DYNAMIC LOAD BALANCING IN A GEOPHYSICS APPLICATION USING STAPL

A Thesis

by

VINCENT SEBASTIEN MARSY

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 Member,	Richard L. Gibson Jr.
Head of Department,	Dilma Da Silva

August 2015

Major Subject: Computer Science

ABSTRACT

Seismic wavefront simulation is a common method to understand the composition of earth below the surface, especially for hydrocarbon exploration. One of these simulation methods is the wavefront construction algorithm. In this thesis, we reduced the load imbalance in a parallel implementation of the wavefront construction algorithm. We added a generic redistribution framework for data structures in the C++ parallel library STAPL. We present a redistribution algorithm for the parallel wavefront construction application which uses the recursive coordinate bisection method to find a near-optimal data distribution of the data. This algorithm leveraged the added redistribution features in STAPL to improve the running time of our application. We compared the run time of the application with and without redistribution on different geophysics models. We show that the proposed redistribution provides up to 9.45x speedup on a Cray XE6m cluster and 11.85x speedup on an IBM BlueGene/Q cluster.

DEDICATION

To my family

ACKNOWLEDGEMENTS

I want to thank my advisor, Dr. Nancy Amato, for her support during my research at the Parasol Lab since August 2012. Dr. Amato help me choose an interesting and challenging problem: Data redistribution in a parallel geophysics application. With Dr. Amato feedback and encouragements I was able to go through all the difficult steps.

I also want to thank Dr. Lawrence Rauchwerger, whose different courses and remarks helped me gain more knowledge about high performance computing and parallel computing in general. Dr. Richard L. Gibson Jr. gave me good suggestions on how to improve this thesis.

I also want to thank every member of the Parasol Lab, and especially members of the STAPL group. Without them, I could not have finished this work. Particularly I want to thank Adam Fidel, Shishir Sharma for sharing their knowledge on the parallel wavefront construction algorithm and the challenges of data redistribution. I also thank Dr. Timmie Smith for sharing is knowledge on STAPL.

Finally I want to thank my parents and friends for their support and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Parallel programming today cannot be overlooked when high performance is needed. From basic personal computers to supercomputers, modern architectures contain multiple cores. Using these processors in parallel to solve problems faster seems obvious. However, writing code to do so is considered harder and more error prone than standard sequential programming. Different programming paradigms are available to provide a way to program various hardware architectures in parallel, all of which provide a balance between ease of use, portability, and performance.

Because large-scale parallel programs work with multiple cores and distributed memory, many requirements are added compared to sequential programs. One of them is to specify the data layout of a data structure. Indeed, not all the elements of a data structure reside on one processor's memory. Typically a data structure, also known as a container, will have its elements distributed more or less evenly amongst all processing elements' memory. In the sequential world, a programmer is used to the spatial and temporal locality when accessing elements of a container. In parallel programs, this locality has an even bigger impact: one processor wants to access elements which reside on the processor's memory. Otherwise the processor needs to access that element remotely, which is costly. Of course some remote calls cannot be avoided, but a programmer will try to specify a data layout that minimizes the number of remote accesses.

Parallel algorithms performs better if the data is well distributed. For some problems, distributing the data is easy, while for other it can be an issue. An example of a problem where the distribution is challenging is a seismic ray tracing application which implements a parallel wavefront construction algorithm [15].

Seismic ray tracing algorithms are used in the field of geophysics to estimate the properties of geological formations. It is used to simulate seismic wave propagation.

In a seismic ray tracing algorithm, a hypothetical earth model is provided with different layers in the earth's crust. The algorithm simulates the propagation of seismic rays from a specific source. This simulation is compared to real world data, which was collected using seismometers. Figure 1.1 shows how the data is collected in the real world. Usually the source can be thumper trucks, which drop heavy weights on the ground to generate a seismic wave. The receivers are seismometers laid out around the source. The source can also be an air gun below a boat which generates a seismic wave. In that case the seismometers are attached behind the boat on a long cable. The hypothetical earth model is iteratively refined by the user until the ray tracing simulation matches the data that was collected using the methods explained above.

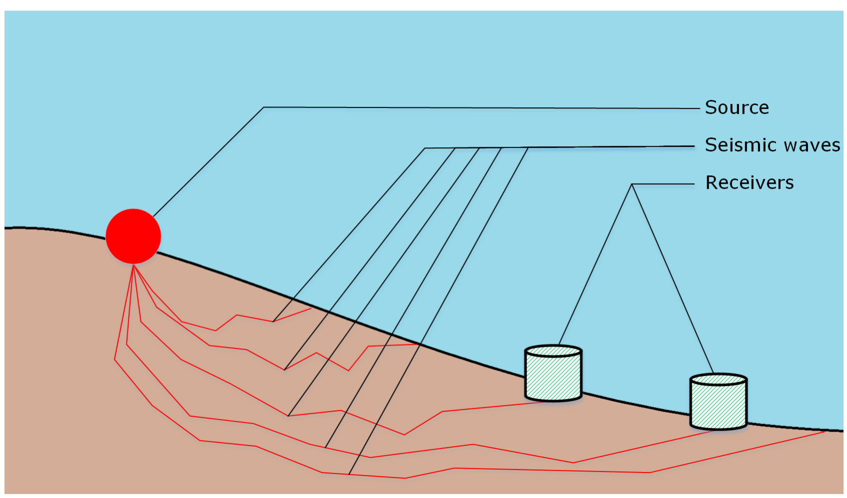


Figure 1.1: An example of a source emitting a seismic wave and the receivers gathering the seismic data.

This method, known as Forward Modeling, has a huge impact and is very important to the hydrocarbon exploration community. Methods other than ray tracing exist to perform seismic modeling, such as finite-difference method (FDM) which attempts to solve exact equations and apply finite-differences techniques to solve many partial differential equations. The advantages of this method is that they can compute the complete solution [31, 22] but the disadvantages are the difficulty to interpret the results, the slow computation time and the high memory requirements of the system [21]. The state of the art ray tracing method is the Wavefront Construction(WFC) method [29, 30, 20, 23, 11, 16, 32] as it provides many of the advantages of the ray tracing method, without its inconveniences [17]. However, a new drawback of the WFC method arises when parallelized [15]: The distribution of data becomes imbalanced during the algorithm execution.

1.1 Contributions

In this thesis, our objective is to improve the distribution of the data in a seismic ray tracing application which implements the parallel WFC algorithm using STAPL. In order to do this, we add a redistribution feature for dynamic container in STAPL. We improve the STAPL framework to offer to the user containers for which the data layout can be defined and updated easily. The infrastructure to perform data redistribution should be modular, extensible and customizable so that it can handle the change from any data layout to another data layout for data structures in STAPL. STAPL already has a framework to represent data structures which will be explained in Chapter 3. The redistribution functions are built on top of this framework in such a way that already existing applications do not suffer from performance degradation due to the new redistributions features. The objectives we cover in this thesis can be summarized in these main points:

1. Extend STAPL's redistribution framework to easily redistribute *any* kind of dynamic containers.
2. Update the Seismic Ray Tracing (`SRT`) application to enable the use of the redistribution framework.
3. Use that framework, in combination with some data balancing technique in `SRT` to obtain better performance and scalability for the parallel wavefront construction algorithm.

Dynamic containers distributor. We will provide a module called the `Distributor` for dynamic containers. This module is responsible for redistributing a `pContainer`, handling all the necessary steps required to finish the redistribution with a `pContainer` in a valid state.

Updating `SRT`. The first parallel implementation of `SRT` was done using an older version of STAPL which did not use the modern `pContainer`s and was not using the most efficient data structures for the work.

Use Recursive Coordinate Bisection (`RCB` [3]) to improve `SRT`'s performance. Once `SRT` was written using the newest STAPL features, it was possible to use the well-known load balancing algorithm `RCB` in coordination with the `Distributor` to automatically load-balance the data structures in `SRT`.

1.2 Outline of Thesis

In Chapter 2 we describe the basics of our seismic ray tracing application: the parallel wavefront construction algorithm it is based on, and STAPL, the C++ parallel library used to implement it. We also discuss the related work. In Chapter 3 we describe the framework for data redistribution in STAPL. We then explain in Chapter 4 the algorithm used to perform a redistribution of the data in the Seismic Ray Tracing application. We then present the results of this redistribution in Chapter 5. Finally, Chapter 6 concludes this thesis.

2. PRELIMINARIES AND RELATED WORK

In this chapter, we present the preliminaries and related work. First we present describe the STAPL parallel C++ library which was used for our implementation. Next, we give an overview of the parallel wavefront construction algorithm and the problem we encounter with it. Then, we describe the related work in the domain of data redistribution. Finally, we describe the different techniques for partitioning data. Specifically, we give details about the Recursive Coordinate Bisection method that we will use in our work.

2.1 STAPL Overview

The Standard Template Adaptive Parallel Library (STAPL) [6] is a parallel programming framework that extends C++ and STL with unified support for shared and distributed memory parallelism. STAPL provides distributed data structures (`pContainer`s) and parallel algorithms. It relies on a runtime system which provides the abstraction for communication between locations. Each location communicates through asynchronous messages called RMIs [25]. A location is an individual processing unit (a process or thread), with its own local memory address space. Accesses of data in a location are *local* if the data is stored in that location's address space, or *remote* otherwise. The number of locations is specified by the user, using a combination of MPI processes, and a number of STAPL threads per process.

Each element in a `pContainer` is identified via a unique identifier: the Global Unique ID (`GID`). This `GID` can be a scalar, but also support complex types such as a string, or user-defined types.

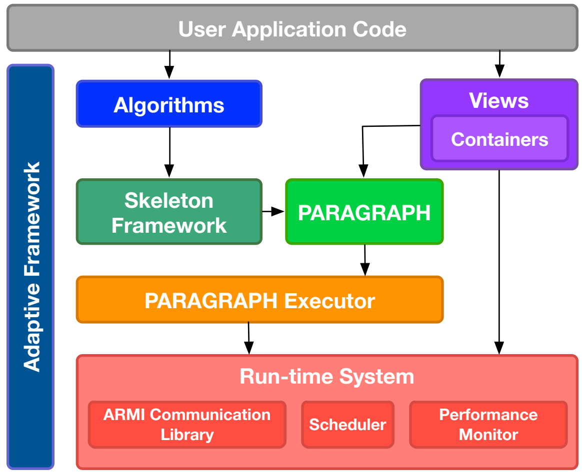


Figure 2.1: The STAPL framework.

