A HIERARCHICAL SYSTEM VIEW AND ITS USE IN THE DATA DISTRIBUTION OF COMPOSED CONTAINERS IN STAPL

A Thesis

by

JUNJIE SHEN

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-Chairs of Committee,	Lawrence Rauchwerger
Committee Members,	Jim E. Morel
Head of Department,	Dilma Da Silva

December 2017

Major Subject: Computer Science

ABSTRACT

In parallel programming, a concurrent container usually distributes its elements to all processing units (locations) equally to maximize the processing ability. However, this distribution strategy does not perform well when we apply nested parallel functions on a composed concurrent container, such as a concurrent vector of vectors or a concurrent map of lists. The distribution of the inner concurrent containers across the system will mess up the locality of the elements in the composed containers, generating a lot of inter-process communication when the nested parallel operations are called to access the container's elements. As the hierarchy in modern high performance computing (HPC) systems become large and complex, a large amount of inter-process communication, especially those between two remote processing units (such as two cores on different nodes), will have dramatic negative impact on the performance of the parallel applications.

In this thesis, we introduce a hierarchical system view that represents the topology of the processing units in a HPC system, and use it to guide the distribution of the composed concurrent containers. It reduces the number of processing elements involved in storing in the inner concurrent containers, which reduces memory usage and improves construction time. It also reduces the amount of inter-process communication by improving the locality of the elements when we apply nested parallel functions on a composed concurrent container.

To evaluate our approach, we implement two concurrent associative multi-key containers, multimap and multiset, in the Standard Template Adaptive Parallel Library (STAPL), and use the hierarchical system view on the distribution of composed 2D and 3D containers. Finally, we show great improvement on both the construction

time and the execution time of the nested parallel functions with various numbers of cores and hierarchies.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor Dr. Nancy M. Amato for her guidance through my undergraduate and graduate studies and life. Without her tremendous support, I would not get so much knowledge and progress on my research.

Also, I want to thank my co-advisor Dr. Lawrence Rauchwerger for his suggestion and recommendation on the direction of my research. I feel grateful that I can work in Parasol Laboratory under he and Dr. Amato's guidance.

I would like to thank Dr. Jim E. Morel from Nuclear Engineering department. The Kripke miniapp developed as part of the CERT project he leads provided me with a lot of experience related to my research.

At last, I want to thank the research staffs members Dr. Timmie Smith, Dr. Nathan Thomas and Dr. Milan Hanus, my fellow students, Adam Fidel and Alireza Majidi, and former students, Mani Zandifar, Ioannis Papadopoulos, Harshvardhan, Dielli Hoxha and Tyle Biehle for sharing the knowledge on STAPL and helping me solve the difficulties during my research. Thanks also to my fellow students Glen Hordemann, Brandon West, Daniel Latypov, Francisco Coral, and Priya Arora.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Nancy M. Amato (advisor) and Professor Lawrence Rauchwerger (co-advisor) of the Department of Computer Science and Engineering and Professor Jim E. Morel of the Department of Nuclear Engineering.

All work for the thesis was completed independently by the student.

Funding Sources

This work was made possible in part by the National Science Foundation (NSF) under Grant Numbers CCF-0833199, CCF-1439145, and CCF-1423111, and by the Department of Energy (DOE) under Grant Number B575363. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the NSF or the DOE.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

As the processing capacity of a single core is limited and the amount of data to process is increasing steadily, High Performance Computing (HPC) [1] becomes the main way to solve large problems efficiently. HPC refers to the parallel processing of data on a system with a massive number of processors that delivers a much higher performance than a general-purpose computer. The concurrent container is one of the main components in parallel programming frameworks that distributes the elements it handles to different processing units (locations) to process them in parallel. Usually, the elements are divided into each location equally, however, this distribution strategy does not perform well when we apply nested parallel functions on a composed concurrent container, such as a concurrent vector of vectors or a concurrent map of lists. It is because the distribution of the inner concurrent containers across the entire system will mess up the locality of the elements in the composed containers, generating a lot of inter-process communication when a nested parallel function is called to access the container's elements. As the hierarchy in modern HPC systems become large and complex, a large amount of inter-process communication, especially those between two remote processing units (such as two cores on different nodes), will have a dramatic negative impact on the performance of the parallel applications.

This thesis introduces a hierarchical system view that represents the topology of the processing units in a HPC system and uses it to guide the distribution of the composed concurrent containers. It reduces the number of processing elements involved in storing in the inner concurrent containers, which reduces memory usage and improves construction time. It also reduces the amount of inter-process com-

munication by improving the locality of the elements when we apply nested parallel functions on a composed concurrent container. To evaluate our approach, we implement two concurrent associative multi-key containers, multimap and multiset, in the Standard Template Adaptive Parallel Library (STAPL) [2], and use the hierarchical system view on the distribution of composed 2D and 3D containers to show great improvement of our distribution strategy on both the construction time and the execution time of the nested parallel functions.

1.1 HPC Introduction

According to the system architectures, generally, there are two types of HPC systems: the shared-memory system and the distributed-memory system (Figure 1.1). In a shared-memory system, all the processors use the same address space and they access the shared memory with an equal priority. The advantage is that the memory access time for each processor is very fast, because they are directly connected to the shared memory. But the shared memory cannot handle a lot of processors at the same time due to its capacity and bandwidth. It is hard to have a shared-memory system with a large scale and a complex hierarchical structure, thus, our work is mainly focused on the distributed-memory system.

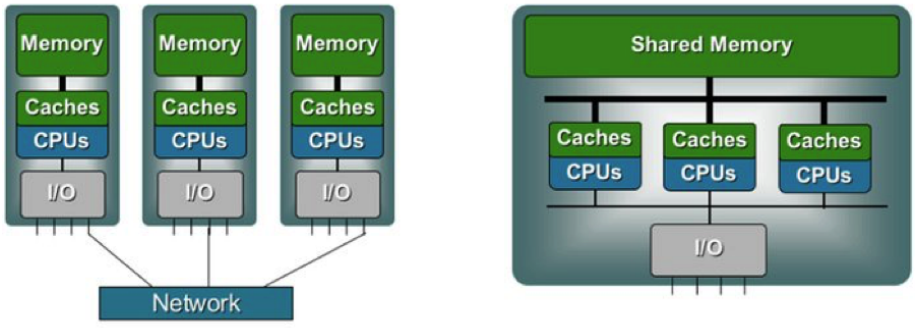

Figure 1.1: Distributed-memory System and Shared-memory System.

Unlike the shared-memory system, each processor in a distributed-memory system has its own memory and is connected with other processors by a high-speed network. Accessing elements in the memory of another processor is much slower than accessing elements in its own memory. But a supercomputer using the distributed-memory system can easily accommodate thousands of processors. These processors are not placed in a linear or arbitrary order. They are usually organized and connected in a hierarchical way (e.g., a 5D torus of nodes, each with 16 cores in shared memory with a high-speed interconnect). These levels are commonly used in a current distributed-memory system: compute nodes, dies (NUMA nodes), cores and threads.

A compute node is a big module in the HPC system, which often consists of multiple dies that are connected with memory banks. A die is usually a NUMA node that contains multiple cores with a shared cache that has uniform memory access. A core is a basic computational unit; it has its own cache and could fork multiple threads to work concurrently. A thread is the smallest unit in the HPC system; all the threads in one core have the same address space.

The data transmission speed within each level of the system hierarchy is also different. Usually, the processing units that are physically adjacent will have high communication speed. Figure 1.2 shows the speed of the hyper transport (HT) link between the non-uniform memory access (NUMA) nodes in one Cray XE6 [3] compute node from the hopper system at NERSC [3]. The communication speed within a NUMA node is much faster than that between two NUMA nodes. Moreover, even the NUMA nodes in the same compute node may have different communication speed. P0 and P1 are two NUMA nodes on the same socket, so they have faster communication speed, which is 19.2GB/s. The communication speed between nodes P0 and P2, which are located on different sockets, has 12.8GB/s communication speed. The speed of ncHT3 link between Hopper compute node is 10.4GB/s. Thus,

how to fully utilize this feature to optimize the distribution of the composed data structures becomes the main challenge of our work.

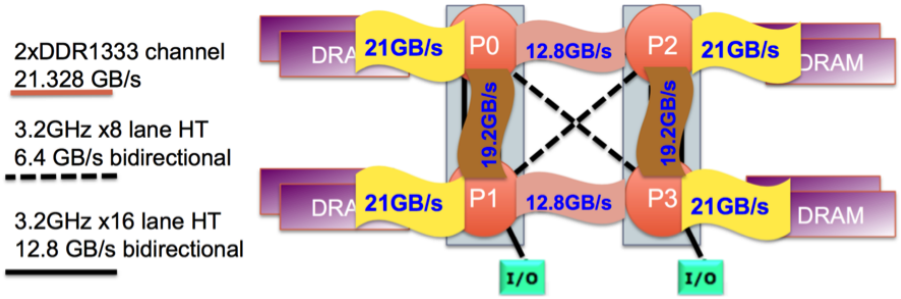

Figure 1.2: Cray XE6 Hopper Node Topology [3]

1.2 STAPL Introduction

The Standard Template Adaptive Parallel Library (STAPL) [2] is a parallel programming framework whose components help users to implement parallel applications without managing the details of data distribution and access. It is developed using the C++ programming language and provides the parallel equivalents of most algorithms and data structures (containers) found in its Standard Library (STL). Moreover, STAPL includes many other concepts and components that are useful for parallel programming, such as parallel matrix [4], graph [5]. Containers in STAPL are built using components of the Parallel Container Framework [6]. Algorithms are expressed using algorithmic skeletons [7, 8]. Its run-time system [9, 10] is responsible for the inter-processor communication and task scheduling to achieve higher load balance.

1.3 Contribution

This work provides the following contributions:

- A STAPL view that represents the topology of system hierarchy to guide the distribution of the composed concurrent containers.
- A convenient and flexible control for the distribution of the composed concurrent containers based on the system hierarchy.
- Locality and memory usage improvement for the construction of a composed concurrent container.
- Performance improvement for the nested parallel algorithms and operations that use composed concurrent containers.
- Extension of STAPL features and functionalities by implementing parallel multimap and multiset containers that use the hierarchical system view to specify their distribution.

1.4 Outline

This thesis will first discuss the related work about the distribution of nested parallelism in other parallel frameworks and the previous work in STAPL that supports nested parallelism on the composed concurrent containers (Chapter 2). Then, we will describe the details about why the commonly used distribution performs poorly on the composed concurrent containers and why our work can solve this problem (Chapter 3). After that, we will present the implementation of the hierarchical system view (Chapter 4) and the multimap and multiset containers (Chapter 5). Next, we demonstrate the performance improvement by conducting the experiments on the composed multimap containers who use the hierarchical system view for their distribution (Chapter 6). Finally, we conclude and discuss future work (Chapter 7).

2. PREVIOUS AND RELATED WORK

A considerable amount of research has been done on parallel programming frameworks. Some of them, such as Intel TBB [11], only target shared-memory systems. As we mentioned before, the structure of the shared-memory system is relatively simple and small compared to the distributed-memory system. Thus, those frameworks have limited distribution strategies for the nested parallelism according to the hierarchical structure of the shared-memory system. Other parallel frameworks may have some techniques that support the nested parallelism for distributed-memory system, but some of them only support 2-level nested parallelism, and other parallel frameworks require users to manually handle the distribution based on the system hierarchy.

