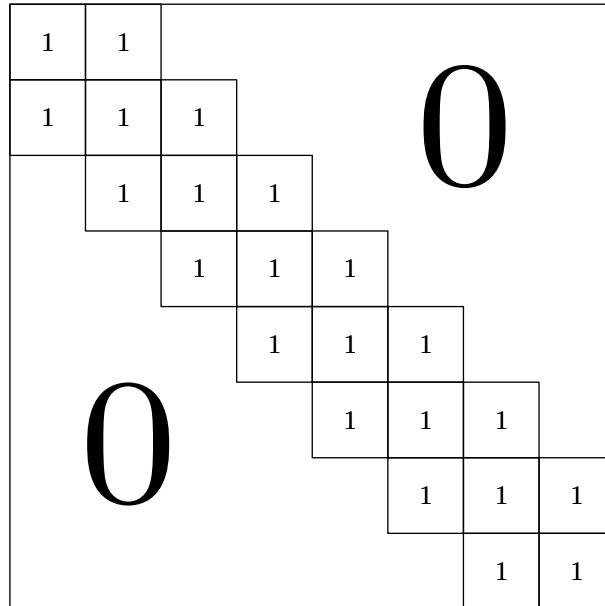
5.2 Sparse Matrix

Figure 5.1: Tri-diagonal sparse matrix illustration.

Sparse matrix results were conducted on three different types of matrices. Choosing the type of inputs to run for the sparse matrix container we wanted to represent cases that are applicable in real world computations.

The first type of matrix we used was a Tri-Diagonal matrix showed in Figure 5.1. Tri-diagonal matrices [6] are used to solve linear equations and have high applicability in statistical physics. The Tridiagonal matrix represents a very unbalanced input therefore it was chosen to see the behavior of the container.

Algorithm 1 Pseudocode for scale-free edge generation

1: **procedure** GENERATEEDGE(i)
2:     $r \leftarrow 2i + 1$
3:     **repeat** $r := h(r)$ until r is even
4:     **return** $(\lfloor i, d \rfloor, \lfloor r, 2d \rfloor)$

Algorithm 2 Pseudocde for random matrix nonzero generation

1: **procedure** GENERATERANDOMMATRICES(n, A, B)
2:     $nonzerosA \leftarrow random\_int(0, n^2 + 1)$
3:     $nonzerosB \leftarrow random\_int(0, n^2 + 1)$
4:     **for** $i = 0..nonzerosA$ **do**
5:         xA_vector.push_back($random\_int(0, n)$)
6:         yA_vector.push_back($random\_int(0, n)$)
7:     **for** $i = 0..nonzerosA$ **do**
8:         $A(x\_vector[i], y\_vector[i]) = 1$
9:     **for** $i = 0..nonzerosB$ **do**
10:         xB_vector.push_back($random\_int(0, n)$)
11:         yB_vector.push_back($random\_int(0, n)$)
12:     **for** $i = 0..nonzerosB$ **do**
13:         $B(x\_vector[i], y\_vector[i]) = 1$

The other type of matrices we decided to use for analyzing the performance of sparse matrices is scale-free graphs [9] because, graphs can be represented using a matrix, and they are ubiqutuous in every network analysis. Scale free matrices are generated using Algorithm 1. The d in the pseudocode represents the average degree in the graph, and the function $h()$ represents a hash function that returns a random number from $0...r - 1$. In terms of matrices, d is the average number 1's in a column. An index that has a nonzero value in the scale-free graph matrix represents an edge. For simplicity, here we used all edges of value 1. The number of nonzero values in the scalefree graph is approximately $d \times n$. We used a d parameter of 100.

The final type of matrix that we used is a random matrix, to compare the performance of sparse and dense matrices. We accepted a user input for the number of rows of the matrix, and then chose a column size randomly. Then, we generated a random number for the number of elements. Then, we created two vectors with random values, the length of the number of elements. Finally, we iterated over the random vectors and used them as indices to set the values in the matrix. The pseudocode for the generation of the random matrix is presented Algorithm 2.