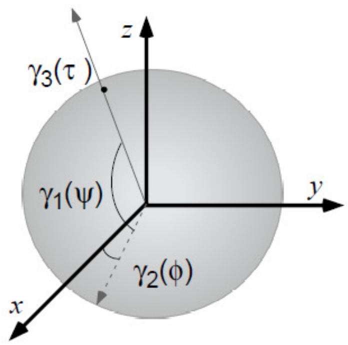
The major components of STAPL to store and retrieve data in `pContainer`s is the *directory*. This directory has two main components: the *registry* and the *manager*. These will be discussed in more detail in Section 3.1. The *directory* is responsible for everything that is required for inserting, reading, writing, or deleting any element of a `pContainer` based on its `GID`. View-based distributions are another first-class concept in the `pContainer` framework. For any `GID` they map it to its location with a two-step process: a partitioning phase, and a mapping phase. These two elements will be discussed in details in Section 3.2, but the key advantage is, from a user's perspective, a simple two-function specification. This makes the programmer's work easier and it also facilitates the change of distribution in the framework itself. Given any new partitioning and mapping functions, the `pContainer` can determine how to redistribute the data to go from the old distribution to the new one as we will see in section 3.3.

2.2 Parallel Wavefront Construction Method

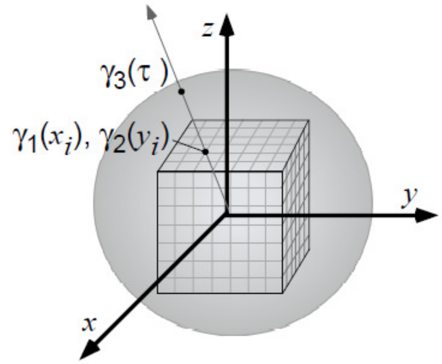
The parallel `SRT` application [15] was originally implemented in STAPL. It is an attempt to parallelize the wavefront construction algorithm [21] using STAPL. In [15], Jain presented a successfully implemented parallel wavefront construction algorithm. A high level description algorithm is presented in Algorithm 1.

An interesting characteristic of the algorithm that explain our choices described in Chapter 4 is the numerical properties of the wavefront. The wavefront represents a surface connecting the points of same travel time along each ray path [21]. The elementary geometric subdivisions of our mesh are quadrilateral shapes. The main idea of the algorithm is that every of these mesh cells respect some conditions in order to keep a minimum level of ray density across the wavefront.

To initialize the mesh of rays, two methods can be used: Take-off angle mesh or Cubed sphere mesh. Both are a set of initial rays exiting from the source. These two methods have advantages and disadvantages. For the take-off angle method we use two ray parameters, γ_1 and γ_2 and we can set the third parameter as the traveltime, $\gamma_3 = \tau$, or arclength, $\gamma_3 = s$ [28]. To generate the initial mesh we just need to connect the points sharing the same travel time along the ray paths ($\gamma_1 = \psi$, $\gamma_2 = \phi$, and $\gamma_3 = \tau_0$). ψ represents the declination, ϕ represents the azimuth, and τ represents the traveltime. The advantage of this method is that this representation is very natural and easy to visualize and implement. However, the density of rays is not well distributed on the sphere, we end up with a much higher density at the top and bottom of the sphere (where the declination angle $\psi = \pm 90°$) as pointed out in [21]. Figure 2.2 (a) shows a representation of the take-off angle mesh coordinates, where the center represents our source. On the other hand, the Cubed-sphere method, presented in [21], lets us use different values for γ_1 and γ_2. These values still identify rays with unique coordinates, which fills the requirement for our algorithm. In that second method, the parameters are generated using an imaginary cube (called focal cube) centered at the source point. To compute the values for γ_1 and γ_2, the rays are projected from the source for a unit traveltime, passing through discretized points on each face of the cube. Each face of the cube has $N * N$ points, which gives us $6 * N^2$ points over all the 6 faces of the cube. These $6 * N^2$ points are uniquely represented as a pair of (x_i, x_j) coordinates, where x_i is the x_1 component of a face, and x_j is the x_2 component of a face. These coordinates are used for γ_1 and γ_2. Specifically, $\gamma_1 = x_i$ and $\gamma_2 = x_j$. That method is less natural than the take-off angle method, but the rays are then evenly distributed over the faces of the focal cube. With the current implementation, as long as the original rays coordinates can be expressed as unique two dimensional points (γ_1, γ_2) the wavefront can be successfully propagated.



(a) Take-off angle mesh



(b) Cubed sphere mesh

Figure 2.2: Different mesh initialization geometries a) Take-off angle mesh coordinates, the ray parameters are defined as $\gamma_1 = \psi$(declination), $\gamma_2 = \phi$(azimuth), and $\gamma_3 = \tau$. b) Cube sphere mesh coordinates, the ray parameters are defined as $\gamma_1 = x_i$, $\gamma_2 = x_j$, and $\gamma_3 = \tau$ [15].

Algorithm 1 Parallel wavefront construction algorithm.

Require: Earth model, mesh description, number of source and their position

Initialize the rays and the ray tubes in parallel using the user specified geometry
while (**true**) **do**
  for $i = 0$ to $\frac{dt_{ray\ tube}}{dt_{ray}}$
    parallel for each ray $\in$ collection of rays
      Trace the rays by one time step
      If the ray segments intersect a surface in the earth model, then create new ray segments if needed
    end parallel for
  end for
  parallel for each ray tube $\in$ collection of ray tubes
    Step the ray tube by one time step
    Interpolating/coarsening the ray tube if necessary
  end parallel for
  if No ray tube remains
    break
  end if
end while

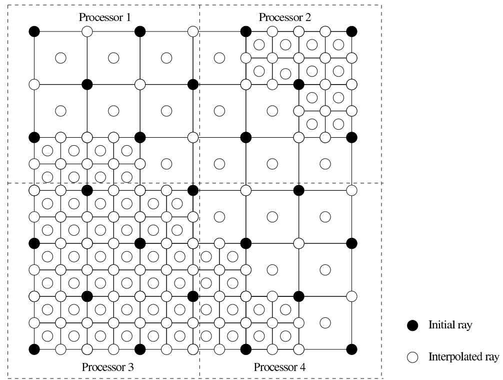


Figure 2.3: Possible load imbalance situation in the seismic ray tracing application [15].

The mesh cells are what we call raytubes. Each raytube represents a set of four or five rays; at each time step of the algorithm, a raytube is going to check if a threshold of error is crossed for the rays it is responsible for. This error can be computed using different physical properties of the rays. These criteria can be the distance between the adjacent rays, the area defined by the four corner rays of a raytube, or more complicated criteria such as the travel time perturbation using paraxial ray tracing [21]. If this threshold is crossed, a raytube *interpolates* to keep the rays density in an acceptable range. Interpolating will divide the raytube into children raytubes. This will increase the number of rays in the area of the wavefront where the raytube is. These new children raytubes will have their properties below the error threshold chosen by the geophysicist.

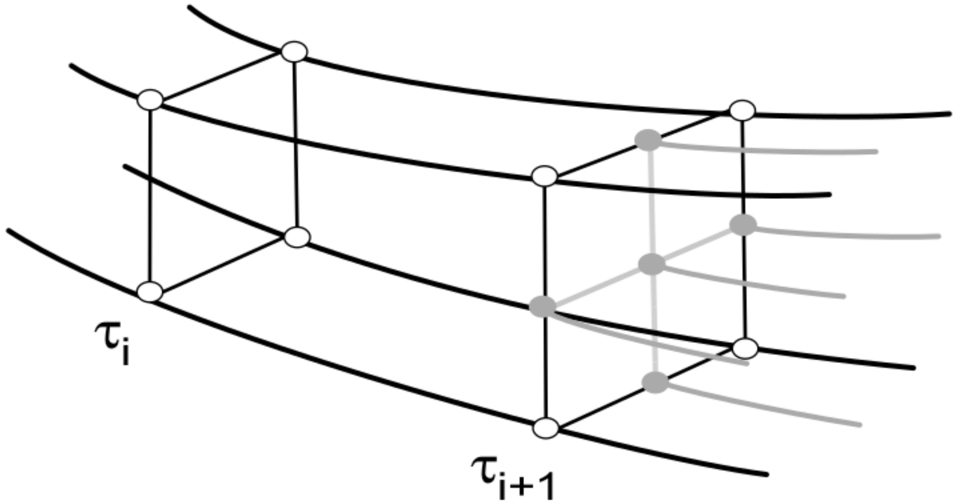
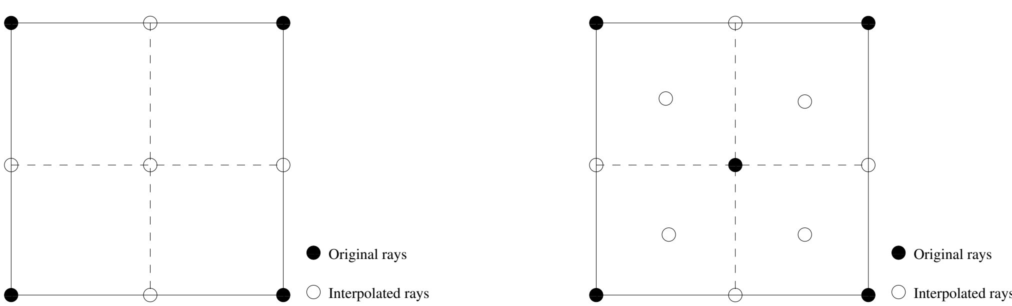


Figure 2.4: A raytube interpolating over time [21].

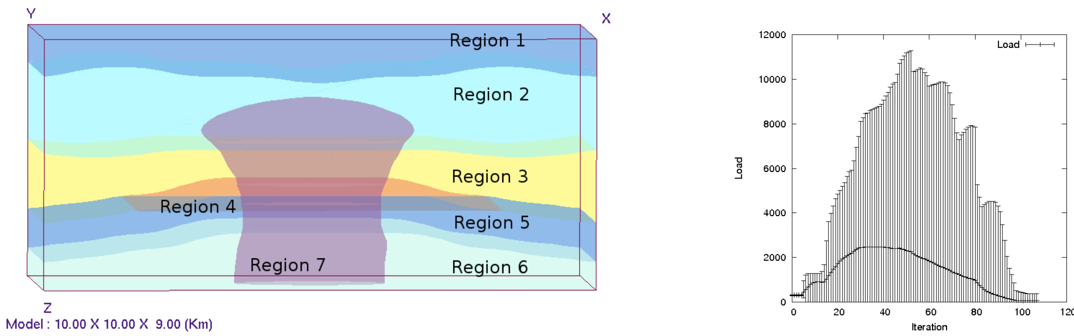
Figure 2.4 shows an example where a raytube interpolates over time, the location one which the original rays and the raytube were now have increased its number of rays and raytubes.