2.1 Nested Parallelism In Other Libraries

X10 [12] is a programming language that designed for the distributed-memory system. The core concepts of X10 for the data distribution and storage are Activity, Place and Partitioned-global. An Activity is a lightweight thread for the task execution. A Place is a collection of Activities on the same processor or node. Partitioned-global represents a global address space for the elements to be accessed by both local and remote Activities. It supports hierarchical parallelism by applying a second level place-to-physical-node mapping, but users need to manually take care of the distribution at each level [13]. Our hierarchical system view uses the location information that is abstracted by the STAPL run-time system to automatically manage the distribution for multiple hierarchy levels.

Another productive parallel programming language, Chapel [14], provides similar concepts to STAPL; a Locale is a computational unit in the parallel architecture

that has uniform access to the memory, such as a node in the cluster architecture; a Domain is an index set used to map elements to the Locales. It shows a global view of distribution to users and makes the nested parallelism much easier by defining the sub-domains at each level [15]. But it has the same problem that the partition from a domain to sub-domains is not based on the system hierarchy; it leaves the burden on users to map the sub-domains to the appropriate Locales for the systems with different hierarchical structures.

Unified Parallel C (UPC) [16] is another programming language with Partitioned Global Address Space (PGAS) like X10. It only supports 2-level nested parallelism for task distribution. However, the data distribution is an important factor that affects the performance of the parallel applications. A poor distribution will generate unbalanced tasks which may not be simply handled by task distribution. For example, if there are two tasks; one task has extremely heavy workload and the other has little workload. The task distribution cannot balance the workload if the tasks are not dividable.

Legion [17] is a parallel programming framework that supports dynamic mapping according to the memory hierarchy in a HPC machine. Its Logical Region is a set of first-class values that may be dynamically allocated and stored in data structures [17]. Logical Region can be divided into sub-regions as needed. While multi-level memory architecture and nested parallelism are detected, the regions will be initially placed in the smallest memory where they fit. They represent the memory hierarchy as a stack with the increasing order of memory bandwidth from the local processor. This linear structure, sometimes, may not correctly reflect the real memory access time from remote locations, and thus, causes unimportant regions to consume the precious fast memory.

Kokkos [18], which is an extension of C++ library from Trilinos package [19],

supports nested parallelism by dividing processing elements into smaller groups at thread level. However, it only works on the shared memory system.

Unlike the related work mentioned above, our work supports both shared-memory system and distributed-memory systems. Its distribution is based on, but not limited to, the system hierarchy. Our work can group arbitrary consecutive levels together, or divide a level into multiple sub-levels according to the users' demand or the hardware properties. For example, the core level and the thread level can be grouped as one level if we only use one thread each core. Also, we can further divide the node level into sub-levels in which the cores in the nodes are on the same socket or not. This gives us the flexibility to choose the features we want for the distribution of the nested parallelism.

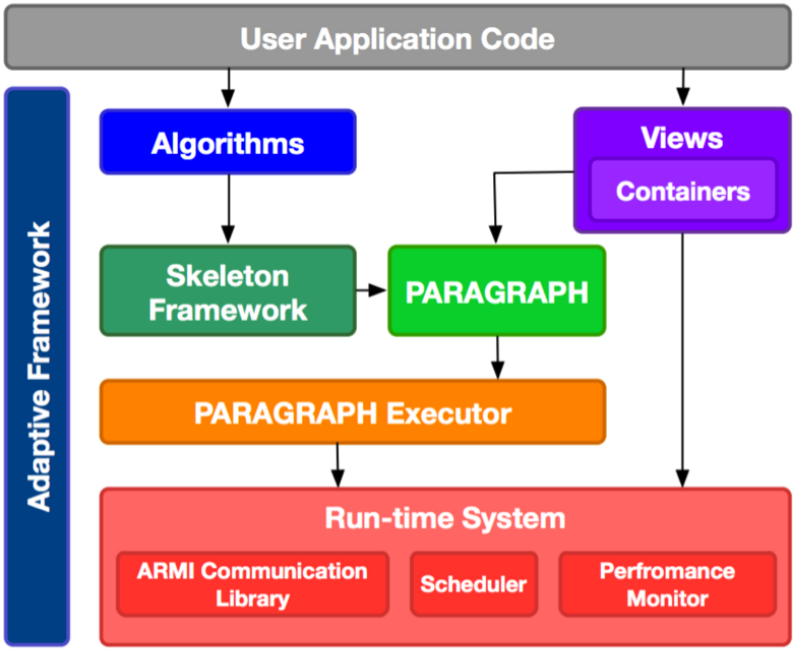

Figure 2.1: The STAPL framework

2.2 STAPL Overview

The Standard Template Adaptive Parallel Library (STAPL) is a parallel programming framework that provides building blocks for users to efficiently implement parallel applications [2]. Its main components include the parallel Containers and Algorithms, Views, Skeletons Framework, PARAGRAPH, and the STAPL Runtime System (Figure 2.1). A container is a concurrent, thread-safe data structure that distributes the elements to all locations. It has similar interfaces to its STL counterpart, but apply parallel methods on the data it stores. An algorithm is a parallel equivalent of the STL algorithm. It is expressed using algorithmic skeletons to represent its dependence patterns, and deal with the data through views [20]. A view is a lightweight abstract data type that references to a set of elements. It works like the iterator in the C++ Standard Library that provides a uniform collection of data access operations for the elements it represents. The skeletons framework [7, 8] simplifies the parallel patterns (e.g. map, reduce, and so on) a developer may use while writing the applications. The task dependencies are represented as PARAGRAPHs, and the abstraction of inter-processor communication is provided by the STAPL Runtime System [9, 10]. Other components that contain the mechanisms and strategies about data distribution and task scheduling are abstracted away from the users.

2.2.1 STAPL Runtime System

The STAPL Runtime System is an abstract layer that supports inter-process communications for parallel frameworks [10]. It uses hybrid MPI + OpenMP or threads model for shared and distributed memory system, and contains ARMI (Adaptive Remote Method Invocation (RMI)), a task scheduler and performance monitor. ARMI is a communication library that provides an abstraction of the inter-process com-

munication for the higher-level STAPL components [9]. It is a platform dependent component in that its running characteristics are dependent on the operating system and the computer architecture, but it has the ability to adjust its features automatically according to different platforms or conditions to improve the performance and resource usage. ARMI provides primitives for communication that are Remote Method Invocations (RMI). RMIs have the flexibility of passing data or calling methods between processors, and thus can be more easily adapted to the needs of the applications. The scheduler arranges the set of task graphs to determine which are to be executed first on each location [9]. The PARAGRAPH Executor is responsible for processing the tasks in a task graph that are ready for execution.

2.2.2 Nested Parallelism

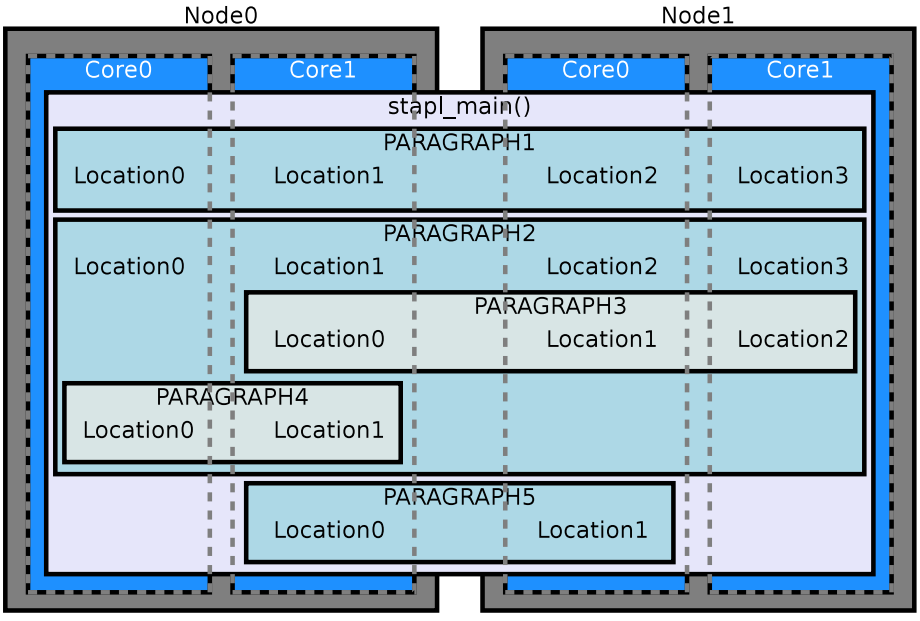

Figure 2.2: Execution model with nested parallel section [21]

The execution of the tasks in a parallel algorithm or operation is decided by PARAGRAPH, a distributed graph represents the task dependencies. Each PARAGRAPH is executed independently. A termination detection algorithm determines when a PARAGRAPH has finished and its results are available to other PARAGRAPHs. When a task itself in the PARAGRAPH is also a parallel algorithm or operation, it will create a new nested parallel section for its task execution and employ a nested termination detection [10]. The STAPL Runtime System supports the creation of parallel sections on arbitrary set of available locations. If a nested parallel section is needed (e.g. creating a composed concurrent container), it could be created using a subset of locations of its parent section. An example of nested parallel sections is shown in Figure 2.2.

2.2.3 STAPL Container Framework

To simplify the process of developing concurrent data structures, the Parallel Container Framework (PCF) provides a set of predefined concepts and a common methodology that can be used to construct a new parallel container through inheriting features from the appropriate concepts [6]. The main concepts defined in the Parallel Container Framework are the Global Identifier, Domain, Distribution, Partition, Partition Mapper and PCF Base Classes. The basic structure of the PCF and the interaction between these modules are shown in Figure 2.3.

The Global Identifier (GID) distinguishes the elements stored in a STAPL parallel container so that the elements can be distributed across the processors. A Domain is a collection of GIDs of the elements in the same container.

Data Distribution is the component that takes charge of the allocation of each element. An element will be stored on a location according to its GID. The Data Distribution uses a Partition to separate the domain into disjoint sub-domains, and

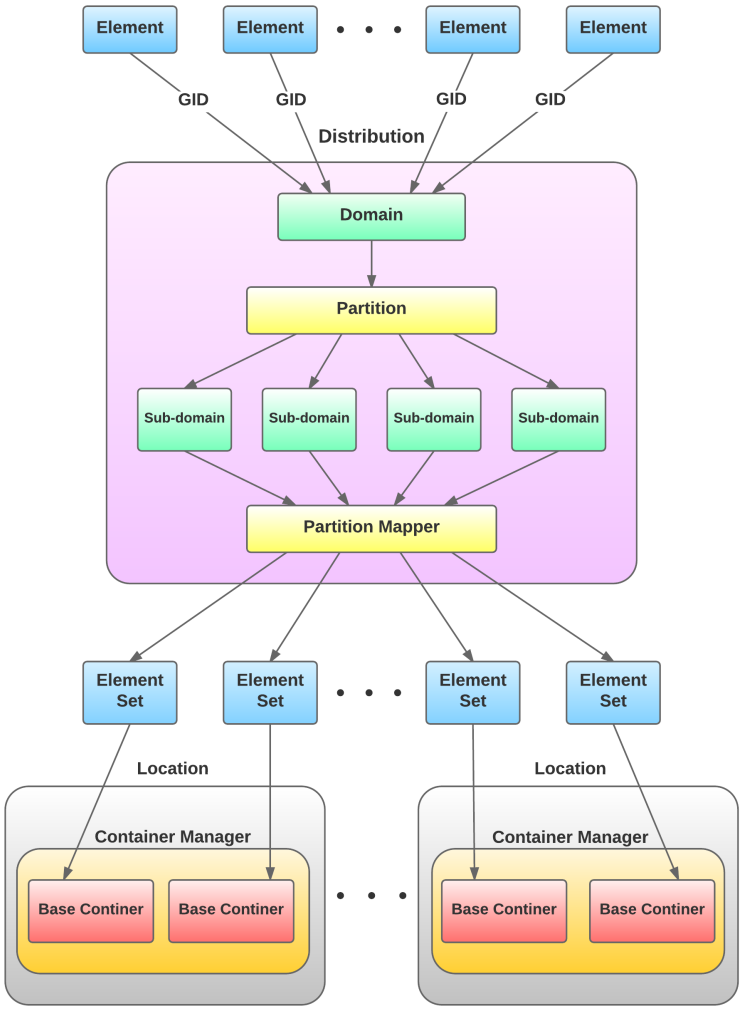

Figure 2.3: pContainer Framework Structure

then, employs a Partition Mapper to determine the location on which the elements associated with a sub-domain should be stored. Usually the Partition equally distributes the GIDs into the sub-domains to keep the balance of workload for each processor.