In Figure 5.2 we show the execution times of the Add operation on a Tri-diagonal graph and on a Scale-free graph from 1 to 512 operations. The maximum speedup that we get on the add operation is for a Tri-diagonal graph is 217x on 512 processors. On the Scale-free graph input we get a max speedup of 236x on 512 cores. The scale free graph shows similar scalability as the Tridiagonal matrix, and is faster. The reason for that is that the data is more broadly distributed and processors get more balanced chunks of data to compute.
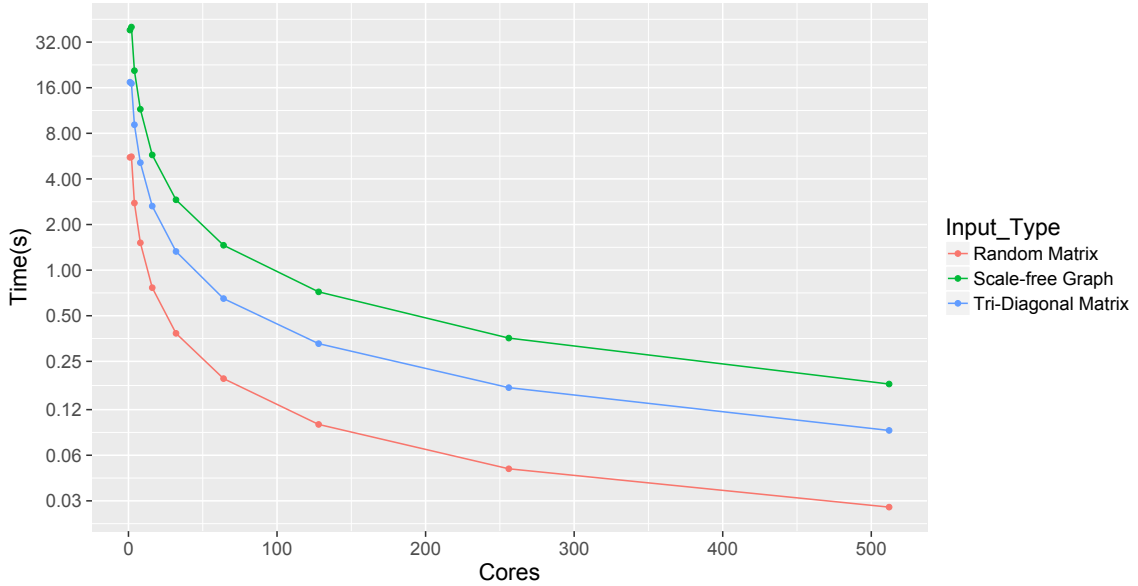
Figure 5.2: Execution times of add operation on tridiagonal, random, and scale-free graph sparse matrices of size 2×10^7 by 2×10^7.

Whereas in the tri-diagonal experiment, some processors get less data and on the same pattern, so the corner processors will be extremly load imbalanced. The random input matrix performs faster than the others because it is balanced, since every processor generates the same amount of nonzeros.

Similarly in Figure 5.3 the sparse subtract operation is shown. We achieve around 300x speedup at 512 cores.