(a) with 4 rays (b) with 5 rays

Figure 2.5: Interpolation of a raytube [15].

In Algorithm 1 at line 11: the interpolation and coarsening of the raytubes is what makes the algorithm's results more interesting than a conventional ray tracing application. However, it is the main cause of poor scalability in the parallel implementation. A raytube interpolation is dependent on the input model and the characteristics of its rays. Some processors can have raytubes which decide to interpolate often, which creates an uneven distribution of the raytubes and rays. Figure 2.5 shows simply how the load imbalance can arise: At every interpolation new rays are going to be created. If we use 4 rays per raytube as shown in (a), then a raytube generates 5 additional rays. If we use 5 rays per raytube as shown in (b) then 8 additional rays are added. Coarsening is the reverse process: When rays of neighboring raytubes are too close to each other, these raytubes can merge back into their original parent raytube. This can also generate an uneven distribution of the rays and raytubes across processors. An example of the load imbalance can be seen in Figure 2.3. Figure 2.6 shows a particular model that exhibits this imbalance behavior. Figure 2.6(a) shows an example earth model for the simulation, and Figure 2.6(b) shows the disparities of load over time on 64 processors. The maximum load is represented by the bars, while the average load is represented by the line. We can see that the maximum load on a processor can be significantly higher than the average load across processors.



(a) (b)

Figure 2.6: (a) A complex salt dome model and (b) its load profile through time. (p=64) (as shown in [15]).

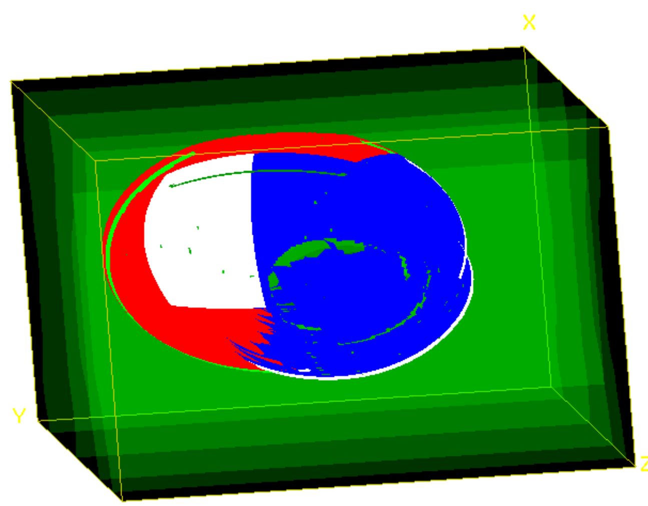


Figure 2.7: Visualization of the wavefront imbalance in the salt dome model (p=4). Each color represents one of the four different processors.

Figure 2.7 is a visualization of the wavefront on four processors.We show only four processors so that the we can easily distinguish between the four areas on the wavefront. Originally, the rays in the wavefront were distributed evenly between each processor. Over time, we notice that a majority of the rays of the wavefront are held by the processors represented in blue.

2.3 Data Redistribution

Several efforts to make parallel programming easier to users exist. Some, such as Cilk [12] and Intel TBB [14], only target shared memory architectures. In that context, the data is present only on one physical memory, and thus the concept of *remote* access is not as important; a poorly laid out data distribution will only show poor spatiotemporal locality, or could lead to false sharing. However, these issues are orders of magnitude less impacting than remote memory accesses, where the latency and bandwidth to access the data reduces performance. In the shared memoy context it is still important that each thread shares the work evenly, but insuring that each thread handles the same amount of data is not a significant challenge. Some frameworks such as Chapel [7], UPC [24], and X10 [10] which target distributed memory architectures encounter the problem of poorly located data and try to solve these issues by providing user-friendly ways to allow data redistribution. In [26], the authors discuss how Chapel users can enable re-allocation of data, but these re-allocations show suboptimal performances due to re-allocation communication performed without aggregation. Aggregating means that if some data needs to be migrated from one processor to another, only one message will carry the data, instead of one message **per datum**. Aggregation avoids a large amount of unnecessary latency. In [26] the authors show that for some specific key cases involving block and cyclic distributions (which are dense regular domains), the authors are able automatically aggregate communication. In [2], the X10 authors present some compiler optimizations to reduce communication overheads, but the data migration is also done without aggregation. The solution we will describe in Section 3.3 does perform an aggregation of the data. UPC presents how carefully crafted collective communication such as an all-to-all communication done with a complexity of $O(n * \log n)$ instead of $O(n^2)$ can improve performance.

However, none of these framework propose a generic algorithm for redistribution. The data migration task is left to the user, who can use the interface provided by these system. In UPC an *exchange* method can perform the collective exchange. In X10 the data migration is the responsibility of the user, with potential compiler optimizations reducing communication. In Chapel, if the users only use Block and Cyclic distributions and specifies them in the code, then the redistribution is hidden from the user. Otherwise the user has to take care of it.

Moreover, the unique key used to represent each element in these frameworks is limited to scalar key types. STAPL's redistribution framework is more generic by allowing any type of key but still aggregating communication for the redistribution.

In this thesis we focus on a centralized redistribution, where each processor shares the same global knowledge about the data imbalance. Another type of redistribution is decentralized redistribution. In that case, each processor does not share global knowledge, but instead evaluate the imbalance using load information from its neighbors processors. For instance, in the *Neighborhood load balancing* algorithm *NeighborLB* [18], each processors compare its load with its neighbors and decide to redistribute data with them. This type of redistribution does not require global communication which could improve the scalability of the algorithm, but could yield to less accurate redistributions.

2.4 Dynamic Data Structures Applications

Other parallel scientific applications show similar issues to SRT. For instance problems like adaptive mesh refinement [4] can exhibit the same behavior: Over time, the data will grow and shrink at different rates on each processor. Many techniques exist to find a better partitioning of the data, each with their advantages and disadvantages which needs to be considered on a per application basis.

Graph partitioning is a popular method for distributing load evenly. It requires the problem to be expressed in form a of a graph $G = (V, E)$ where V are vertices and E edges. These edges are links that connect vertices. These links represent certain properties about the connection between two vertices.

A graph partitioner then computes sub-domains with the same number of vertices and which also satisfies constraints on the edges. For instance, the constraint can be that the sum of the edges connecting each sub-domain is minimized. The disadvantages are that the problem needs to be expressed as a graph, and the computation of this graph partitioning can be costly [8].

When the problem can be expressed in a n-dimensional space, we can use a geometric partitioner instead. One advantage of such partitioner is that the problem does not need to be expressed as a graph. Objects to partition must be represented as a n-dimensional unique coordinate. The performance of geometric partitioner is better than the graph partitioner, since the constraints are simpler than some constraints on edges. We saw in section 2.2 how rays in the mesh in the parallel wavefront construction algorithm can be represented in a 2D space using (γ_1, γ_2) coordinates, which makes geometric partitioning a good candidate for SRT.

There exist a few frameworks which provide geometric partitioning, Zoltan [9] is one such framework. The interoperability of STAPL with different frameworks makes it easy to use one of `Zoltan`'s partitioning methods to find a fast and good partitioning of the rays.

Various geometric partitioning methods exits, a popular one is is Hilbert space-filling curve [13]. Hilbert ordering linearly orders and generates an evenly balanced partitioning. However, its main disadvantage is that each region does not share the same number of neighbor partition. With our raytubes, we want to minimize the communication between processors, so we decided to us another geometric partitioning technique: Recursive Coordinate Bisection(`RCB`). `RCB` guarantees us that each sub-region will have at most four neighbors sub-regions.

The Recursive Coordinate Bisection (`RCB`) algorithm was described by Berger and Bokhari [3]. The algorithm operates on data represented in a spatial domain. Each datum has an $x_1, x_2, ..., x_n$ unique representation in a n-dimension domain. During the first step of the `RCB` algorithm, the domain is cut into two regions. That cut is orthogonal to one of the coordinate axes of the original domain. The cut is done such that there is half of the elements on one side, and half of the elements on the other side. The next steps is repeating recursively the cutting on the two sub-regions computed at the previous step. The recursion stops when we reach the desired number of created sub-regions. These sub-regions are the sub-domains that we will use to partition our data.

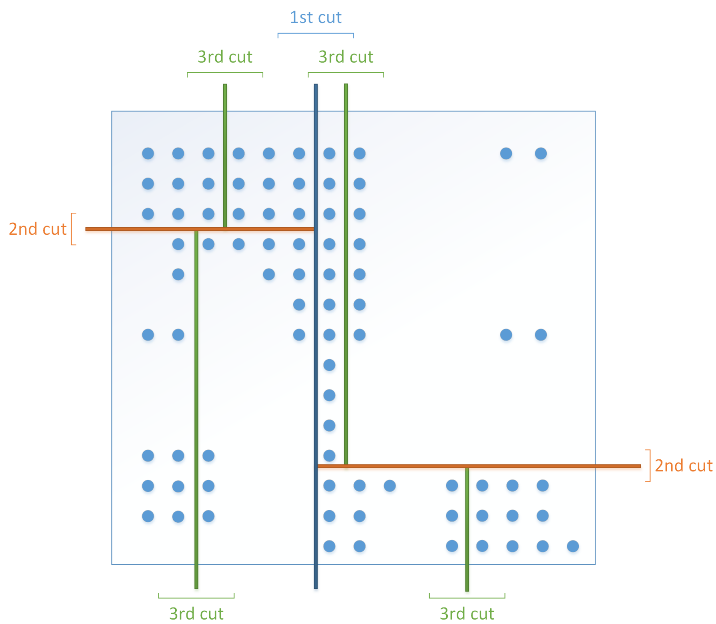


Figure 2.8: An example RCB decomposition in a 2D space. Each cut is done such that the number of elements on each side of the cut is the same.