The PCF Base Classes are implemented as the building blocks for programmers to avoid the tedious and repetitive work for distribution and element access by building

new containers that derive from these base classes. They are generic classes that use template parameters to meet users' needs and provide the basic functionality of data distribution, element access, and container management.

2.2.4 View-based Distribution

The view-based distribution is a key concept in STAPL that gives a convenient way for users to define a customized distribution for their parallel containers. As we mentioned in section 2.2.3, users are required to provide a partition that divides elements to multiple subsets and a mapper that maps a subset to a corresponding location to make a customized distribution. Users can also specify this partition by defining a view.

$$View = \{Collection,\ Domain,\ Mapping\ functor\} \tag{2.1}$$

A view is defined by Equation 2.1. Collection is the underlying collection of elements; Domain is the set of elements that are referenced by the view; and the Mapping functor decides the mapping from View's domain to the Collection's domain.

$$V_{System} = \{Location\ container,\ Location_{domain},\ LID \rightarrow Location\} \tag{2.2}$$

$$V_{Partition} = \{V_{System},\ Partition_{domain},\ PID \rightarrow LID\} \tag{2.3}$$

$$V_{Elements} = \{V_{Partition},\ Elements_{domain},\ GID \rightarrow PID\} \tag{2.4}$$

The View-based distribution is a nested view defined by Equation 2.2, 2.3 and

2.4. The $V_{Elements}$ is a view decides how the elements' id (GIDs) are divided into multiple subsets (partitions). The $V_{Partition}$ is a view describes how a partition id (PID) is mapped to a location id (LID) in the V_{System}. The V_{System} is a system view indicates which locations should be used in the distribution. Thus, we use only one view to wrap all the information a container needs for its distribution.

Users do not need to create these views by themselves. STAPL provides predefined frameworks such as balance distribution, cyclic distribution, arbitrary distribution and so forth, for users to build a view-based distribution easily. Users can build a customized distribution by passing a domain of elements that will be distributed, a set of locations that will be used to store the elements, a functor that maps a GID to a partition ID, and a functor that maps a partition ID to a location ID to the view-based distribution framework which creates a view that represents the distribution they want for their parallel containers.

2.2.5 STAPL Redistribution

Many concurrent containers, such as vector, list and map, allow users to dynamically insert or remove elements. The balance of the original distribution will be broken if a considerable number of elements are inserted or removed. Another situation is that a concurrent container may be successively used in two algorithms that require different distribution strategies. Thus, STAPL provides a feature that allows a user to redo the distribution for an existing concurrent container as needed [21]. The user only needs to pass a view-based distribution to the redistribution function of a concurrent container, then the data stored in the container will be reallocated based on the new distribution.

3. NESTED DISTRIBUTION OF COMPOSED CONTAINER

In this chapter, we will discuss the difference between traditional distribution and our hierarchical distribution including their advantages and disadvantages using examples of 2D and 3D composed containers.

3.1 Balanced Global Distribution

The balanced distribution across the entire system is commonly used as the default distribution strategy in a concurrent container. It equally distributes the elements to all locations so that the elements can be processed by multiple processing units in parallel.

The default constructor of a composed container in STAPL uses balanced distribution for all the concurrent inner containers it has. Without information about the system hierarchy, it either restricts nested containers to a single location or distributes them across all locations. A concurrent container will create base containers on each location to hold its elements. If the elements stored in a base container are also concurrent containers, the elements of the inner concurrent containers will be distributed again. Thus, these base containers do not store the real data. They hold a place in the location to track their elements. Only the base containers created by the innermost concurrent containers own the data we want. However, we are continually creating base containers, and the data are distributed across the locations at each level during the process of constructing a composed container.

Comparing with a 1D container who has the same amount of elements as the 2D composed container distributed across the system at both levels, the number of elements stored on each location is unchanged, but the original locality of the elements in the composed container has been messed up as the elements are distributed across

the entire system.

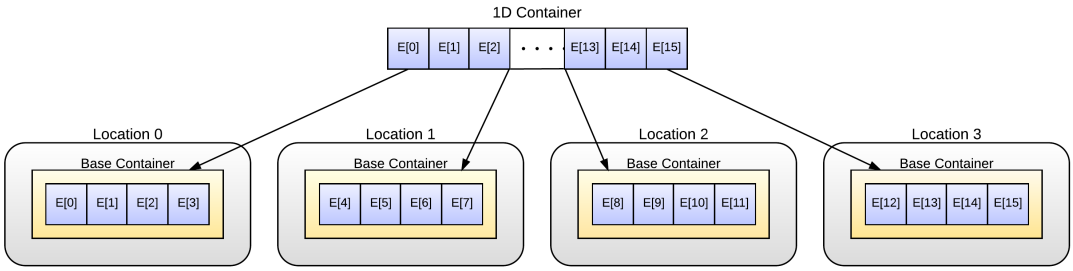

Figure 3.1: 1D Container Default Distribution

Figure 3.1 shows the allocation of 16 elements in a 1D concurrent array on 4 locations. The yellow rectangle around a set of elements represents the base container that holds the elements on each location. We can see that there is one base container created on each location, and each base container has 4 elements. When a parallel operation is applied to the 1D array, it will call the base containers on each location and let them to process its local data at the same time to improve the performance.

If we use a 4 by 4 2D concurrent array to hold the same amount of elements, the composed array will have 4 inner containers, which is a 1D concurrent array, and each inner container will have 4 elements. The allocation of elements in this 2D concurrent array is shown in the Figure 3.2. The 2D array first is distributed like a 1D array whose size is 4. It creates one base containers on each location to

hold a 1D array. Then, the elements in the 1D array will also be distributed to all the locations. The 1D concurrent array will create a base container on each location again and store its elements into them. A 1D concurrent array has 4 elements, so each base container only holds one element. There are 4 base containers created on each location by the 1D concurrent container. Thus, there are 4 base containers created by the 2D concurrent array and 16 base containers created by the inner 1D concurrent arrays on 4 locations. It requires 16 more base containers than the 1D container to hold the same amount of elements.

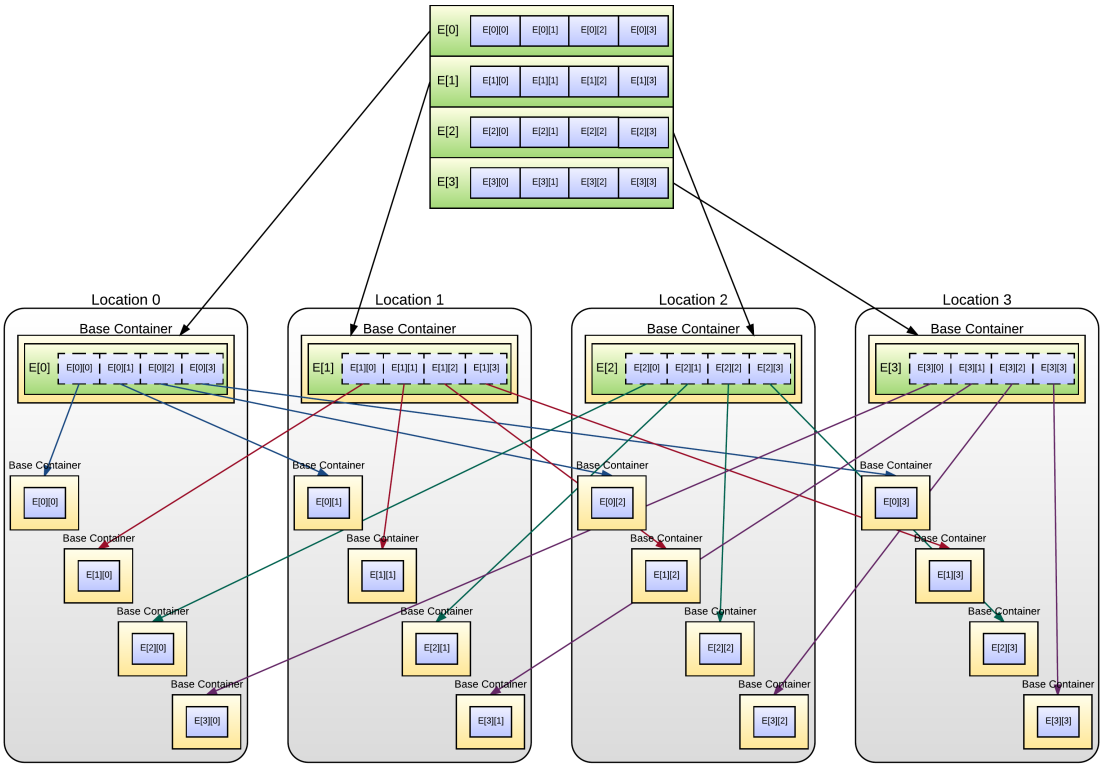

Figure 3.2: 2D Container Default Distribution

When we apply a 2-level nested parallel function on a 2D composed container, the outer container will inform all the base containers on each location to process the first level parallel function. Then, the second level parallel function will be applied to every inner container stored in the base containers. Each inner container is also required to communicate with its base containers on every location to process their elements. For example, if we want to modify the element E[0][3], we need to first communicate with the base container on location 0 to find its inner container E[0], and then, track down to its base container on location 3 to access the element E[0][3]. Therefore, the communication pattern at each level of an composed container with the global balanced distribution is a complete graph.

We can deduce the structure of a N-dimensional composed container from that of a 2D composed container. The disadvantage of the global balanced distribution in the composed concurrent container is that the elements stored on the same location do not have a good spatial locality because they are not from the same outer container. Whenever it has a nested level, the number of base containers created will increase and the tracing from the outermost container to a innermost element will require more communication between different locations. Especially when the number of locations is very large, the overhead of the communication and the construction of the base containers is significant when compared with the 1D concurrent container although they handle the same amount of elements.

3.2 Hierarchical Distribution

The hierarchical distribution is different from the global balanced distribution. It distributes the composed concurrent container based on the structure of the system hierarchy. The outer containers may be distributed at the node level and the inner containers could be distributed at the core level. Distributing a container at the node

level means that the elements of the container will be distributed to all the locations in the same node. But no matter which level we use, the principle of hierarchical distribution is to group the locations based on the structure of system hierarchy, and let a smaller set of locations to handle the distribution of its inner containers. A location is physically closer to the locations in the same set than the locations outside the set, and is likely to have lower communication overhead with it.