In Figure 5.4 we show the execution times of the scale operation on sparse matrices. The scale matrix operation on the tridiagonal matrix shows around 200x speedup on 512 cores. The confidence intervals are not visible in the graph. There is a small drop from one core to two cores due to communication and synchronization. This is expected in many experiments due to the overhead of parallelism. The reason why the random matrix is the fastest is because of the perfectly balanced pattern.
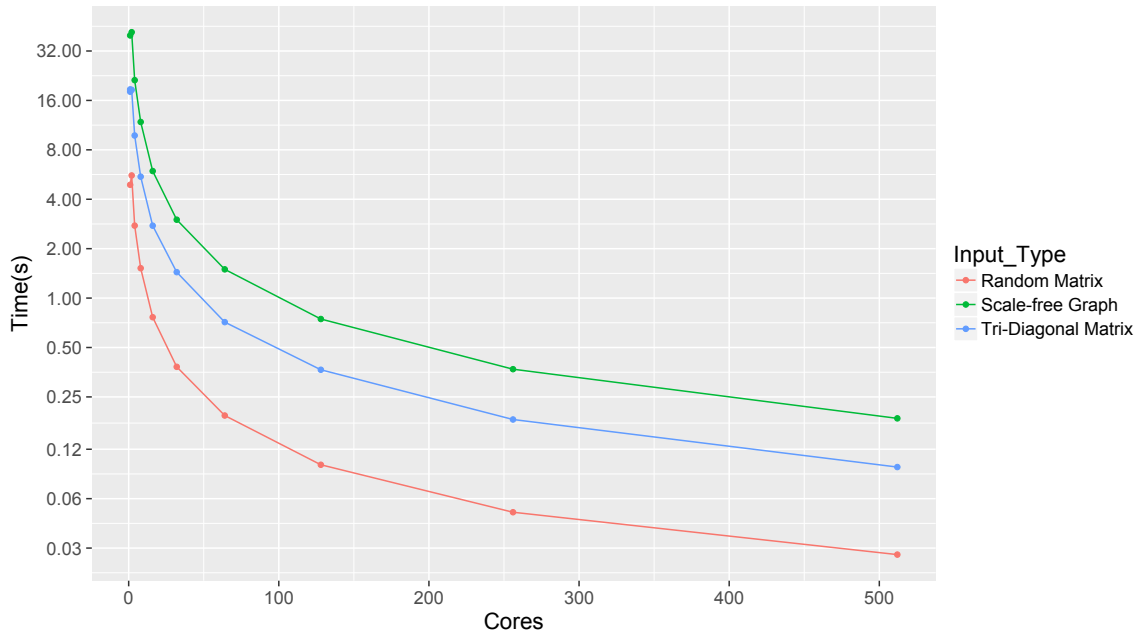
Figure 5.3: Execution times of subtract operation on tridiagonal, random, and scale-free graph sparse matrices of size 2×10^7 by 2×10^7.
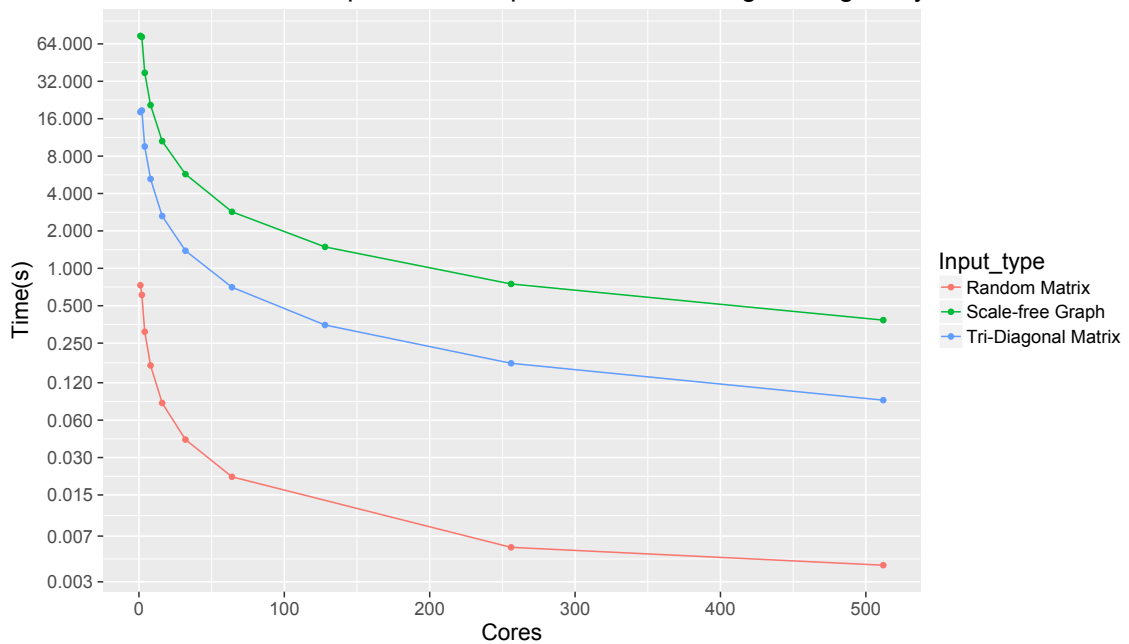