Figure 2.8 shows a potential decomposition of points in a 2D space. In this example we would have three levels of cut, leading to eight different regions. These regions would contain the same number of points.

The original algorithm was created a number of sub-regions equal to a power of two, but the implementation we use in `Zoltan` [9] can handle any number of sub-regions. Indeed, nothing prevents us to cut a region in n sub-regions instead of only two regions.

When presented in 1987, it was advertised as a good static load-balancing algorithm, but it actually exhibits good properties for dynamic load balancing. Indeed, throughout the execution of the program, the `RCB` algorithm will produce incremental partitions, which minimizes the changes of each region. This will reduce the amount of communication needed during the redistribution phase. Each of the sub-region in a 2D space is made with cuts parallel to the main axis of our domain, these sub-regions are rectangles by definition, so they share at most four sides with other sub-regions.

SRT shows a particular problem that is not frequently seen in other scientific applications, which is a strong coupling of two different data structures: one for the rays and one for the raytubes. The flexibility of STAPL makes it easy for an application to make use of the same partitioning for these two different data structures. We will give more details about this strong coupling in chapter 4

3. DATA REDISTRIBUTION IN STAPL

In this chapter, we go into details on how to perform redistribution in STAPL. First, we describe in details the components of the `pContainer` that will need to be updated. Next we describe in details the interface to specify the partitioning in STAPL, using *View-based* distributions. Then we describe the algorithm for redistribution of a `pContainer`in stapl and discuss its correctness.

3.1 Parallel Container Overview

The parallel container framework was first described in [27]. A parallel container (`pContainer`) is an object oriented implementation of a data structure in STAPL's parallel environment. It is a distributed data structure that holds a finite collection of elements in a non-replicated fashion. The `pContainer` is defined by a global directory, as well as one *container manager* per location, which will handle the different local base containers. Each type of `pContainer` has a specific type of base containers. For instance a `stapl::unordered_map` will have as a base container the `std::unordered_map` so that the complexity to find/insert/erase local elements is $O(1)$. Figure 3.1 shows a breakdown of the `pContainer` framework.

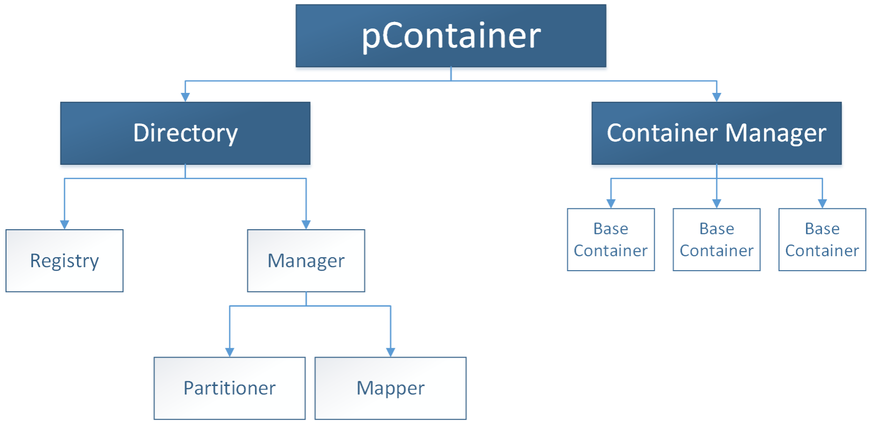


Figure 3.1: The `pContainer` Framework.

The directory has two important components: the manager and the registry. The manager is responsible for knowing which location is responsible of any given `GID`. To be able to answer this, it relies on the partition and the mapper that will be described in Section 3.2. The registry on the other hand is responsible of knowing if a `GID` is present or not in the `pContainer`. If an RMI reaches a location with the intention of updating some `GID`'s value, the registry will queue that request until the `GID` is indeed registered. This permits asynchronous use of the `pContainer`s, providing scalable performance. The registry is also the component responsible for the knowledge where each `GID`s are actually stored. Some containers can bypass the manager and register a key at a different location. This is true for elements that have been migrated using the per-element migration; in such cases, the manager would still be responsible for knowing the location of the registry that knows where the `GID` is.

The container manager is responsible for local base containers. For some distributions there will be only one base container per location. Other distributions, such as cyclic distributions, may place multiple base containers on a location. Each base container is uniquely identified by its Partition ID (`PID`).

3.2 View-based Distribution

As we can see in Figure 2.1, `pViews` [5] are one of the major component of STAPL. They are used to get a lightweight representation of containers. They are inspired from the C++ STL (The Standard Template Library) iterators. Like algorithms in the STL that only use iterators as input and output, `pViews` in STAPL are used in a similar way to provide a decoupling of containers and algorithms. Equation 3.1 shows what the necessary components of a `pView` are (as described in [5]).

$$View = \{Container, Domain, Set\ of\ operators, Mapping\ functor\} \tag{3.1}$$

But `pViews` can give an abstract representation of more than `pContainers`. In fact, the `pContainer`'s partition and mapper described in Section 3.1 are now decoupled from the `pContainer` through a view-based distribution.

$$V_{System} = \{Locations, \mid Locations \mid, \emptyset, ID\} \tag{3.2}$$

$$V_{Partition} = \{V_{System}, Partition_{Domain}, \emptyset, PID \rightarrow\ LID\ functor\} \tag{3.3}$$

$$V_{Elements} = \{V_{Partition}, \texttt{GID}s_{Domain}, \emptyset, GID \rightarrow\ PID\ functor\} \tag{3.4}$$

The distribution of a container can now easily be replaced by providing the triply-nested view defined by Equations 3.2, 3.3, and 3.4. The first `pView`, described in Equation 3.2, is the system `pView`. It provides an abstraction over the actual system, with a set of locations, the number of locations, and the unique location IDs (LID). The second `pView`, described in Equation 3.3, is the partition view: it provides an abstraction over the first view, it is also providing the knowledge of the total number of partitions for the container (i.e. the $Partition_{Domain}$) and a partition ID (PID) to LID mapping function. The last `pView`, described in Equation 3.4, is the elements view: it provide an abstraction over the second view, it is also providing the knowledge of the domain of the `GID`s and a `GID` to PID mapping function.

The user does not need to create these three views herself, the interface in STAPL gives access to the most common distributions in a single line of code (cyclic, block, block-cyclic) as well as an arbitrary distribution, where the user specifies the `GID`s' domain, the $GID \rightarrow\ PID\ functor$, and the $PID \rightarrow\ LID\ functor$. The rest is taken care of by the framework. These are the most important components for the user: the $GID \rightarrow\ PID\ functor$ is the functor used by the `pContainer`'s Partitioner to figure out in which Partition a `GID` has to go, while the $PID \rightarrow\ LID\ functor$ is the functor used by the `pContainer`'s Mapper to figure out on which location a partition is supposed to be located.

Such a view-based distribution can be used to initialize a `pContainer`'s distribution and it is also used to trigger a redistribution.

For instance a user will simply express a new view-based distribution in C++ by passing the two functors we described above:

```
auto vb_ds = stapl::arbitrary(
      container_domain,
      stapl::get_num_locations(),
      GID_to_PID_functor,
      PID_to_LID_functor);
```

The container's domain contains information about the set of elements that the container can hold. Then, the interface to redistribute a `pContainer` is simply:

```
my_container.redistribute(vb_ds);
```

In the next section we are going to see in details what happen once the redistribution is triggered.

3.3 Parallel Container Redistribution

Whenever a `pContainer` needs to be redistributed, both the directory and container managers need to be updated. The directory's manager needs to be updated with the new partition and mapper, which is easy due to the view-based distribution described in the previous section. Its registry needs to ensure that `GID`s are registered on the correct location. Indeed, each key needs to be registered where the Partitioner and Mapper expects the key to be. This means that in addition of updating the container manager's base containers, that registry also needs to be updated.

To be able to aggregate communication, even for arbitrary distributions, we use a well-known and efficient all-to-all communication pattern referred as the *butterfly* communication pattern. It is a good way to share information among all locations in $O(\log n)$ steps, where n is the number of locations.

Algorithm 2 `pContainer` redistribution.

Require: A `pContainer`

```
Identify how many elements of the pContainer to send to other locations using the new distribution Partitioner and Mapper
Perform a butterfly all-to-all communication so that each location knows how many elements to expect
Ship the data to other locations in an aggregated message
while (there is still elements to receive from other locations) do
    Receive elements and add them to their respective new base containers
end while
Register the new elements
Advance epoch
```

Algorithm 2 shows the high level algorithm for the redistribution. It is based on recent previous work in STAPL where the butterfly was used for a different purpose. It was exchanging `GID`s sets, to figure out the exact set of `GID`s each location was about to receive. This is a sufficient strategy for containers where `GID`s are scalar, but it was undefined for non-scalar key types. To be generic and work with any `GID`s we extend the algorithm in the following manner: first each location checks for each of its registered `GID`s where they should move, and keeps a record of, according to the new partition-mapper, where each key is now supposed to go. It then proceeds to the *butterfly* algorithm, where as a post-condition each location has knowledge of exactly how many elements it is supposed to contain. A location then sends one asynchronous message to each location, with all the data to ship. This aggregation avoids the need to send an RMI per element. It then waits until it receives all RMIs (if any) and registers the new set of `GID`s on that location. It then increments its epoch by one.

Incrementing the epoch guarantees that the redistribution is fully completed before doing other work on the data, without requiring an expensive global synchronization. For instance, if a location l_1 proceeds to send new RMIs to location l_2 after its part of the redistribution is done, but before l_2 is completed, then l_2 will not execute these RMIs before it also increments its epoch (i.e. finish its part of the redistribution).

4. REDISTRIBUTION IN SRT USING STAPL

As we saw in section 2.2, the seismic ray tracing application was implemented using STAPL. To handle redistribution, we use the latest features from STAPL as described in the previous chapter. In this chapter, we first present the redistribution algorithm in `SRT`. Next, we discuss the data structures we use in `SRT`. Then, we present how the Recursive Coordinate Bisection technique is used to improve locality of reference with the final objective of improving the application's performance. Finally we discuss about the frequency of the redistribution in `SRT`.

4.1 Redistribution Algorithm

In section 2.2 we described the parallel wavefront construction. Algorithm 3 is an improvement over Algorithm 1, where we add the redistribution features. In [15], Jain mentionned that a redistribution could be perform but this was not implemented.