3.2.1 2D Hierarchical Distribution

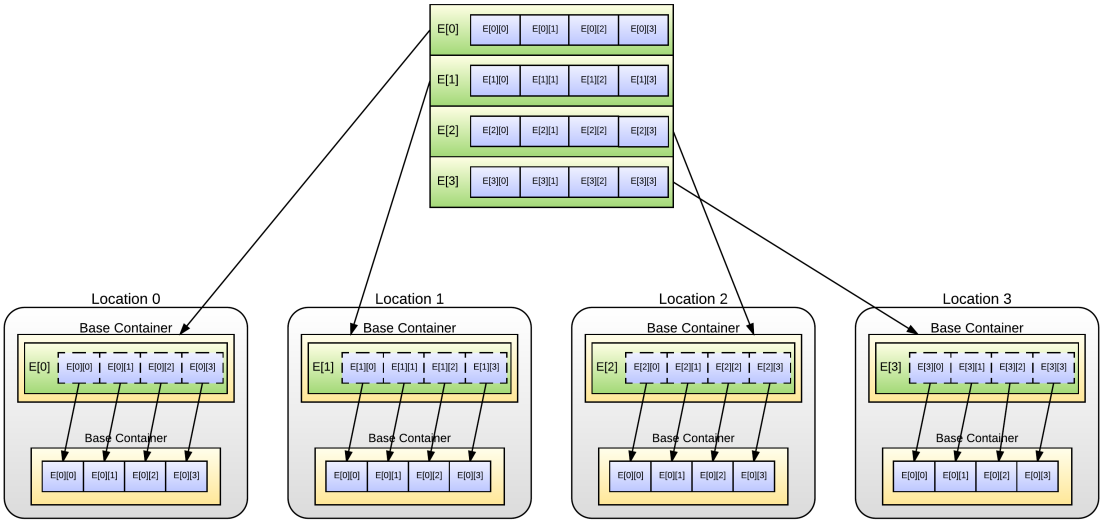

Figure 3.3: 2D Container Hierarchical Distribution

Distributing a container to a smaller location set will make each base container to hold more elements because the number of elements on each level is unchanged. And the locality of the elements will be improved because more consecutive elements

are put together in the same location. The number of base containers is decided by how many locations are used in the distribution at that level, thus the number of the base containers created by the hierarchical distribution is less than that created by the global balanced distribution. When a composed concurrent container is used in a nested parallel function, there will be less communication required to trace the innermost elements through the base containers. Moreover, the communication is limited in the same location set at each level. This reduces the cost that generated by the communication between remote locations (e.g. two locations on different nodes).

Figure 3.3 shows an example of the hierarchical distribution on the same 2D composed array as the one we showed in the Figure 3.2. The composed array has 4 inner 1D arrays, E[0] to E[3].

The distribution of the outer 2D array is same as the default distribution that creates 1 base container on each location to hold a 1D array. But the elements in the 1D arrays will be distributed to a smaller location set. For example, here we will limit the distribution of the 1D array in 1 location. So all the 4 elements in a 1D array will be distributed to the same base container on the local location.

We can see that there are 4 base containers created by the inner 1D arrays and totally 8 base containers created on 4 locations. It is much less than the number of base containers created by the composed array with the global balanced distribution. The benefits are the same if we use four location sets instead of four locations because more consecutive elements will be put on the same location, and the communication from a base container to its elements will be faster.

3.2.2 3D Hierarchical Distribution

The hierarchical distribution of a 3D composed container on a 2 by 2 hierarchical system is shown in Figure 3.4. The container has 4 elements on each level and totally

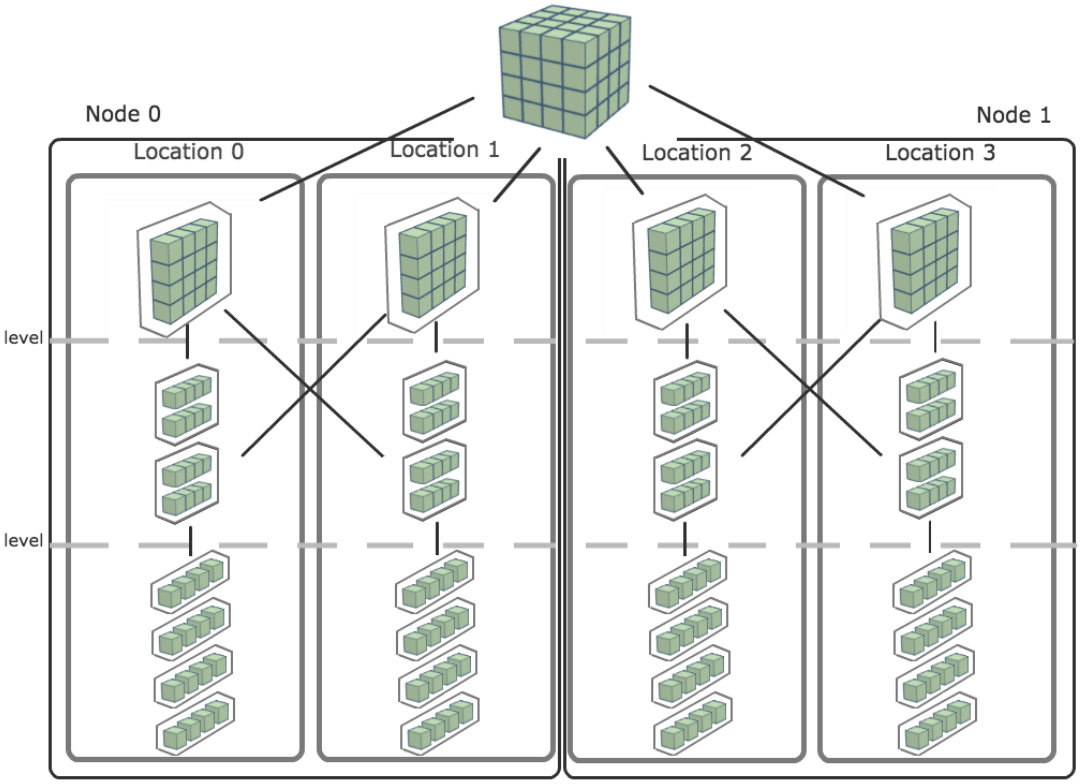

Figure 3.4: 3D Container Hierarchical Distribution

64 elements. It is distributed to a system with 2 nodes; each node has 2 locations. First, we distribute the 3D composed containers to all locations. Thus, each location receives one 2D container. Then the 2D inner containers on the Node 0 will be distributed to all the locations on the Node 0 (location 0 and 1). The inner containers on the Node 1 will be distributed to all the locations on the Node 1 (location 2 and 3). Each location will contain four 1D containers. The two 1D containers that are from the same 2D container will be grouped together in a base container. Finally, the 1D containers are distributed to the same location as the location its parent container stored, keeping them on a single location. Each location will store 16 elements. The 4 elements that are from the same 1D container will be put in one base container.

4. IMPLEMENTATION OF HIERARCHICAL SYSTEM VIEW

The Hierarchical System view is implemented using STAPL graph container and STAPL graph view [5] (Figure 4.1) because the hierarchy in a distributed-memory system can be represented as a tree structure. The STAPL graph container is a concurrent container that distributes its vertexes and edges to all locations and provides parallel methods to access and modify the vertexes and edges it stores. STAPL graph has several template properties that can be customized by users to adjust the behavior of the graph as needed. We choose the static undirected graph as the base class of our work. The static property means that the graph cannot be modified by dynamically adding or removing vertexes after its construction. Because the system hierarchy is decided at runtime, and usually the system structure will not be changed during the process of a parallel algorithm. We use the undirected property because the distribution of the nested parallelism always follows the hierarchical structure, whose order is from the top level down to the bottom level, so it is unnecessary to use directed edges. The main components of STAPL Hierarchical System View are Location Vertex, Hierarchy Container and Hierarchical System View.

The Location Vertex is the vertex property used in the undirected Hierarchy Container to represent a set of locations at that level. It contains the information of its level, child vertexes, descendant vertexes and the locations it represents. The child vertexes of a vertex v are the connected vertexes of v at the next level. The descendant vertexes are all the vertexes below v in the graph.

The Hierarchy Container is a data structure that represents the whole system hierarchy. It inherits the basic functionalities from the STAPL static undirected graph and constructs a tree graph using the location vertexes. It offers the methods

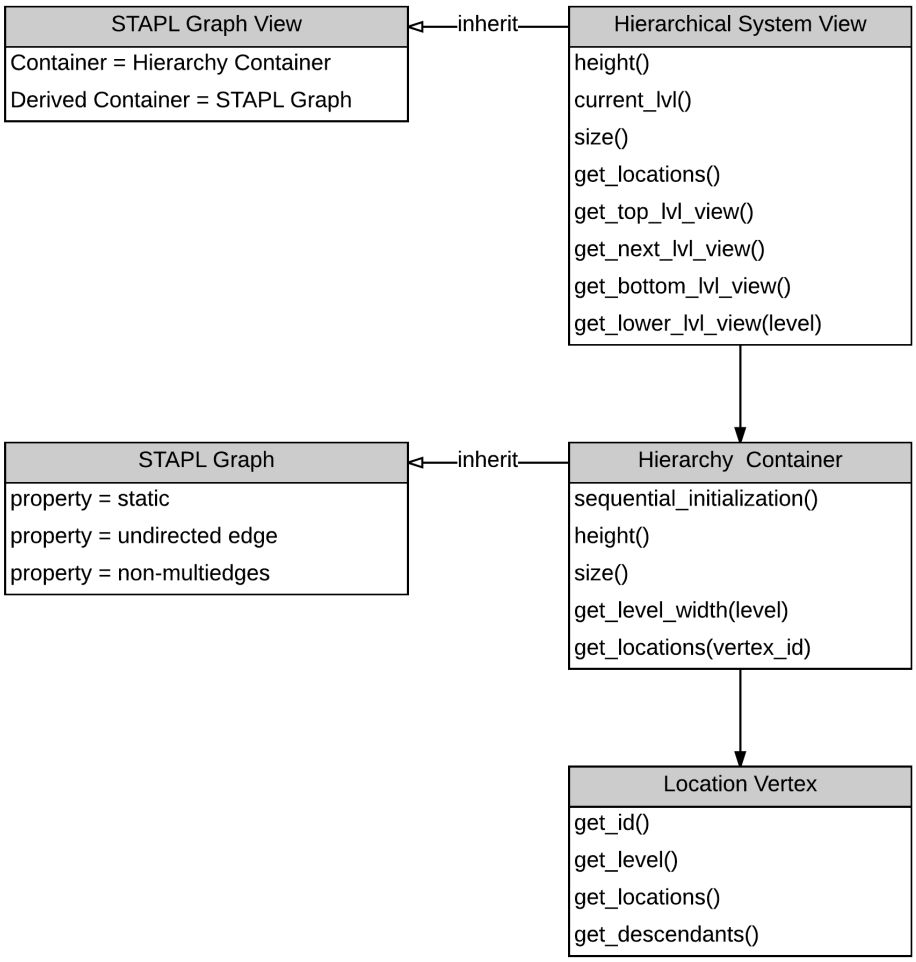

Figure 4.1: Hierarchical System View Structure

to find a certain vertex and get its information by invoking the corresponding methods in the location vertex. A vector of integers will be passed to the constructor of Hierarchy Container to indicate the height of the tree and the width of a vertex at each level. This vector is usually specified by a user at the runtime using

the $STAPL\_HIERARCHICAL\_PROC$ environment variable and returned by the *get_hierarchy_widths*() function. For example, if the vector is {2,4}, each vertex at the first level has two children and each vertex at the second level has four children. This will generate a 3 level tree graph showed in Figure 4.2. The top level will always be a single vertex that represents all the locations in the system. One thing that deserves our attention is that any level whose vertex width equals to one will generate exactly the same view as that of its parent level. We can reuse the view of the previous level to achieve the same distribution.