Figure 5.4: Execution times of scale operation on tridiagonal, random, and scale-free graph sparse matrices of size 2×10^7 by 2×10^7.

Each processor has the same amount of nonzeros placed in different places, which when seen from a sparse viewpoint look identical.

Generally, we see a similar trend with all the sparse matrices that we have generated since our distributions are only focused on the nonzero values. This shows that there is a consistent scalability with the CSR base container that we have chosen.

5.3   STAPL Sparse Matrix vs STAPL Dense Matrix

The STAPL dense matrix is implemented assuming no space efficiency optimizations. In the following study we show the performance difference of the two types of matrices in three different types of matrices.
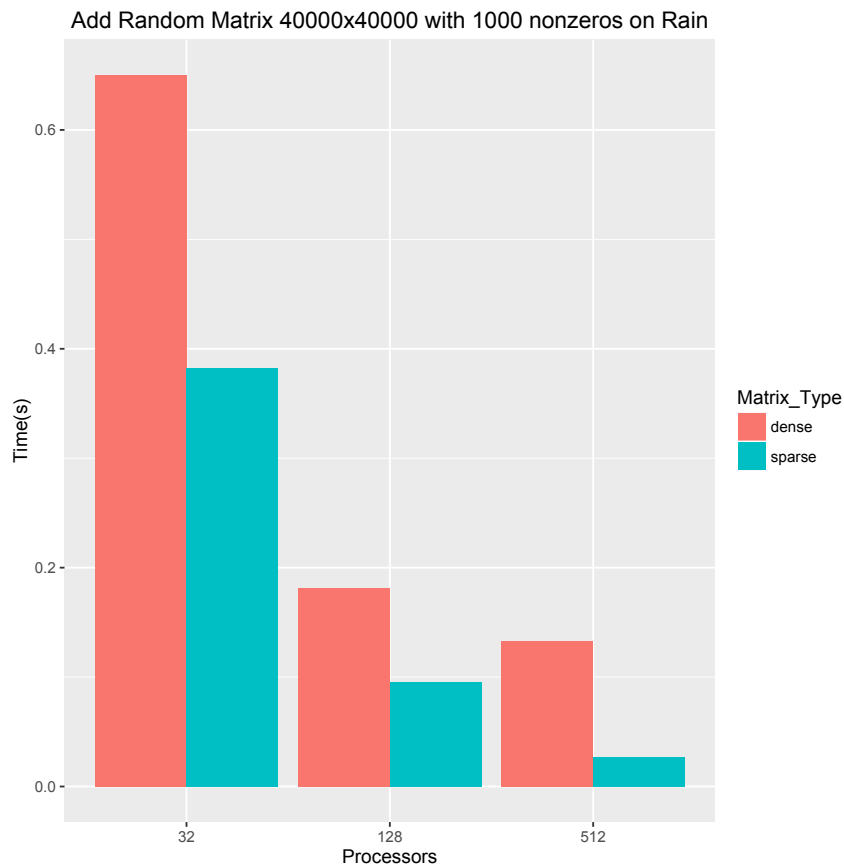

Figure 5.5: Execution times for sparse and dense matrix on a random matrix.

The random matrix generated in Figure 5.5 is created using random values for the number of elements and the indices where the nonzero values are placed. The matrix is 40000 by 40000 with only 1000 nonzero elements. We can see that the sparse matrix outperforms the dense matrix since the number of nonzero elements is very small compared to the matrix size.