Algorithm 3 Parallel wavefront construction algorithm with redistribution.

Require: Earth model, mesh description, number of source and their position

Initialize the rays and the ray tubes in parallel using the user specified geometry
while (**true**) **do**
    for $i = 0$ to $\frac{dt_{ray\ tube}}{dt_{ray}}$
        parallel for each ray $\in$ collection of rays
            Trace the rays by one time step
            If the ray segments intersect a surface in the earth model, then create new ray segments if needed
        end parallel for
    end for
    parallel for each ray tube $\in$ collection of ray tubes
        Step the ray tube by one time step
        Interpolating/coarsening the ray tube if necessary
    end parallel for
    if No ray tube remains
        break
    end if
    if Load balancing needed
        Use load balancing algorithm
    end if
end while

Algorithm 4 SRT redistribution.

Require: A ray collection and raytube collection.

1: Compute imbalance ratio by finding the maximum load and the average load
2: **if** imbalance ratio greater than imbalance threshold **then**
3:     Compute better partition with Zoltan's `RCB`
4:     Redistribute ray collection
5:     Redistribute raytube collection
6: **end if**

Lines 16 and 17 of algorithm 3 are the new steps performed in our parallel wavefront construction algorithm. Algorithm 4 is the detailed algorithm of the redistribution in SRT. Line 16 of Algorithm 3 is the line 2 of Algorithm 4, and line 17 of Algorithm 3 is the line 3 through 5 of Algorithm 4. All the steps are performed by a `LoadBalancer` module. The `LoadBalancer` module has a simple interface presented below:

```
template <class Raytube_Collection_Type, class RayCollectionType>
class LoadBalancer
{
  LoadBalancer(RayCollectionType* r,Raytube_Collection_Type* rt);
  bool IsRedistributionNeeded();
  void FindBetterPartition();
  void Redistribute();
};
```

In the `SRT` application, the load balancer object is instantiated once. This generic interface lets the user experiment with different load balancing methods.

The `LoadBalancer` has access to the two data structures so that it can call the `redistribute` algorithm of STAPL when necessary. Internally it has to create the new distribution, using the view-based distribution interface presented in 3.2. In our case, we get a new arbitrary distribution, where the Partitioner has information gathered from the `RCB` decomposition. The method `IsRedistributionNeeded` is simply returning a Boolean indicating if a redistribution is needed or not. This is line 2 of Algorithm 4. In our case we compute the imbalance ratio, and if it crossed the threshold, it returns $TRUE$. The imbalance ratio and choice of this threshold will be discussed in section 4.4.

Line 3 of Algorithm 4 is the `FindBetterPartition` method. It calls the Zoltan RCB algorithm and gets the different cuts information. These cuts are stored in the functor that will be used as the Partitioner. More details about this step are given in section 4.3

Finally, the method at line 4 and 5 of Algorithm 4 are done by `Redistribute`. This method generates the new arbitrary view-based distribution and calls STAPL's redistribute method for the ray collection and the raytube collection.

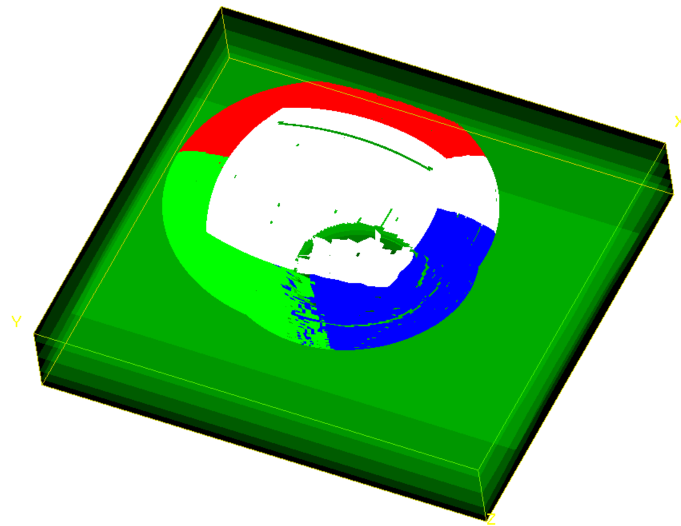


Figure 4.1: Visualization of the wavefront after redistribution in the salt dome model (p=4). Each color represents one of the four different processors.

In Figure 2.7 from section 2.2 we presented a visualization of the wavefront on four processors, we could see that one of the four locations was handling the majority of the rays. Figure 4.1 is the visualization of that same wavefront, in the same Model at the same time step using our new algorithm. We can notice that the four processors, in white, red, blue, and light green share rays more evenly. Rays are now evenly distributed among processors, which will decrease the wavefront overall propagation time.

4.2 Choice of Parallel Data Structures

In [15], Jain presents the original implementation of the parallel wavefront construction algorithm in STAPL. We present the data structures we used to store the rays and raytubes. These data structures rely on the view-based distribution to specifiy their data layout

4.2.1 Ray collection

We use the `pUnorderedMap` container to store the rays. The reasons behind this choice were simple: A parallel data structures was needed for the rays, the requirements were:

- Add rays in the container
- Support the interpolation of a new ray between existing rays
- Lookup rays quickly

As an associative container, the `pUnorderedMap` was a good choice since it let easily the user access to the rays based on its unique identifier: the 2D coordinates (r, c) of a ray. These (r, c) coordinates are unique coordinates obtained using the initial mesh decomposition presented in 2.2. The (r, c) are computed using the unique (γ_1, γ_2) coordinates of a ray. The Key/Value mapping provided by this associative container was a perfect fit here. Other containers such as the `pGraph` can provide this mapping as well, but since no relationship between the elements was required. In other words, there was no need for edges connecting the rays. For that reason, the simpler `pUnorderedMap` data structure was chosen.

The `pUnorderedMap` uses the `std::unordered_map` as its underlying base container, which is a hash table. It uses a hash function. In our case the hash function hashes the rays (r, c) coordinates. A hash function [19] is a function that can map the key to a fixed size digital datum. This fixed sized value, called hash code, can then be used to infer the location of the element in the hash table. The hash function needs to be good to ensure the $O(1)$ operations, but for 2D coordinates this is not a problem.

Table 4.1: Complexity of `pUnorderedMap` basic operations.

Container	`pUnorderedMap`
Insertion	$O(1)$
Lookup	$O(1)$
Traversal	$O(n)$

The `pUnorderedMap` is a generic data structure, the declaration of it is user friendly. The type for the key and value needs to be provided, as well as three parameters: The hash function, and the view-based partitioner and mappers that were described in section 3.2.

```
template <class Ray, class Interpolation_Manager>
class ray_collection
      : public p_unordered_map<RayId,
                                Ray,
                                hash<Ray_Id>,
                                view-based_Partition,
                                view-based_Mapper>
```

As for the STL, The genericity of STAPL let us specify the types without too much code. Tracing the rays in another important step in the algorithm, and this is done using a `pAlgorithm` from STAPL. The complexity of the operation is an $O(\frac{n}{p})$ operation, when n is the `pContainer` size and p is the number of locations. This is true in theory, but this assumes that each location holds $O(\frac{n}{p})$ elements. Indeed, each location takes care of tracing the rays it holds locally, but if we were to be in the degenerate case where one location holds $O(n)$ elements and all the other locations hold $O(1)$ elements, then the complexity of the tracing step would be $O(n)$, which is not better than sequential. In practice, the running time of the tracing is bounded by the most loaded location, as we explained in section 2.2.

4.2.2 Raytube collection

The raytube collection is using STAPL's dynamic graph: the `pGraph`.

```
template <class Raytube_Type>
class raytube_collection
    : public p_graph<DIRECTED,
                     MULTIEDGES,
                     Raytube_Type,
                     int,
                     view-based_Partition,
                     view-based_Mapper>
```

The `pGraph` needs to know about the Raytube type, and we use edges to represent the parent-child relation between raytubes and children raytubes. These children raytubes are the one created when an interpolation in required. We also pass the view-based partitioner and mappers that were described in section 3.2.

As every `pContainer`, the `pGraph`vertices are identified using a unique `GID`. This is a scalar called vertex descriptor. In this thesis, we use a specific formula to choose the `GID` of a descriptor. We do this such that the view-based partition of the graph can take advantage of the view-based partition we determined for the `pUnorderedMap` and which we will describe more in details in section 4.3.

In order to reuse the same decomposition for the raytubes and the rays, we needed a way to map the raytubes `GID`s (called descriptors) which are represented as scalars in the `pGraph` to the same location where most of its rays are. For this, we use a combination of a raytube's top-left ray's r and c coordinates, as well as the raytube's current level of interpolation to encode a unique raytube descriptor. This technique makes the reuse of the `RCB` decomposition for both the rays and raytubes.

The formula used to encode the raytube's descriptor is presented in equations 4.1 to 4.4

$$1D\ ray\ id = r * number\ of\ columns * c \tag{4.1}$$

$$shift\ of = \log_2(Descriptor_{MAX\ VALUE}) - \log_2(Max\ interpolation\ level) + 1 \tag{4.2}$$

$$shifted\ level = current\ interpolation\ level \ll shift\ of \tag{4.3}$$

$$raytube\ descriptor = ray\ id \mid shifted\ level \tag{4.4}$$

As we can see in equation 4.1, we first generate a 1D representation of the 2D (r, c) pair. It is easy since the maximum column number (c_{max}) is known. Then we reserve a few bits to encode the interpolation level of the raytube. That way we will not have conflicts of descriptor unicity between raytubes of different interpolation level but who take care of the same top-left ray. This is shown in equations 4.2 and 4.3. Finally, as we can see in equation 4.4, we get a unique raytube descriptor by combining the two sets of bits from equations 4.1 and 4.3 using a simple "OR" operation.