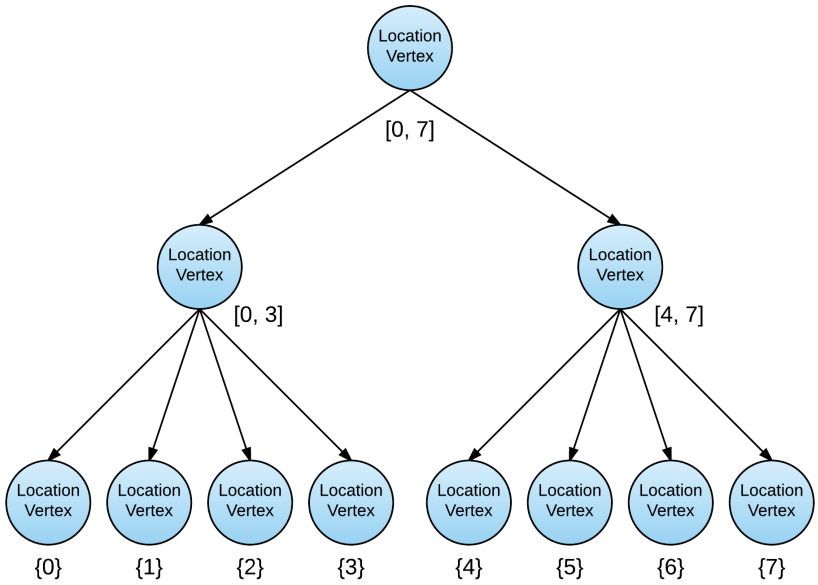

Figure 4.2: Hierarchical System Graph

4.1 Hierarchical System View

The Hierarchical System View is a graph view of the Hierarchy Container. It is a lightweight representation of a set of Location Vertexes in the Hierarchy Container. A user could access a Location Vertex and its information by calling the methods of the Hierarchical System View. It also provides the methods to get a sub-view for the top level, the next level, the lower level and the bottom level (Figure 4.3). The top level sub-view is a view of the hierarchical graph root, which is a single vertex that represents all the locations in the system hierarchy. The bottom level sub-view is a view with references to the vertexes at the lowest level that represents only one location for each vertex. The next level sub-view is the view of the vertexes at the next level. The lower level sub-view allows users to specify a lower level that the sub-view should be created. It can be used when the height of system hierarchy is larger than the dimension of the composed container, and let the users pick a suitable level in the system hierarchy for their distribution. The only thing required is that the specified level must be lower than current level because a vertex at a higher level represents a larger domain of locations, it will violate the rule that the location domain of a nested parallel section should be a subset of its parent location domain. Also, when the height of system hierarchy is less than the dimension of the composed containers, users can reuse the hierarchical system view of an appropriate level multiple times to give the best distribution for all the inner containers.

4.1.1 Sequential and Parallel Initialization

When a Hierarchy Container is constructed, its vertexes are empty. We will do the vertex initialization when the Hierarchical System View of that container is created. The reason behind is that we provide two ways to initialize the vertexes in the Hierarchical Container: the first one is to use the sequential initialization

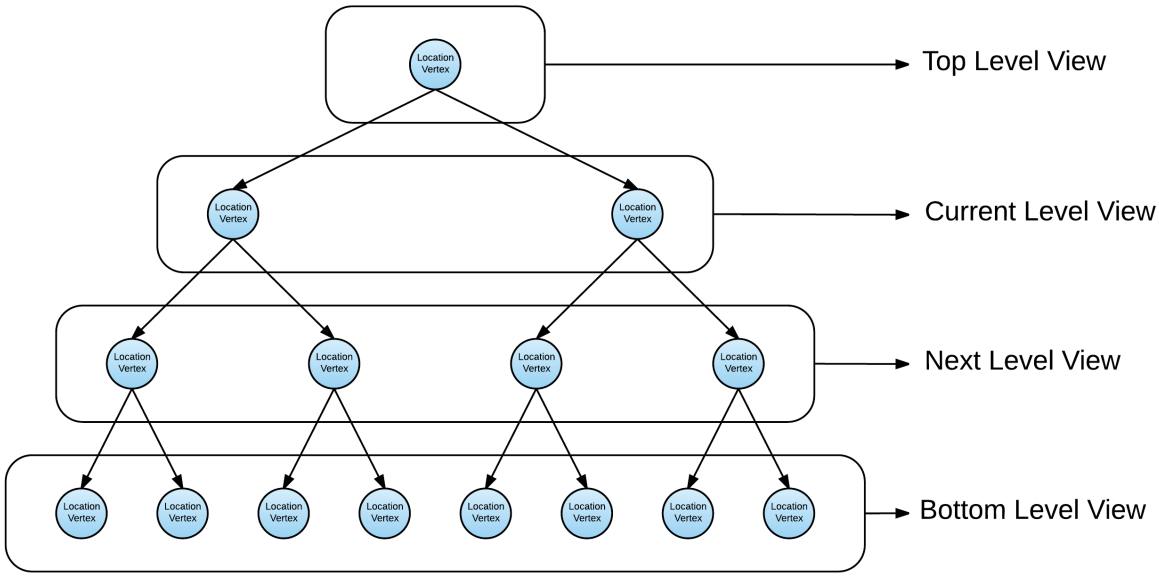

Figure 4.3: Hierarchical System View

function that computes and assigns the location domain for each vertex from the top level to the bottom level when the constructor of the Hierarchical System View is called; the other way is to call a generation function that initializes each vertex of a Hierarchy Container using all processing units in parallel, and then returns the Hierarchical System View.

If the Hierarchical System View is just created by its constructor, the sequential initialization function in the Hierarchical Container will be called to set the vertex properties. Thus, we recommend creating the Hierarchical System View using its generation function if the system hierarchy is very large because the sequential vertex initialization is not efficient although it may be only called once for each run of the

application.

4.1.2 Building Composed Distribution by Hierarchical System View

There are three steps to build a composed container by the hierarchical system view. First, a user needs to make a generator to specify the distribution for each element that is a concurrent container. The distribution can be easily made by passing a suitable level of hierarchical system view to a predefined view-based distribution framework provided by STAPL. Then, the user can build a composed distribution using the generator, and use the composed distribution to create a composed container. The hierarchical system view can also be used to redistribute an existing concurrent container. We only need to make a view-based distribution or a composed distribution by the hierarchical system view, and pass it to the redistribution function of a concurrent container. An example can be found in the following pseudo-code.

The pseudo-code in Figure 4.4 uses hierarchical system view to create composed containers. The outer container is balanced distributed to the top level of the hierarchical system view, which contains all the locations. The second level containers are cyclic distributed to the location sets at the second level of the hierarchical system view. All remaining levels except the last are arbitrary distributed to the location sets at that same level. The last level elements are blocked distributed to a single location represented by the vertexes at the bottom level of the hierarchical system view.

```
// generate distribution for each element who is a concurrent container.
struct generator
{
  hierarchical_system_view  hie_sys_view;

  distribution operator()(element) {
    switch(element.level)
      case 0:
        return balance_distribution(element.domain, hie_sys_view.top_level());
      case 1:
        return cyclic_distribution(element.domain, hie_sys_view.next_level());
       ...
      case i:
        return arbitrary(element.domain, hie_sys_view.lower_level(i),
                         GID_to_PID functor, PID_to_LID functor);
       ...
      case last_level :
        return block_distribution(element.domain, hie_sys_view.bottom_level());
  }
};

// make hierarchical system view by a vector of widths for each level
hierarchical_system_view = make_hierarchical_system_view(vector<int> level_widths);

// make composed distribution
composed_distribution comp_dist(generator(hierarchical_system_view));

// make composed container
outer_container<inner_container> composed_container(comp_dist);

// redistribute an existing container
existing_container.redistribution(comp_dist);
```

Figure 4.4: Use cases for Hierarchical System View in the distribution of composed containers

5. IMPLEMENTATION OF ASSOCIATIVE MULTI-KEY CONTAINERS

The reason for choosing multimap and multiset containers as the test object of our hierarchical system view is because the associative containers are often used to construct a composed container. Although multi-dimensional array is another most commonly used composed container, we already have implemented multiarray and matrix in STAPL to handle the distribution of the composed concurrent array, but the multimap and multiset have not yet been implemented in STAPL.

The STAPL multimap and multiset are designed to be the parallel version of the STL multimap and multiset, therefore, they will provide the same user-level facilities as their STL equivalents. Superficially, STAPL multimap and multiset are similar to their STL equivalents, but hide the detail of the concurrency and distribution management in the implementation of the STAPL containers. The components of the STAPL Container Framework that are specialized for multimap and multiset are the base container, the base container traits, distribution, container manager, the container class and the container traits.

The Traits are the template classes that can be passed to their corresponding components to specialize the properties according to users needs. It abstracts the types of primary components for the developer to access them rapidly and conveniently, while hiding the other private or subordinate component types.

The container classes of STAPL multimap and multiset have the interfaces that are similar to their STL counterparts; most member functions in STL multimap and multiset are provided in STAPL equivalents with the same function names, return types and parameters. Furthermore, the STAPL containers also provide extra functions for users to manipulate the data, such as applying asynchronous methods to a

given element. In the definition of these member functions, a key will be transformed to a unique GID to support the data distribution, because the key of the element in a multimap or a multiset is not necessarily exclusive. Thus, when an element with a key is passed into a function, the multiplicity of the key in this container will be calculated first, and then the multiplicity will be paired with the key to compose an instance of the multiKey structure as the GID of this element. The multiKey contains the type cast operator to allow implicit conversion from multiKey type to Key type. This allows the use of the container to be unaware of the multiplicity. Unordered multimap and unordered multiset also use the same multiKey concept as their GID type. In the ordered associative containers, the GID is sorted in the domain, so we also design a templated multiComp class as the comparator of the multiKey to keep GIDs in order.

The distribution classes of STAPL multimap and multiset are derived from the base distribution and associative distribution classes. The former is the common base class that provides essential functionality for all distributions of STAPL parallel containers. The latter is a subclass of the former that provides functionality exclusively for the distribution of STAPL associative containers. The distribution of STAPL multimap and multiset uses the balanced partition as the default partition if the user does not specify it. The balanced partition evenly divides the domain into subdomains that contain contiguous ranges of GIDs. It guarantees that the difference between the sizes of any two subdomains is at most one. When an element is inserted into a STAPL multimap or a STAPL multiset, its GID will be registered into the vector directory instance of the container. The vector directory is a class built for dynamic-sized containers to manage the distributed GIDs. It can determine the location on which a GID is located and allow user to invoke the methods on the location of GIDs without using external entities to know exact locality information.