Note that the indexing operator for the sparse matrix is much slower than the indexing of the dense operator, since it has to do a transformation on each value to determine where the nonzero value exists.
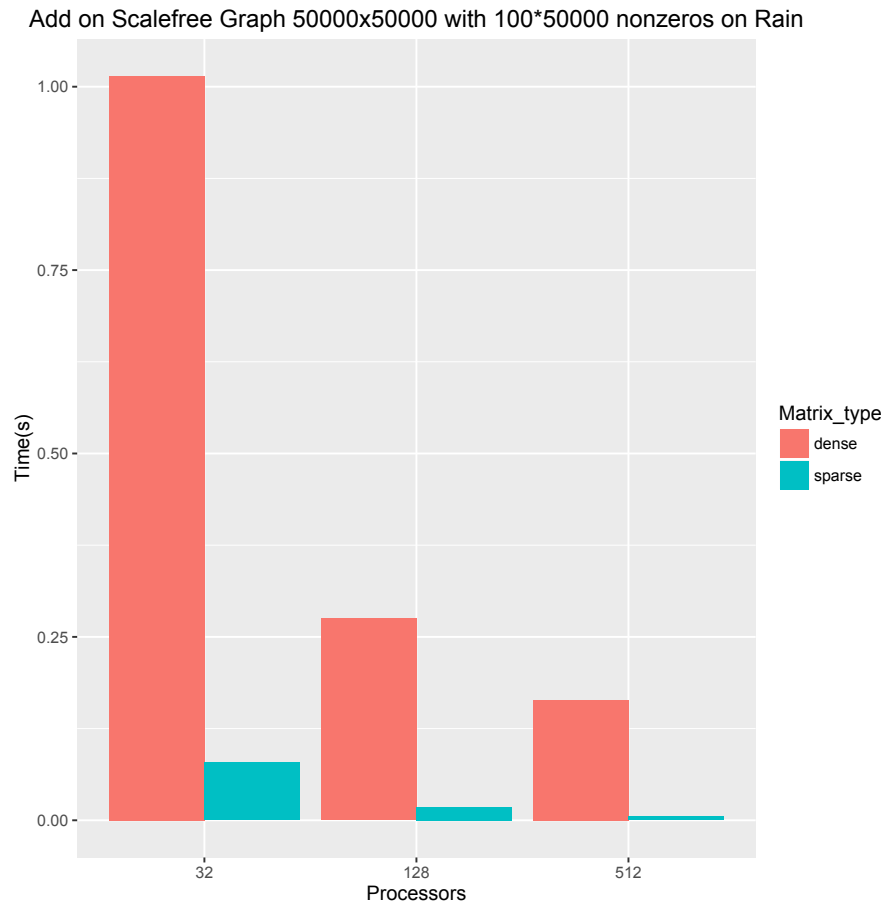

Figure 5.6: Execution times for sparse and dense matrix on a scale-free graph.

However, since the sparse matrix is only focusing on the nonzero values, it has the performance edge. Further, the sparse matrix also uses much less storage than the dense matrix.