When it comes to using the ray partitioner for the raytube, we find the original (r, c) coordinates from the descriptor with the following information. Equation 4.5 gives us a mask of the useful bits to find the (r, c) coordinates. The other bits contains the interpolation level information, and are not useful here. They are only present to guarantee a unique descriptor for that particular raytube.

$$useful\ bits = D_{MAX\ VALUE} \gg \log_2(Max\ interpolation\ level) + 1 \tag{4.5}$$

$$1D\ ray\ id = raytube\ descriptor\ \&\ useful\ bits \tag{4.6}$$

$$r = \frac{1D\ ray\ id}{number\ of\ columns} \tag{4.7}$$

$$c = (1D\ ray\ id) \bmod (number\ of\ columns) \tag{4.8}$$

From these useful bits mask, we find the ray id expressed in one dimension using equation 4.6. With trivial mathematical operations we can find easily (r, c) as shown in equations 4.7 and 4.8

4.3 Usage of Recursive Coordinate Bisection

`RCB` will help us easily define partitions that contain approximatively the same number of rays. It also gives us an advantage concerning the raytubes: some raytubes will have to read remote rays; these raytubes are the raytubes on the boundaries of a processor. For these raytubes, it is possible to minimize the number of remote calls between raytube and rays by keeping the raytubes aligned with the rays. The solution chosen here is to always place a raytube where its top-left ray is. If we redistribute raytubes without regard to the rays' distribution, the number of remote calls could be much higher.

Since each ray is represented by 2D coordinates, the Recursive Coordinate Bisection method was a good fit to this problem. We have a higher chance of having rays that are physically close to each other in the same partition. We use `Zoltan`'s `RCB` implementation to generate p areas, where p is the number of locations in our STAPL program.

Once we have a good RCB decomposition, we give this decomposition's information to the pContainer's partitioner, which will use the different areas solved by the RCB algorithm. During the interpolation phase, if new rays and raytubes need to be added, they will be added according to these areas.

At each time step, interpolation and coarsening will happen, changing the load per location. The load-balancing analysis in combination with STAPL's redistribution algorithm will take care of balancing the application.

4.4 Frequency of Redistribution

Redistribution comes with a cost: analyzing the load and generating a new RCB decomposition adds some overhead t_a. Performing the redistribution itself, adds some extra overhead t_r caused by determining how to aggregate and move the data around. All useful work (propagation, interpolation, coarsening, etc.) step takes t_c. Without redistribution, t_c can be higher due to the fact that one over loaded processor could potentially do most of the work while other processors sit idle. With redistribution, t_c should be smaller. If $t_a + t_r$ for that step is larger than the improvement for t_c at the next step, then the overall running time might actually be worse. To solve this we propose to analyze the average and the maximum load of each processor. From these measures we compute the imbalance ratio as presented in equation 4.9. The maximum load is the number of rays on the location that has the highest count of it. The average load is the sum of all the rays divided by the number of locations.

$$imbalance\ ratio = \frac{maximum\ load}{average\ load} \tag{4.9}$$

If the imbalance ratio crosses a certain threshold, then the RCB analysis and the redistribution itself are triggered as we shown in section 4.1

We will discuss in the next Chapter the impact of choosing a different imbalance ratio on the overall running time.

5. PERFORMANCE EVALUATION

We presented the design and implementation of our load balance technique for `SRT`. In this chapter, we discuss the performance of the application on a Cray XE6m cluster we use at the Parasol Lab at Texas A&M University. We focus on the propagation phase of the algorithm. The initialization has no load imbalance problems, and was shown in [15] to be negligible compared to the overall runtime. All experiments were ran at least 32 times, and confidence interval levels of 95% are shown when possible. We first introduce the clusters hardware and software specifications on which the experiments were run. Next, we present the three different models that were used in `SRT`. We then present the results. We discuss how the change of the imbalance threshold impacts performance. We briefly discuss the impact on memory of our redistributed version. Then we discuss about strong scaling experiments, scalability, and speedup of our `SRT` implementation compared to the imbalanced-`SRT`.

5.1 Machine Specifications

We ran experiments on Rain, a Cray XE6m cluster. Table 5.1 shows the hardware configuration for the cluster, it has 576 compute cores connected in a 2-D torus fashion. Given the communication pattern in SRT, where the mapping is done on a mesh, the 2-D torus configuration suits us well.

Table 5.1: Cray XE6m (aka. Rain) 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
Average memory per core	2 GB
Interconnect	1 Gemini routing and communication ASIC per two compute node. 48 switch ports per Gemini chip (160GB/s internal switching capacity per chip). 2-D torus interconnect

Table 5.2 shows the software installed when we ran experiments, g++4.9 was our C++ compiler, and when Zoltan was used it was part of Trilinos 11.12.1.1. STAPL relies on the C++ Boost library, on this machine version 1.56 was used.

Table 5.2: Cray XE6m (aka. Rain) software specifications.

OS	Cray Linux Environment
Compilers	Cray g++ version 4.9.2
MPI	version 2.0
Libraries	Cray Trilinos version 11.12.1.1 Boost version 1.56

We also ran experiments on Vulcan, an IBM BlueGene/Q cluster at Lawrence Livermore National Laboratory. We can see from Table 5.3 that the configuration is different from Rain: we have access to more cores, but each core is less powerful, and on average each core can only use up to 1GB of memory. In Table 5.4 we notice that the latest compiler we can use is not as up to date as Rain: g++4.7.2 was released in 2012, while g++4.9 in 2014. The support for the C++11 features is better in the latest version.

Table 5.3: IBM BlueGene/Q (aka. Vulcan) hardware specifications.

Total number of cores	392,216
Processor Type	1.6 GHz PowerPC A2 processors
Memory	16 GB per compute node
Average memory per core	1 GB
Interconnect	5-D torus interconnect

Table 5.4: IBM BlueGene/Q (aka. Vulcan) software specifications.

OS	PowerPC 64 Linux Environment
Compilers	g++ version 4.7.2
MPI	version 2.1
Libraries	Trilinos version 12.0.1 Boost version 1.53

5.2 Input Parameters

In the next section we will present results for three different earth models The first one is the most interesting, as it is the most realistic model and also exhibits the highest ratio of imbalance. We present that model in Figure 5.1, and then we will show how the other models follow the same performance trends.

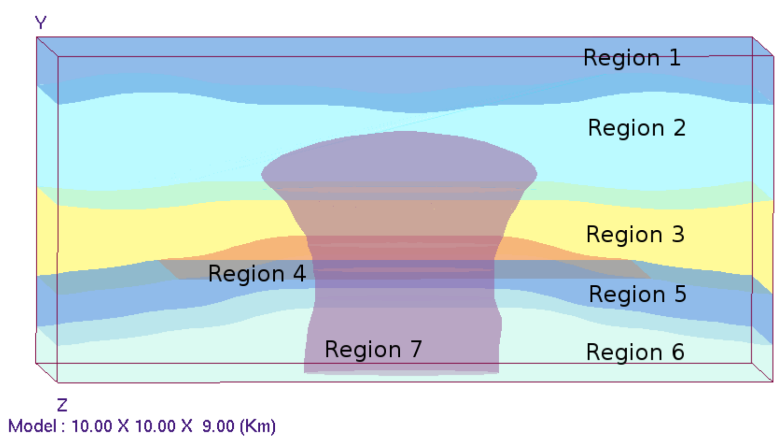


Figure 5.1: Model 1: Salt dome model [15].

Figure 5.1 is that realistic model, the high number of different regions with different properties makes the ray diverge multiple times through the execution. It is composed of seven different regions. Six of them are layered and the seventh region is a salt dome in the middle of the area. The characteristics of these regions are presented in Table 5.5

Table 5.5: Model 1: Salt dome model regions [15].

Region 1	V_p=2.7 km/s, V_s=1.5 km/s, ρ=2.55
Region 2	V_p=3 km/s, V_s=1.73 km/s, ρ=2.5
Region 3	V_p=3.2 km/s, V_s=1.8 km/s, ρ=2.55
Region 4	V_p=3.3 km/s, V_s=1.9 km/s, ρ=2.7
Region 5	V_p=3.4 km/s, V_s=1.9 km/s, ρ=2.67
Region 6	V_p=3.6 km/s, V_s=2.1 km/s, ρ=2.7
Region 7	V_p=4.78 km/s, V_s=2.7 km/s, ρ=2.2

In Table 5.6 we explain the different parameters used in our test, these can be changed by the user.

Table 5.6: Model 1: Salt dome model simulation settings.

Wave type	P
Initialization	Cube sphere
Time step for tracing	0.02
Time step for propagation	0.04
Maximum interpolation level	20
Patch test	Single Sided Balanced
Patch test upper threshold	0.0005
Patch test lower threshold	0.00005
Interpolation method	Central finite difference

The following models are simpler models, but also benefited from redistribution Figure 5.2 is also realistic. It represents a salt canopy configuration. The seven regions are presented in Table 5.7 and the simulation settings are shown in Table 5.8.

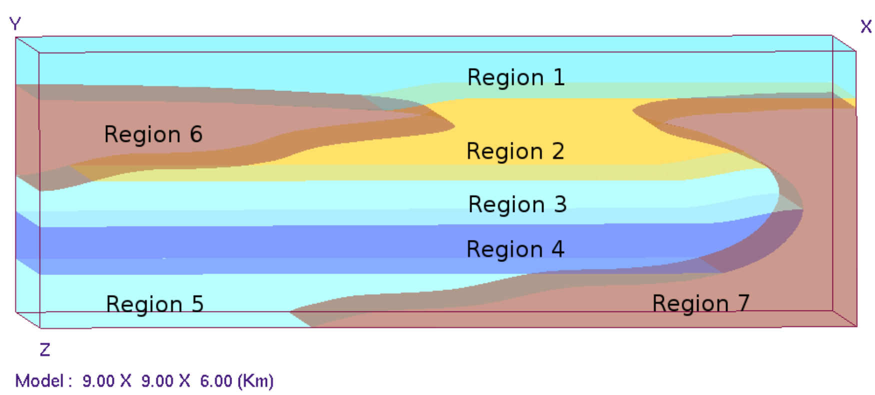


Figure 5.2: Model 2: Salt canopy model [15].

Table 5.7: Model 2: Salt canopy model regions [15].