The container manager of the multimap and multiset containers store the elements of the container mapped to a location by the partition and mapping. After mapping the subdomains to the locations, the elements with the GIDs in one subdomain will be sent to a base container, which is contained in the corresponding location that the subdomain should belong to. A Container Manager is the component that stores base containers and provides the access of the base containers on a location. The associative container manager is a base class implemented for associative container in the STAPL Parallel Container Framework. Deriving from the base class, the container manager classes of STAPL multimap and multiset inherit the basic functionality of supervising the base containers. It knows which base containers the elements reside locally. The container manager provides functions that can invoke base container functions on an element without specifying in which base container the element is located.

The base container, also called bContainer, is a subunit that holds a part of the elements to share the workload of STAPL container. The base containers of current STAPL containers are implemented based on their STL counterparts, whose properties are suitable for the requirement of STAPL containers. For example, the base containers of STAPL multimap consist of STL multimap and an integer class member, CID, which is the identifier of this base container. However, other libraries containers or a customized container can also be used to build the base container according to programmer's need. Except directly invoking the function of its STL counterpart, the base containers of STAPL multimap and multiset provide additional functions to support the interaction of PCF components, such as retrieving the identifier of a base container or returning a domain of all the GIDs in this base container. Moreover, to make the GIDs fit in with the features of STAPL multimap and multiset, a unique function is added to decrease the multiplicity of all the elements with

a certain key by one each time an element with that key is erased. It is unique for multiple associative containers, whose GID is a key-multiplicity pair.

To apply a method of STAPL multimap or multiset on a given element, the element's key will be first translated to the GID type, multiKey, and then invoke the related function of the container distribution. If the GID exists in the domain, the distribution will find which location the element is located and invoke the function of the location's container manager. The container manager will find in which base container the element is contained. Finally, the function of base container is called to access the element.

6. PERFORMANCE EVALUATION

In this chapter we present the methodology used in our tests and the performance evaluation of our work. We will first address the experimental environment, including the specification of the hardware and the software. Next, we will present the scalability tests for the STAPL multimap and multiset containers. Then, we will make the performance evaluation for the 1D, 2D and 3D containers that are using our hierarchical distribution, and comparing their results with those who are using the global balanced distribution. Note that for the remainder of this chapter,the global balanced distribution will be referred to as the default distribution.

6.1 Experimental Environment

Our experiments are conducted on Cray XE6m (rain) supercomputer. It is a distributed-memory system using 2-dimensional torus architecture. It has totally 576 cores contained in 24 nodes that are connected by Cray Gemini high-speed interconnect. The details of the hardware configurations can be found in Table 6.1.

There are 24 nodes in Cray XE6m; 12 of them have 2 AMD 16-core processors and the others have one 16-core processor with accelerators. The hierarchical structure of one node is shown in Figure 6.1. This is a node that contains two 16-core processors with 32 GB DDR3 Memory; one processor has two dies and each die contains 8 cores with a 6 MB shared L3 cache.

The software we used in our experiments is shown in Table 6.2. We use g++ 4.9.2 as the compiler of STAPL. And we install the boost 1.56 on rain because STAPL depends on the functionalities provided by several Boost libraries.

There is run-to-run variability in the execution time when we run the same program with a fixed number of cores. To minimize the impact of that variability, we

Table 6.1: Cray XE6m hardware specifications.

Board count	6
Nodes per board	4
Node count	24
Cores per node	32 on 12 nodes 16 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

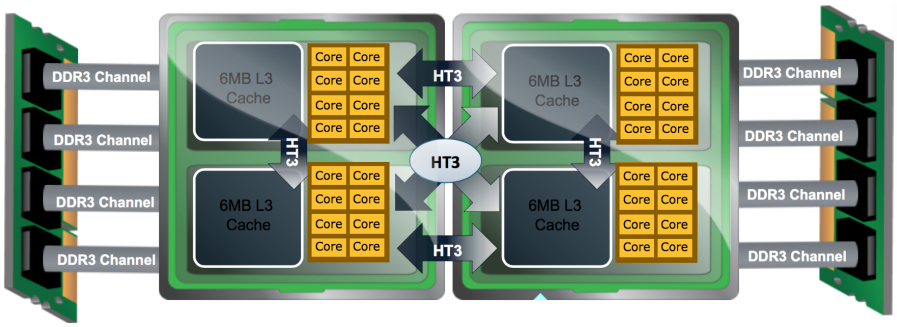

Figure 6.1: Hierarchical Structure of a Node in Cray XE6m

collect the results of the experiments with the same configuration 32 times, and then report the mean and 95% confidence interval using a normal distribution.

Table 6.2: Cray XE6m software specifications.

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

6.2 Scaling test of Associative Multi-Key Containers

Scalability is a characteristic that indicates the ability of an application to efficiently deal with the data using increasing numbers of parallel processing units (processes, cores, threads etc.). We did both strong and weak scaling tests on three basic functionalities of the STAPL multimap and multiset containers: insert, find, and erase. They are the most frequently used functions of these two containers and greatly affect the speed of data processing in a parallel program.

6.2.1 Strong Scalability Evaluation on STAPL Multimap and Multiset

Strong scaling refers to the test of the application running time using an increasing number of cores and a fixed input size. In strong scaling tests, an application is considered to scale linearly if the ratio of the performance improvement is equal to the ratio of the increased number of cores (Equation 6.1). The scale of running the application with k cores, S_k, is defined as the ratio between t_1 and t_k, where t_1 is the amount of time to complete a work unit with 1 core and t_k is the amount of time to finish the same unit of work with k cores.

$$S_k = \frac{t_1}{t_k} \tag{6.1}$$

Figure 6.2 shows a good scalability of all three functions, insert, find and delete, for STAPL multimap and multiset containers. We can see when the number of cores

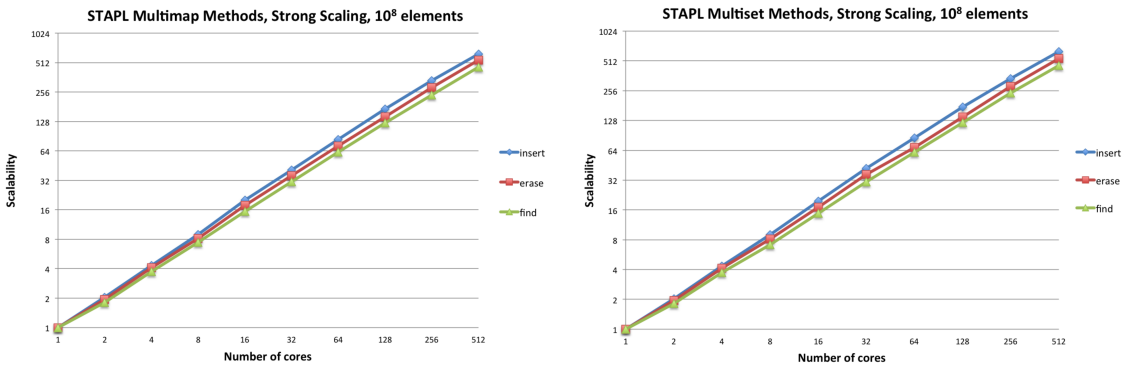

Figure 6.2: Strong Scaling Test for Multimap and Multiset using 100,000,000 elements

is doubled, we cut more than half of the execution time. This is because the operation time for the insert, find and delete in multimap and multiset is $O\log(n)$. It means the average operation time will increase as the number of elements increases. When we distribute the elements to multiple locations, we are not only cut off the workload on one location, we also reduce the average operation time since each base container now only holds a part of the elements. That is why our scalability is a little bit higher than the linear scalability.

6.2.2 Weak Scalability Evaluation

Weak scaling is another basic evaluation method that measures the parallel performance of a given application. In this test, the number of elements stored on each location is constant no matter how many cores we use. Weak scaling differs from the strong scaling, which focuses on the measurements of CPU limitation (CPU-bound), by emphasizing the presentation of the memory effect (memory-bound) on the parallel application. If the amount of time to complete a work unit with 1 processing element is t_1, and the amount of time to complete k of the same work units with k

cores is t_k, the weak scaling efficiency is defined as:

$$S_k = \frac{t_1}{t_k} \tag{6.2}$$

Because t_k is greater than t_1, in order to show the percent increase in execution time, we report the normalized execution time, T_k, which is defined as:

$$T_k = \frac{t_k}{t_1} \times 100\% \tag{6.3}$$

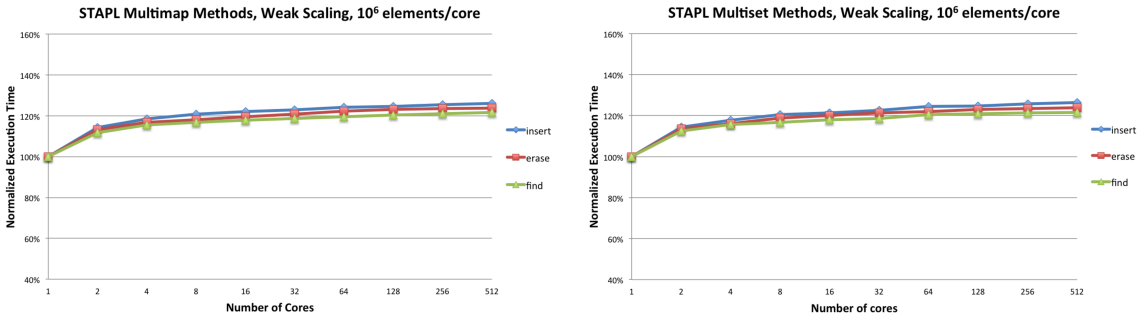

Figure 6.3: Weak Scaling Test for Multimap and Multiset using 1,000,000 elements on each core

Because the workload of each processor will not change, in theory, the execution time should be fixed as well. Thus, when we double the number of processors, the smaller variation of execution time, the better efficiency we have. From the graphical view, an ideal scaling should be a horizontal line.

Figure 6.3 shows the scalability of multimap and multiset in the weak scaling test. We can see that the scales of the execution time on each case are steady; the connecting line is approximately horizontal, especially when the number of running

cores is larger than 2. The normalized execution time on 2 cores is higher than 1 due to the communication overhead. Although there is more than 10% increase from 1 core to 2 cores, we can see that the variation of the execution time on the remaining core counts is very small.

6.3 Evaluation for Composed Containers Using Hierarchical System View

To evaluate the performance of applying our hierarchical system view on the composed containers, We make the distribution at two hierarchy levels in our experiments, the NUMA node level and the core level. We choose these two levels is because the communication speed between these two levels is significantly different. As we mentioned before, the inter-NUMA node communication speed will be much slower than that of the intra-NUMA node. Thus, we can clearly show how the hierarchical distribution improves the performance of the composed containers by reducing the expensive inter-NUMA node communications in the parallel operations. Although the hierarchical system view can represent a system hierarchy with more than 3 levels, a two-level hierarchy is enough for us to demonstrate our work because the 2D and 3D composed containers are the most frequently used composed containers.

In our experiments, The hierarchical structure will be represented as $N \times M$, where N is the number of NUMA nodes and M is the number of cores used per node; the product of $N \times M$ is the total number of cores used in this experiment. Basically, the number of cores per node will not be very large since each NUMA node only has 8 cores.

The way of the hierarchical distribution for the 2D composed containers is same as what we demonstrated in the example of section 3, Figure 3.3. We first distribute the elements in the composed multimap to all locations using the top level of our

hierarchical system view. Then we limited the distribution of the elements in a 1D inner container to one core.