In Figure 5.6 we show the performance difference of the STAPL sparse matrix and the STAPL dense matrix. The matrix size here is 40000 by 40000 with 1000*40000 nonzero values. The nonzero values are distributed on a scale-free graph manner. Since the number of nonzeros is a lot less than the size of the matrix, we clearly see that the sparse matrix outperforms the dense matrix. On 32 cores the sparse matrix is about 14x faster, whereas on 512 cores the sparse matrix is 32 times faster.
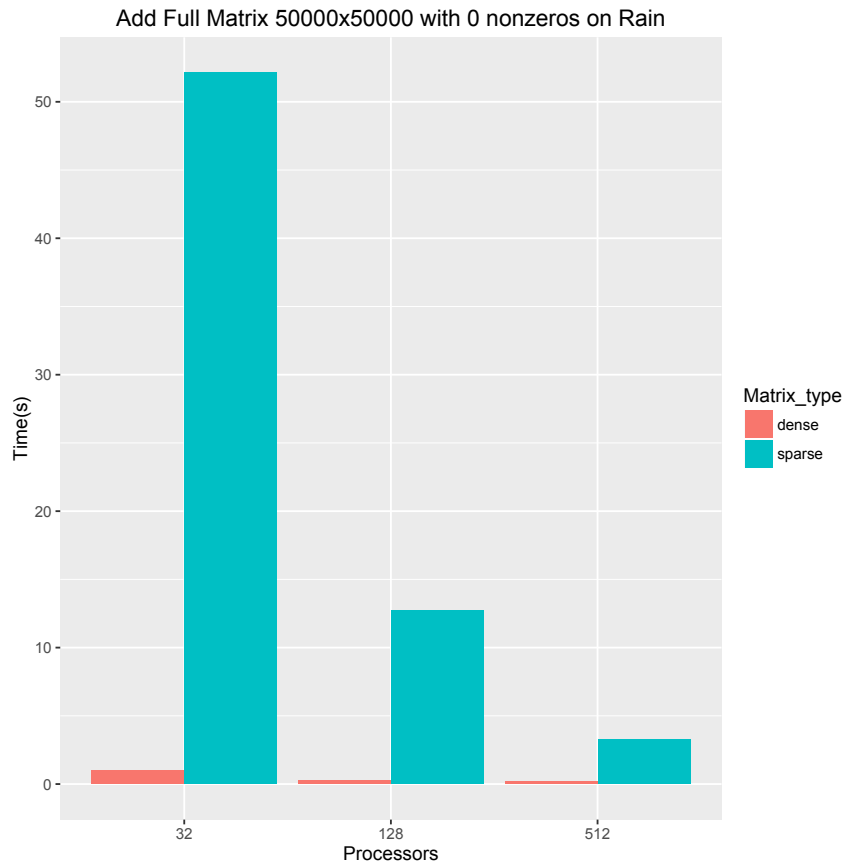

Figure 5.7: Execution times for sparse and dense matrix on a full matrix.

In the above experiments we see this behavior because the sparse matrix is focusing only on the nonzero values, whereas the dense matrix is also iterating (and storing) the zero values.While the number of nonzero elements is very small, the dense matrix is doing many computations on zeros, which are uneccessary.

In Figure 5.7 we see that the dense matrix drastically outperforms the sparse matrix. The full matrix used in this experiment is a completely filled matrix with random values. In other words, there are no-nonzero values. Since the dense indexing operator is much faster than the sparse indexing operator the dense matrix gets the performance edge. On 512 processors the dense matrix is 52x faster than the sparse matrix. Note that in this case the sparse matrix also uses more memory because of the extra data structures needed to maintain the index mapping functions. In 32 processors it is 52x faster.

In conclusion, we saw the scalability performance of the STAPL sparse matrix and we see that it shows scalability up to 512 cores on Rain, even for completely dense graphs. We also saw that the space efficiency of the sparse matrix allows us to run very large inputs. Further, we showed that the sparse matrix outperforms the dense matrix on different types of sparse matrices, even when the sparsity is around 50% nonzeros.

5.4 SUMMA

In Figure 5.8 we show the execution times for the Fine Grained SUMMA algorithm. The algorithm shows scalability of 6x at 512 processors. For each of the different types of inputs we see the similar behavior of SUMMA algorithm. It shows scalability among all inputs provided. It is clear that the overhead of task creation and management creates a huge drawback because it doesn't allow us to use large input sizes in the computation. Another factor that drastically deteriorates performance of this