Region 1	V_p=3 km/s, V_s=1.73 km/s, ρ=2.5
Region 2	$\begin{pmatrix} 20.28 & 13.104 & 15.028 & 0. & 0. & 0. \\ 13.104 & 20.28 & 15.028 & 0. & 0. & 0. \\ 15.028 & 15.028 & 22.542 & 0. & 0. & 0. \\ 0. & 0. & 0. & 4.498 & 0. & 0. \\ 0. & 0. & 0. & 0. & 4.498 & 0. \\ 0. & 0. & 0. & 0. & 0. & 3.588 \end{pmatrix}$ ρ=2.4
Region 3	$\begin{pmatrix} 25.9 & 6.825 & 7.075 & 0. & 0. & 0. \\ 6.825 & 25.9 & 7.075 & 0. & 0. & 0. \\ 7.075 & 7.075 & 23.775 & 0. & 0. & 0. \\ 0. & 0. & 0. & 7.325 & 0. & 0. \\ 0. & 0. & 0. & 0. & 7.325 & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.525 \end{pmatrix}$ ρ=2.5
Region 4	V_p=3.4 km/s, V_s=1.83 km/s, ρ=2.67
Region 5	V_p=3.8 km/s, V_s=1.9 km/s, ρ=2.7
Region 6 & 7	V_p=4.78 km/s, V_s=2.7 km/s, ρ=2.2

Table 5.8: Model 2: Salt canopy model simulation settings.

Wave type	P
Initialization	Cube sphere
Time step for tracing	0.02
Time step for propagation	0.04
Maximum interpolation level	20
Patch test	Single Sided Balanced
Patch test upper threshold	0.0001
Patch test lower threshold	0.00002
Interpolation method	Central finite difference

Figure 5.3 is more regular, and not as realistic. It is an earth model of a simple region with two cylindrical inclusion. The three regions are presented in Table 5.9 and the simulation settings are shown in Table 5.10. We will see how this model is less prone to imbalance will benefit less from redistribution.

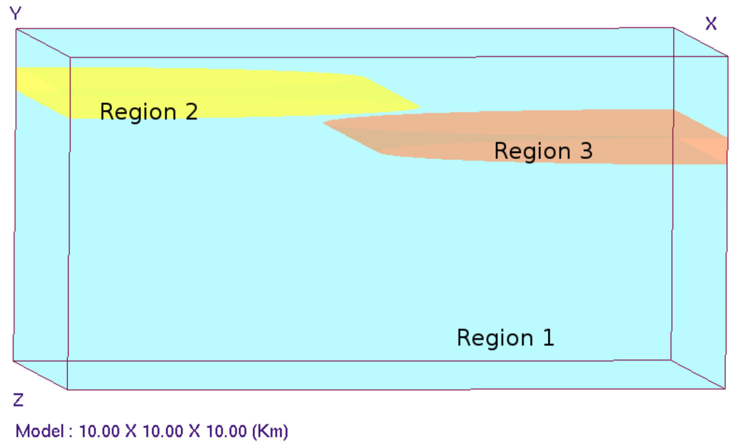


Figure 5.3: Model 3: Cylindrical inclusions model [15].

Table 5.9: Model 3: Cylindrical inclusions model regions [15].

Region 1	$\begin{pmatrix} 25.9 & 6.825 & 7.075 & 0. & 0. & 0. \\ 6.825 & 25.9 & 7.075 & 0. & 0. & 0. \\ 7.075 & 7.075 & 23.775 & 0. & 0. & 0. \\ 0. & 0. & 0. & 7.325 & 0. & 0. \\ 0. & 0. & 0. & 0. & 7.325 & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.525 \end{pmatrix}$ ρ=2.5
Region 2	V_p=2.8 km/s, V_s=1.5, ρ=2.6
Region 3	V_p=2.8 km/s, V_s=1.5, ρ=2.6

Table 5.10: Model 3: Cylindrical inclusions model simulation settings.

Wave type	P
Initialization	Cube sphere
Time step for tracing	0.02
Time step for propagation	0.04
Maximum interpolation level	20
Patch test	Single Sided Balanced
Patch test upper threshold	0.0001
Patch test lower threshold	0.00002
Interpolation method	Central finite difference

5.3 Results

In this section we will first study how the imbalance threshold choice can impact performances, then we look at the memory usage, and finally we show different running time results. The imbalanced version we experiment against is `SRT` with the latest `pContainer`, but with no redistribution overhead at all.

5.3.1 Choice of imbalance threshold

We tried different imbalance threshold and studied the improvement over SRT without redistribution. Figure 5.4 shows the improvement of the propagation time when redistributing versus the original SRT. The results have been scaled from 0 to 1 so that we can show a clear trend for the imbalance ratio. As we expected, picking a ratio of 1.0 is not optimal, as we pay the cost of redistributing too often. In our experiments, a ratio between 1.2 and 1.8, regardless of the processor count, was optimal. In the case of Forward modeling, where the user will refine the model iteratively, every model will have a similar physical layout for which finding the best threshold will be easy.

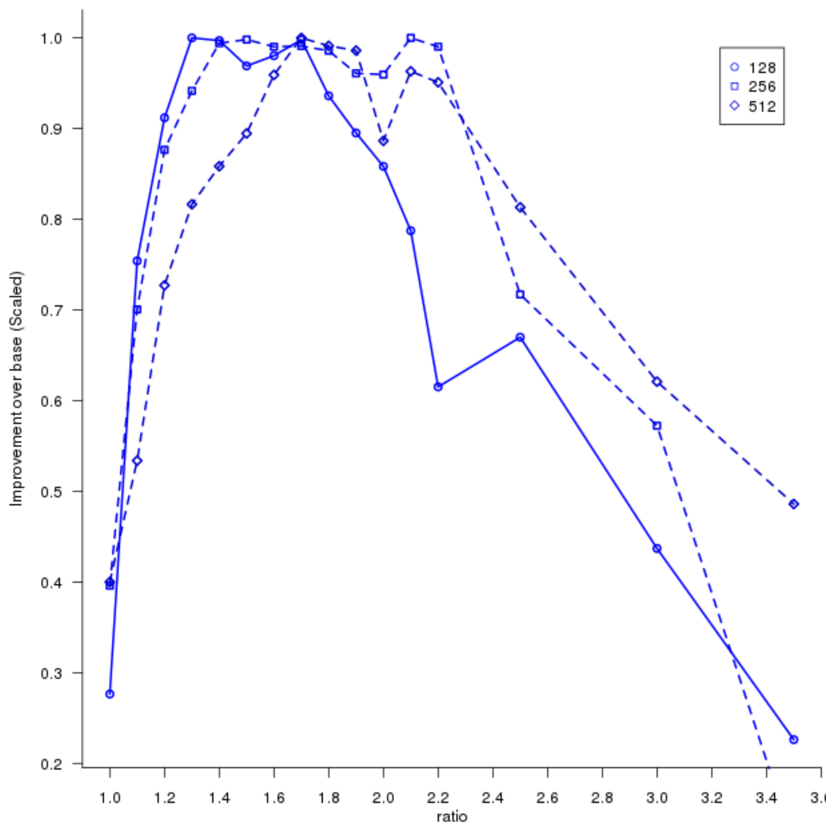


Figure 5.4: Speedup of redistributed (th=1.6) vs. imbalanced. Scaled from 0 to 1 to exhibit the best threshold.

Figure 5.5 shows the actual speedup at a fixed threshold of 1.6. We can see that we always get improvement. The variation of improvement will be discussed when we show scalability results in section 5.3.3.

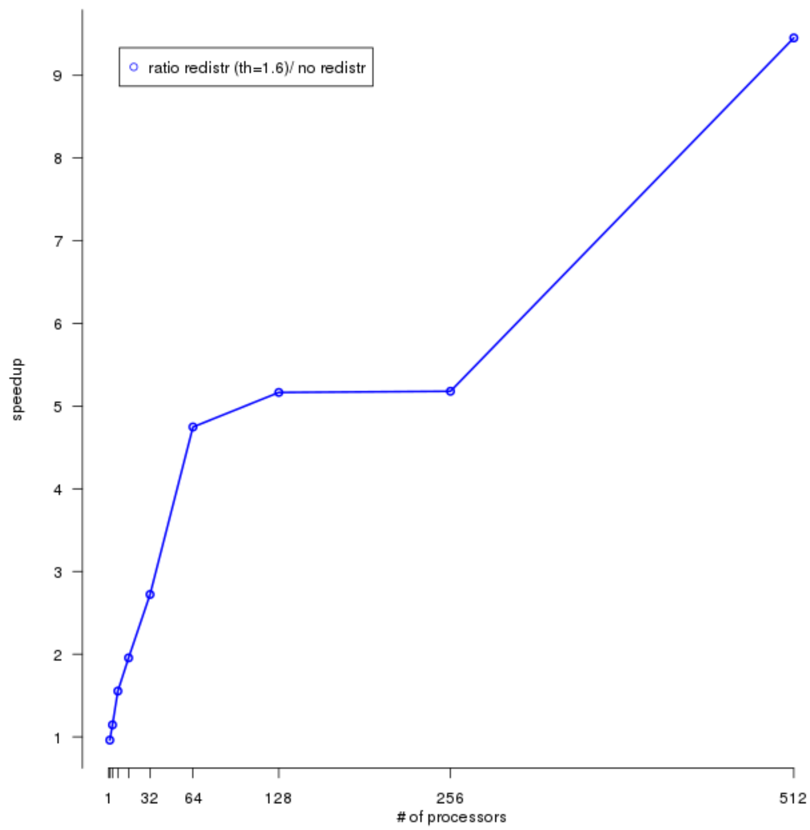
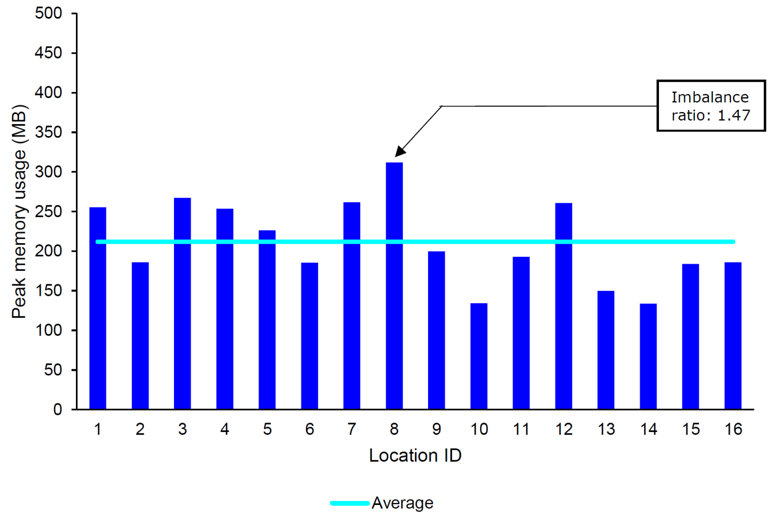
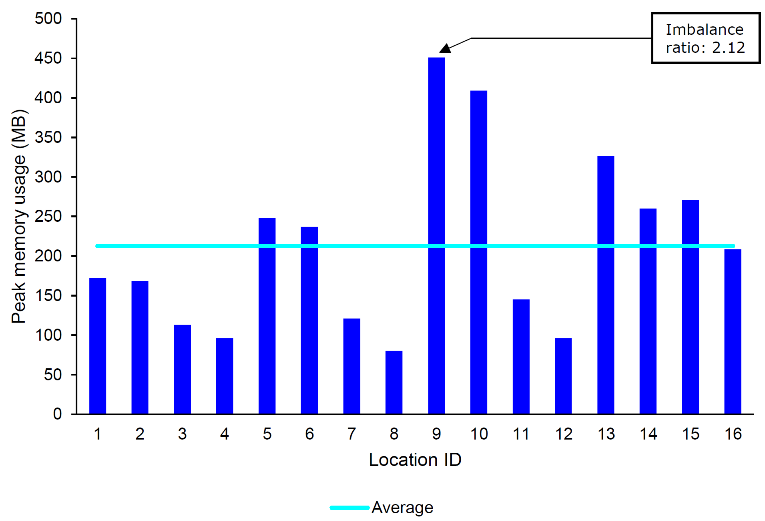


Figure 5.5: Speedup of redistributed version (th=1.6) versus imbalanced version in Model 1 on Rain.