The hierarchical distribution of the 3D composed multimap can also refer the example in section 3, Figure 3.4. The elements of the 3D containers are distributed to all the locations; the 2D inner containers are distributed within a node, and the 1D inner containers are allocated to a single core.

Here, we only demonstrate the evaluation on the composed multimap containers whose inner containers are 1D or 2D concurrent array because multimap and multiset have almost the same properties and one can infer the behavior of the composed multiset from the results of the composed multimap. The size of the composed container on each level is roughly equal. If the total number of elements is S, then the 2D container size is $\sqrt{S} \times \sqrt{S}$ and the 3D container size will be $\sqrt[3]{S} \times \sqrt[3]{S} \times \sqrt[3]{S}$.

6.3.1 Evaluation for The Construction of The Composed Containers

In this experiment, we evaluated the construction time for the composed containers using default distribution and hierarchical distribution. The system hierarchical structures used in these tests are $N \times 4$. It means that if the total number of cores is 16, we will use 4 NUMA nodes with 4 cores each.

From the Figure 6.4, we can see that the construction time of 1D container decreases. It is because its base containers are initialized with fewer elements when we use more cores. But the construction time of 2D and 3D composed containers with the default distribution does not decrease because their construction process requires more communication to create the base containers for the inner concurrent containers; also creating the base container will take time and memory space. As we mentioned in Chapter 3, 2D and 3D composed containers will create more base containers than the 1D container even if they hold the same amount of elements.

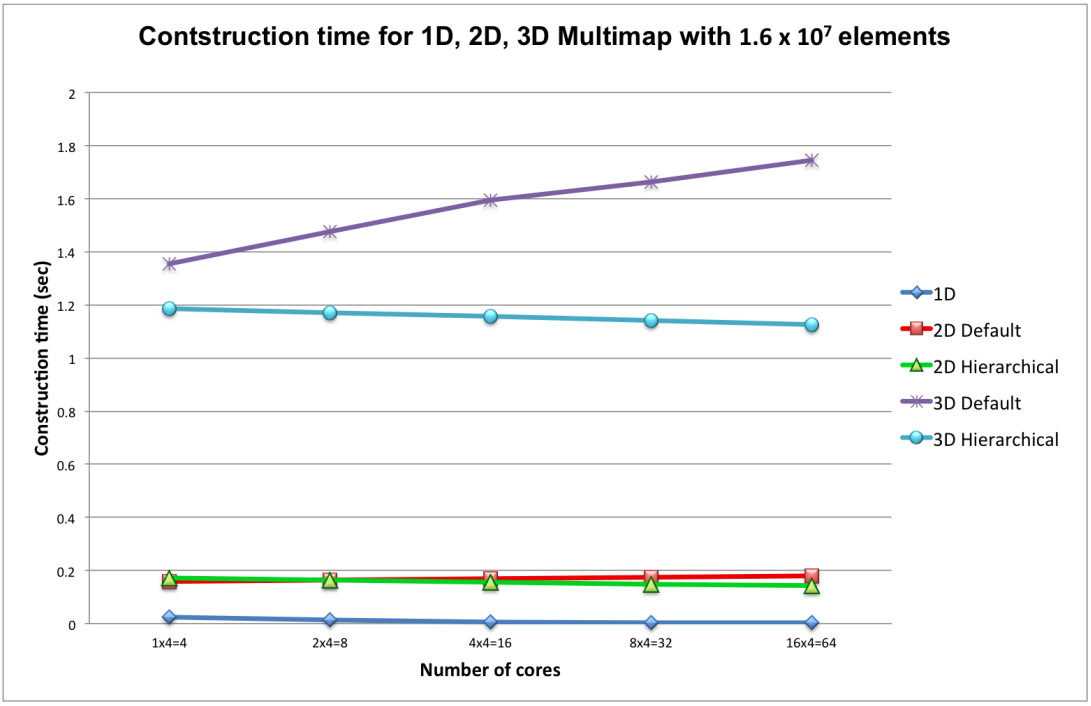

Figure 6.4: Construction time for 1D, 2D and 3D Multimap with 1.6×10^7 elements

The amount of communication and the created base containers will increase as the number of cores increases.

The results show that the composed multimaps using hierarchical system view for their distribution have a shorter construction time than that using default distribution because we reduce the number of base containers and communications especially those expensive inter-NUMA node communications. The improvement is not obvious when we use a small number of cores. The reason is that, first, our work does not save a lot on a small hierarchy because the amount of its remote communications is relatively small; also, there is an overhead of querying the location domain from hierarchical system view for each concurrent inner container. But when the system hierarchy is large, we save about 10% construction time in average for 2D composed multimaps, and for 3D composed multimaps, we cut more than 30% construction

time.

6.3.2 Evaluation for Parallel Functions on The Composed Containers

A composed container may be built only once in a parallel algorithm, but it may be used in the nested parallel functions multiple times. Thus, the performance of applying a nested parallel function on the composed container is of greater importance.

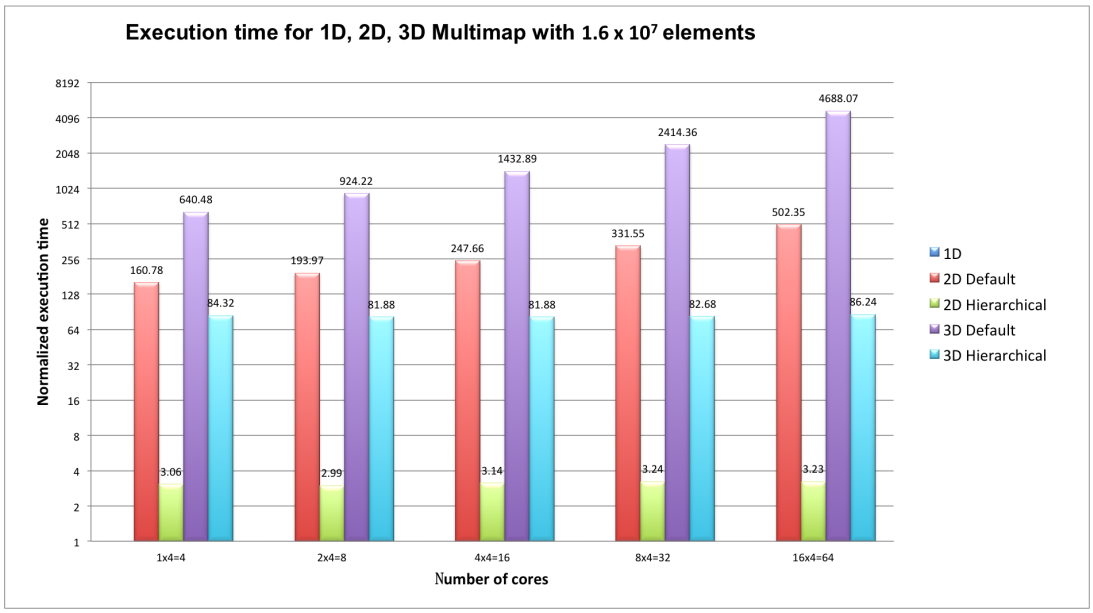

Figure 6.5: Execution time for 1D, 2D and 3D Multimap with 1.6×10^7 elements

Figure 6.5 shows the results of our experiments on the systems whose hierarchical structure are $2^x \times 4$. Here we evaluate the performance of the composed multimaps with 1.6×10^7 elements by applying a nested parallel function that modifies the value of all the elements in the container. We also normalized the execution time based on that of the 1D container and we use the logarithmic scale because the execution time of the 2D and 3D composed container with the default distribution is much longer

than that of the 1D container. Moreover, it grows dramatically when the number of cores increases. However, the performance of the composed multimaps that uses the hierarchical system view for their distribution is much closer to the execution time of the 1D multimaps comparing with the containers with the default distribution. The execution time of a composed multimap with the default distribution increases from 160 times to 502 times of the 1D execution time when the number of cores increases from 4 to 64. The execution time of our 2D composed multimaps with hierarchical distribution is only triple as long as the 1D execution time. The execution time scale of the 3D composed multimaps with default distribution increases from 640 to 4688, while the hierarchically distributed container remains around 80.

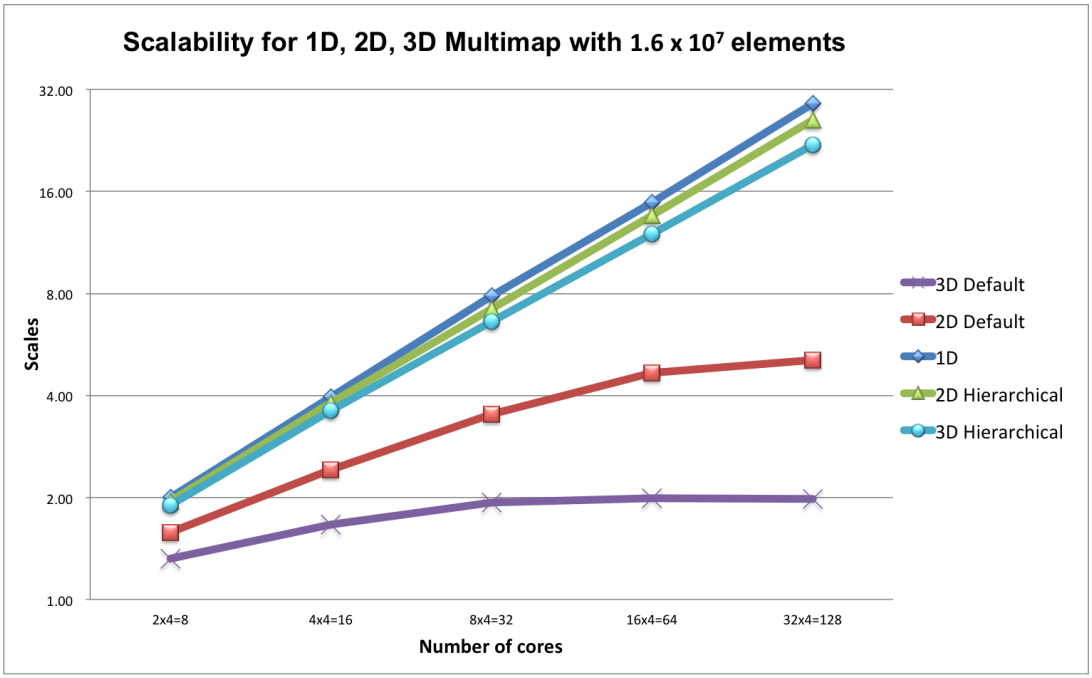

Figure 6.6: Scalability for 1D, 2D and 3D Multimap with 1.6×10^7 elements

The most important thing is that it has a good scalability. It means the larger

problem and system we deal with, the more time we could save using the hierarchical distribution. The scalability of the 1D, 2D and 3D composed multimaps are shown in Figure 6.6. The scales are calculated by the Equation 6.3. S_k is the scale value and t_k is the execution time using k cores. t_4 is the base of the scale that is the execution time of the experiments using one NUMA node with 4 cores.

$$S_k = \frac{t_4}{t_k} \tag{6.4}$$

We can find that the 1D multimap and our hierarchical distributed multimaps maintain a good scalability. The results are displayed as a straight line; it cuts nearly half of the execution time when the number of cores is doubled. However, the scalability of the multimap with the default distribution is very poor due to its high communication overhead. The performance improvement is really small when the number of cores is large.