implementation is the extensive amount of communication needed in a fine grained computation for each minimal task.
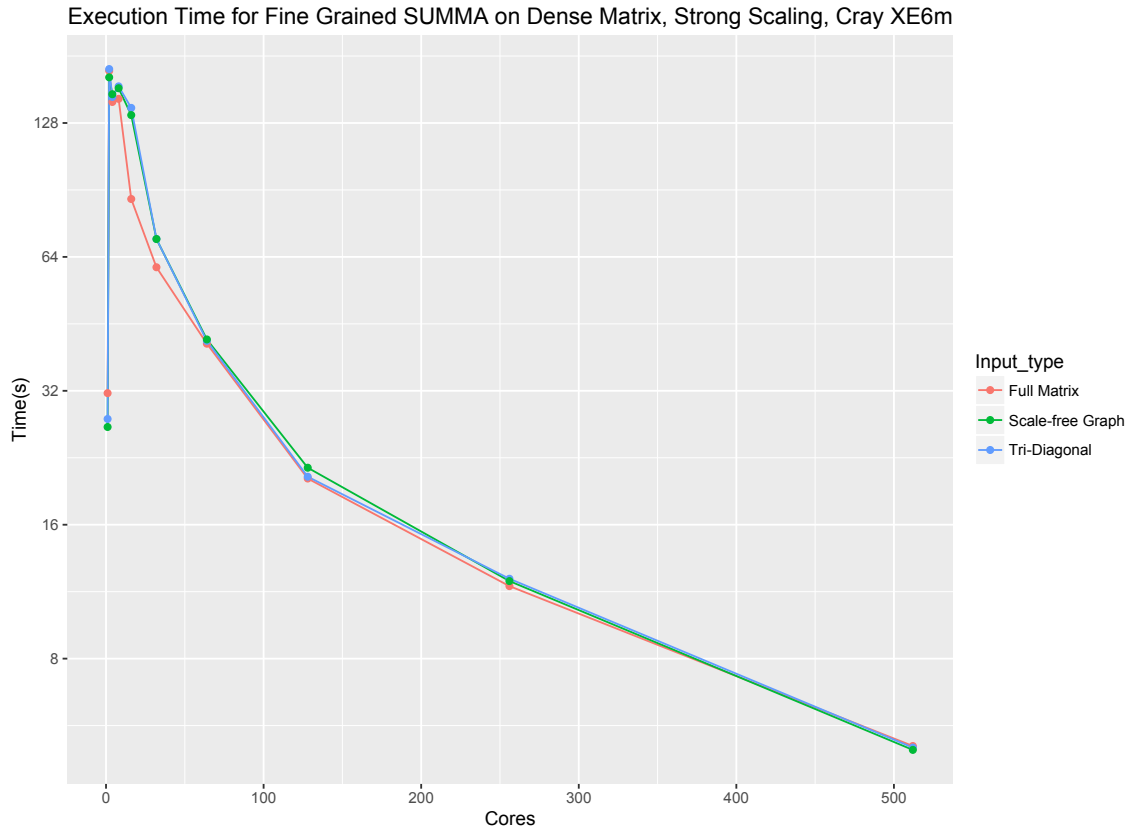
Execution Time for Fine Grained SUMMA on Dense Matrix, Strong Scaling, Cray XE6m

Figure 5.8: Fine-grained SUMMA execution time with dense matrices of size 250×250.

6. CONCLUSIONS AND FUTURE WORK

In this thesis we have developed a parallel sparse matrix container in the STAPL library and we have implemented the SUMMA matrix multiplication algorithm. These components have been added to the STAPL library. We have shown that the usage of parallel sparse containers is beneficial for any parallel library because they allow us to write sparse specific algorithms and save large amounts of storage. Further, we have provided the SUMMA skeleton which is the first step to complicated algorithms and applications developed with algortihmic skeletons. This work has opened the door to many different possibilities for research which could range in Adaptive Matrix Containers, Algorithmic Skeletons, and Data Distribution strategies. The next few sections detail the possibilities of how this work can be extended.

6.1 Parallel Hybrid Matrix Container

One possibility, which would allow the STAPL users to get most benefits of sparse and dense matrices is a parallel hybrid matrix container. The idea is to have a threshold value for which the construction of the matrix would decide whether to use a dense or sparse matrix base container. However, ideally, not at a global level, but instead on a location level. The base container instantiation occurrs at runtime, therefore to be able to initialize such container, the programmer needs to specify an option that would identify which base container to use at the instantiation of the `pContainer`.