We also evaluate the performance of our work on a fixed core count but different hierarchical structures. From Figure 6.7, we can find that the execution time on different hierarchical structures is very close because the communications are limited in the NUMA nodes and their inter-process communications are very fast. The execution time with larger node counts and smaller cores per node counts is a little bit faster because distributing the inner containers to a smaller location set makes the better data locality in the middle level. Also the competition of the shared cache is low because we only use partial cores of a node. There are many other factors that affect the performance, such as the property of the system hardware, the hierarchy levels that the user is using for the distribution, and whether the number of elements fits the size of the memory at that level. But these effects are less impressive than the performance gap between the composed containers with the hierarchical distribution

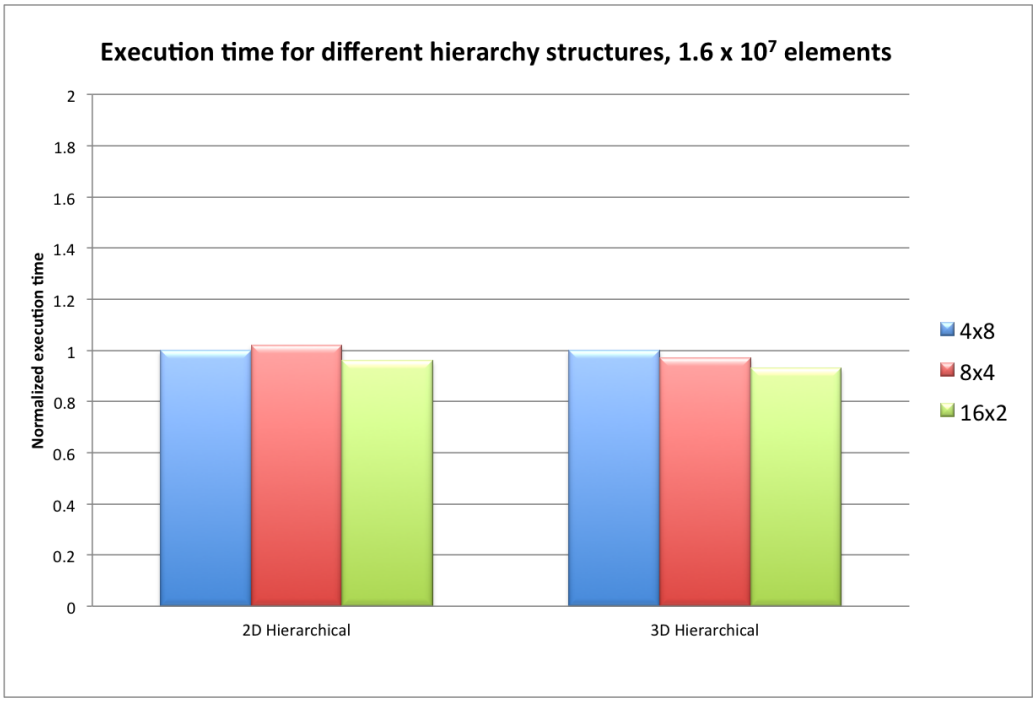

Figure 6.7: Execution time for different hierarchical structure with fixed core counts

and the containers with the default distribution.

6.3.3 Evaluation on The Redistributed Composed Containers

The hierarchical system view can work perfectly with the redistribution mechanism of the STAPL concurrent containers to flexibly adjust the distribution of an existent composed container at any time. It allows the users to maximize the performance of their parallel applications when the requirements of the distribution for the same container are changed in different algorithms. An even better use case may be implemented in the future is to automatically detect the available processing units in the system and redistribute the containers based on the view of current system hierarchy.

In the experiments of the container redistribution, we use composed 2D arrays to test the performance of our work. First, we construct a 2D array using the default

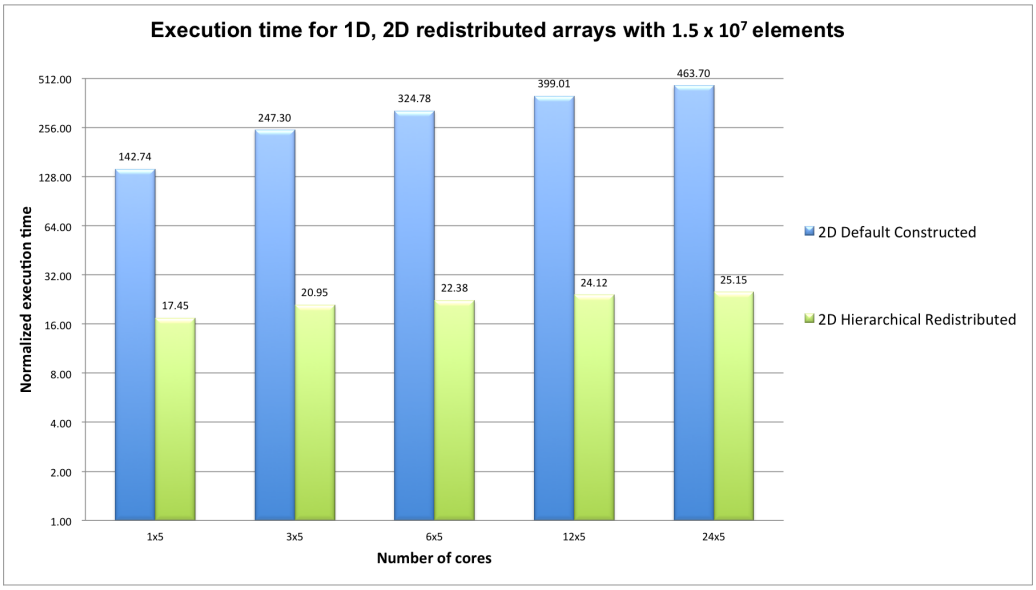

Figure 6.8: Execution time for 1D and 2D redistributed arrays with 1.5×10^7 elements

distribution and apply parallel operations on it to test its performance. Then we redistribute this 2D array using our hierarchical system view and record its execution time in the parallel operations again.

To show the flexibility of our work on the systems with different hierarchies, We run the redistribution experiments on the $N \times 5$ hierarchical systems for the 2D arrays with 1.5×10^7 elements. The experiment results shown in Figure 6.8 show that the hierarchical distribution still got great performance improvement in the container redistribution even on a hierarchical system with an odd node number per block.

7. CONCLUSION AND FUTURE WORK

In this thesis, we developed a hierarchical system view that represents the topology of the system hierarchy, and use it as a guide to the distribution of a composed container to replace the inefficient global balanced distribution mechanism. We discussed the principle of the global balanced distribution and our hierarchical distribution along with their advantages and disadvantages. We also described the implementation of the hierarchical system view and the implementation of concurrent multimap and multiset containers. At last, we conducted experiments for our hierarchical distribution on the composed multimap containers, and compared their results with the containers using default distribution to show the improvement of our work on both performance and scalability.

Future research on the hierarchical system view could focus on reducing the overhead of querying the location domain in which a container will be distributed. It is because our work requires the specification of the location domain for each element when we are constructing a composed container. The overhead will be obvious when the size of the composed container is large.

Another direction of the future work can be an automatic detection of the system hierarchy. Currently, the information of the system hierarchy used in our work is passed by the variable $STAPL\_PROC\_HIERARCHY$ at the runtime. It is necessary to figure out the system hierarchy automatically when there is no user specification. A feasible way is to test the communication speed between each pair of locations so that we can decide the system hierarchy based on that information.

Overall, the distribution of the composed container in a reasonable way is a key point to improve the performance of nested parallelism.

REFERENCES

[1] J.J. Dongarra, A.J. van der Steen. *High Performance Computing Systems: Status and Outlook*. Acta Numerica, pp. 01-16, 2012

[2] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Tkachyshyn, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger. *STAPL: Standard template adaptive parallel library*. In Haifa Experimental Systems Conference, Haifa, Israel, May 2010.

[3] John Shalf. *Cray XE6 Architecture*. NERSC XE6 User Training, pp. 19-21, Feb 7, 2011.

[4] Antal Buss, Timmie Smith, Gabriel Tanase, Nathan Thomas, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger. *Design for Interoperability in STAPL : pMatrices and Linear Algebra Algorithms*. In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Jul 2008.

[5] Harshvardhan, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger. *The STAPL Parallel Graph Library*. In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Tokyo, Japan, Sep 2012.

[6] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedhal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger. *The STAPL Parallel Container Framework*. In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), Feb 2011.

[7] Mani Zandifar, Nathan Thomas, Nancy M. Amato, Lawrence Rauchwerger. *The STAPL Skeleton Framework*. In Wkshp. on Lang. and Comp. for Par. Comp.

(LCPC), pp. 176-190, Hillsboro, OR, USA, Sep 2014.

[8] Mani Zandifar, Mustafa Abdujabbar, Alireza Majidi, David Keyes, Nancy M. Amato, Lawrence Rauchwerger. *Composing Algorithmic Skeletons to Express High-Performance Scientific Applications*. In Proc. ACM Int. Conf. Supercomputing (ICS), pp. 415-424, Newport Beach, CA, USA, Jun 2015.

[9] Nathan Thomas, Steven Saunders, Tim Smith, Gabriel Tanase, Lawrence Rauchwerger. *ARMI: A High Level Communication Library for STAPL*. Parallel Processing Letters, 16(2):261-280, Jun 2006.

[10] Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger. *STAPL-RTS: An Application Driven Runtime System*. In Proc. ACM Int. Conf. Supercomputing (ICS), pp. 425-434, Newport Beach, CA, USA, Jun 2015.

[11] J. Reinders. *Intel Thread Building Blocks: Outfitting C++ for Multicore Processor Parallelism*. OReilly, first edition, 2011.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. vonPraun, and V. Sarkar. *X10: an object-oriented approach to non-uniform cluster computing*. In Proc. of the 20th ACM SIGPLAN OOPSLA, New York, NY, 2005.

[13] K. Ebcioglu, V. Saraswat, and V. Sarkar. *X10: programming for hierarchical parallelism and non-uniform data access*. In International Workshop on Language Runtimes, OOPSLA, Vancouver, BC, 2004.

[14] B. Chamberlain, D. Callahan, and H. Zima. *Parallel Programmability and the Chapel Language*. Intel Journal of High Performance Computing Applications, vol. 21, no. 3, pp. 291-312, August 2007.

[15] Bradford L. Chamberlain Steven J. Deitz David Iten Sung-Eun Choi. *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*. In Proceedings of the 2nd USENIX conference on Hot topics in parallelism, Berkeley, CA, USA, 2010.

[16] UPC Consortium and others. *UPC language specifications v1.2*. Lawrence Berkeley National Laboratory, 2005. http://escholarship.org/uc/item/7qv6w1rk.

[17] S. Treichler, M. Bauer, A. Aiken. *Language Support for Dynamic, Hierarchical Data Partitioning*. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2013.

[18] H. Carter Edwards, Christian R. Trott, Daniel Sunderland. *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*. Journal of Parallel and Distributed Computing, Volume 74, Issue 12, December 2014.

[19] Christopher G. Baker and Michael A. Heroux. *Tpetra, and the use of generic programming in scientific computing*. Scientific Programming, 20(2):115-128, 2012.

[20] Antal Buss, Adam Fidel, Harshvardhan, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger. *The STAPL pView*. InWkshp. on Lang. and Comp. for Par. Comp. (LCPC), Oct 2010.

[21] Vincent Marsy, Nancy M. Amato, Lawrence Rauchwerger. *Dynamic Load balancing in A Geophysics Application Using STAPL*. Master's thesis, Department of Computer Science, Texas A&M University, August 2015.