Global-Adaptive Matrix Container. The first step of achieving this goal would be designing an adaptive matrix container, which based on a threshold, would decide whether to use a sparse or dense base container. This could easily be achieved by passing in the values of the **domain** and **number of non-zeros** to the tem-

plate arguments of the matrix. Then, the programmer would need to propagate that knowledge when deciding which base container to instantiate (on base_container_traits). This way, we would have a runtime specification of the base container at a global level. In other words, all locations would have either a sparse base container or a dense base container, but it would be the same on all locations.

Processor-Level Adaptive Matrix Container. We know that in the `pContainer` framework every location may have multiple base containers. This occurs because of the data distribution. Each base container may have a different number of elements. When a base container is initialized, one will have to determine what internal data-structure is instantiated, in this case, a sparse matrix or a dense matrix.

This will require a modification to the `pContainer` organization to support functionalities correspondent to the underlying base container types. The `pContainer` methods will have to be aware of the underlying sequential base container, and will have to mimic the correct methods.

Supporting such operations will require a completely new base container integrated in the STAPL framework. The `pContainer` for matrices that inherits from the multiarray may or may not be used, depending on the base container that is implemented. However, the specification of every method has to be defined for sparse and dense containers because they will have a different **m_data** data structure. The **m_data** data structure inside a `pContainer` refers to the underlying base container storage and is used to access/modify the underlying data. It's interface and API can be different for different base containers, therefore, the programmer must support both. Or, they must develop a way which would, at compile time, decide the paths that the execution will take, depending on the underlying base container.

Having such a container will increase the size of the executable drastically because it will have to generate code for two different types of sequential containers, and their

37

parallel counterparts in STAPL.

6.2 Data Distribution Strategies

The STAPL sparse matrix has opened the doors to multiple available projects for research in the domains of `pContainers` and parallel algorithms. Support for efficient parallel algorithms tailored for sparse matrices has now a foundation, and there are many ways in which it could be improved.

A large variety of data distribution algorithms can be implemented in STAPL to improve the performance of sparse algorithms drastically. Further, they can and should be combined with data-structures that best fit the needs of the algorithm. Again, this opens the door for multiple adaptive algorithms tailored for specific sparse algorithms.

REFERENCES

[1] An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[2] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Stapl: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM.

[3] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.

[4] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors. Vol. 1: General Techniques and Regular Problems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[5] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.

[6] Irina Mazilu, Dana Mazilu, and H Thomas Williams. Applications of tridiagonal matrices in non-equilibrium statistical physics. *Electronic Journal of Linear Algebra*, 24(1):3, 2012.

[7] Eurpides Montagne. An alternative compressed storage format for sparse matrices. In *Computer and Information Sciences: ISCIS 2003, LNCS 2869, Springer-Verlag*, 1992.

[8] Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. Stapl-rts: An application driven runtime system. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 425–434, New York, NY, USA, 2015. ACM.

[9] Peter Sanders and Christian Schulz. Scalable generation of scale-free graphs. *Inf. Process. Lett.*, 116(7):489–491, July 2016.

[10] Martin D. Schatz, R. A. Van De Geijn, and Jack Poulson. Parallel matrix multiplication: A systematic journey. *(submitted to) SIAM Journal on Scientific Computing*, 2014.

[11] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, pages 59–70, 1998.

[12] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL Parallel Container Framework. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 235–246, San Antonio, Texas, USA, 2011.

[13] Robert A. van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. Technical report, Austin, TX, USA, 1995.

[14] A. Yazici and C. Sener. *Computer and Information Sciences – ISCIS 2003: 18th International Symposium, Antalya, Turkey, November 3-5, 2003, Proceedings.* Number v. 18 in Lecture Notes in Computer Science. Springer, 2003.

[15] Mani Zandifar, Nathan Thomas, Nancy M Amato, and Lawrence Rauchwerger. The stapl skeleton framework. In *Languages and Compilers for Parallel Computing*, pages 176–190. Springer, 2014.