5.3.2 Memory usage



(a)



(b)

Figure 5.6: Salt dome model peak memory usage with 16 locations on Universitas (a) with redistribution (th=1.6) and (b) without redistribution.

Memory usage is an important factor when running an application, in the previous section we show that each core has around 2GB of RAM available. If all the cores on a node became highly overloaded, the application would terminate. With redistribution, Figure 5.6 shows a memory analysis using the tool Massif from the Valgrind suite. It lets us analyze the heap memory usage of our application. In both runs the average memory usage was 212 MB. With redistribution we can see that the maximum imbalance ratio of memory is 1.47 with a location reaching 312 MB. Without redistribution, a processor peaks at 451 MB, giving us an imbalance ratio of 2.12. Massif was not available on Rain, and the instrumentation slows down and requires a lot of extra memory, for that reason we decided to run it on Universitas, a local shared-memory machine on 16 cores, with 64GB of RAM.

Depending on which machine the application is ran on, memory consumption can have an important impact, for instance the Vulcan supercomputer at Lawrence Livermore National Laboratory only has 1 GB of RAM per core, so if we decide to increase the precision of our algorithm, we need to make sure that some nodes are not going to be overloaded.

5.3.3 Performance results

In Section 5.3.1 and above we compared the improvements of Model 1 with a specific maximum interpolation level set to 3, limiting the number of interpolation a raytube can perform to at most three levels. This interpolation level is used in SRT's implementation to guarantee the uniqueness of rays and raytubes `GID`s. With the current implementation, that maximum level can be raised to 22, which means that each raytube could interpolate 22 times, which would create 4^{22} children raytubes for each original raytube. The maximum of 22 is set by the size in bits of a `long unsigned int` which is 64 bits. If in the future this becomes a problem, that size could be easily changed to more bits, 128 bits for instance. In that case a raytube could interpolate 53 times, creating 4^{53} children raytubes.

However, this limit can also be used to artificially limit the interpolation in order to reduce memory usage. Indeed, if a system is limited in memory, without redistribution, a processor could easily run out of memory if it decides to interpolate too many times.

Theoretically, raising the maximum interpolation level by k can make a location interpolate and produce 4^k times more raytubes. Indeed, each raytube creates 4 children raytube at each interpolation phase. The number of rays only increase by a factor of 1.6^k or 2.6^k if we use the raytube with four rays or raytube with five rays version. This means the actual memory footprint increases at a lower rate than the work. Even though we grow only by a factor of 2.6^k the number of rays, each new raytube has to read the value of four rays, which means that, ignoring any caching mechanisms, there would be 4^k more work.

This exponential growth is a major problem, a good maximum level must be chosen by the geophysics user, or a higher error threshold must be chosen to limit interpolation. This depends on what the user expects for her results. In our experiments we set that level high enough to let the algorithm interpolate as much as required by the model settings.

We now present various results for our three models, we will present and discuss strong scaling experiments, scalability measurements, speedup, and a breakdown of the main steps of the application.

5.3.3.1 Model 1 - Salt dome

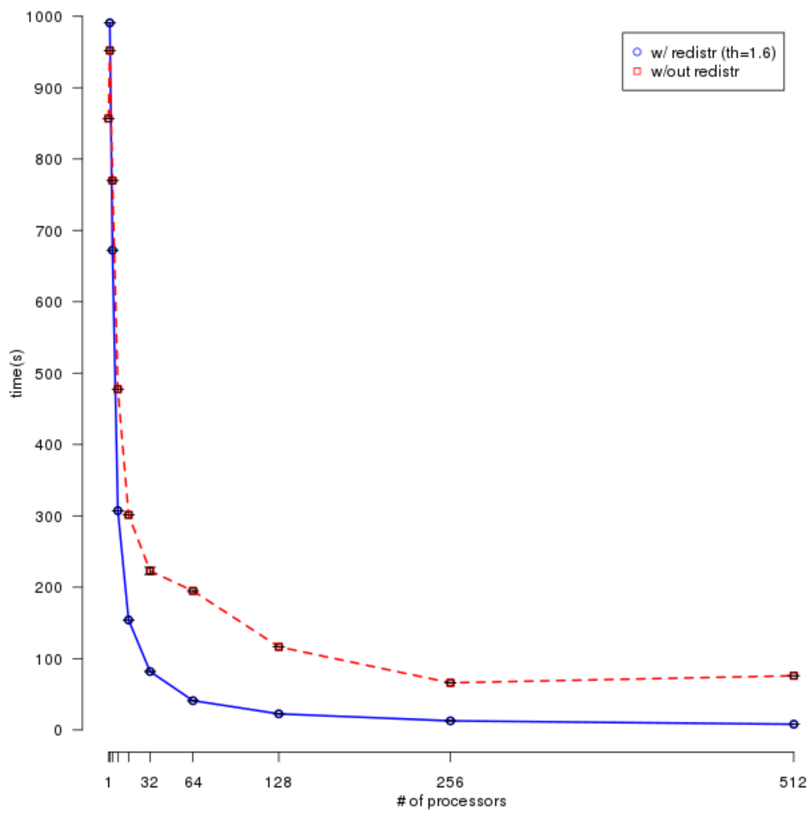


Figure 5.7: Strong scaling of the redistributed (th=1.6) and imbalanced version in Model 1 from 1 to 512 processors on Rain. Confidence Intervals 95%.

Figure 5.7 shows the strong scaling of the application from 1 to 512 processors, we can see that the trend with and without redistribution is around the same. Figure 5.8 is a zoomed view from 128 to 512 processors we can see that at 512 processors we are almost six times faster. We can see that the benefits of redistribution increase as the processor count increases. At 32 processors we went from 222 seconds to 81 seconds, giving us only a 2.75x improvement. However, at 512 processors we went from 76 seconds to 8 seconds, giving us a 9.5x improvement (or a 90% reduction in the running time).

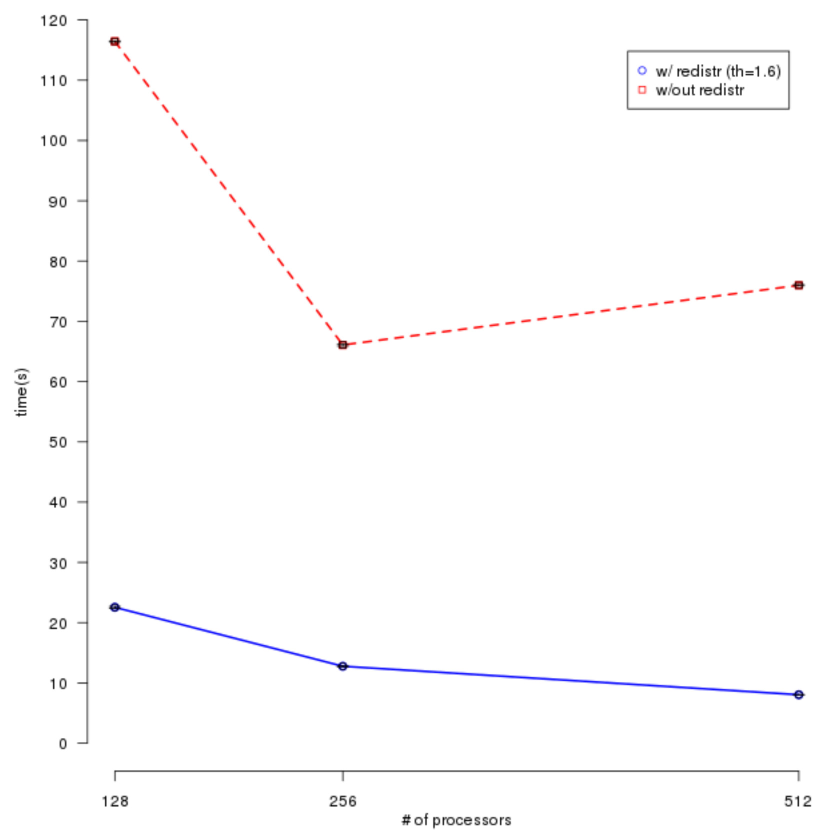


Figure 5.8: Strong scaling of the redistributed (th=1.6) and imbalanced version in Model 1 from 128 to 512 processors on Rain. Confidence Intervals 95%.

Figure 5.9 shows the scalability of the two versions. The redistributed version scales better and with more consistently. The reason the version without redistribution doesn't scale as regularly is because of the original distribution. Indeed, without redistribution, the original partitioning is done such that each processor start with a block of the same size, with an equal number of rays and raytubes. Depending on the processor count, the block size will be different. Even though these blocks hold the same number of initial rays and raytubes, it is does not take into account which raytubes are going to interpolate often. Sometimes the original partitioning will have a better distribution of the raytube once they decide to interpolate. In our experiments, the non redistributed version cannot scale better than 10, this is consistent with the results from [15]. This lack of regularity of the non-redistributed version explains the speedup plot from figure 5.5.

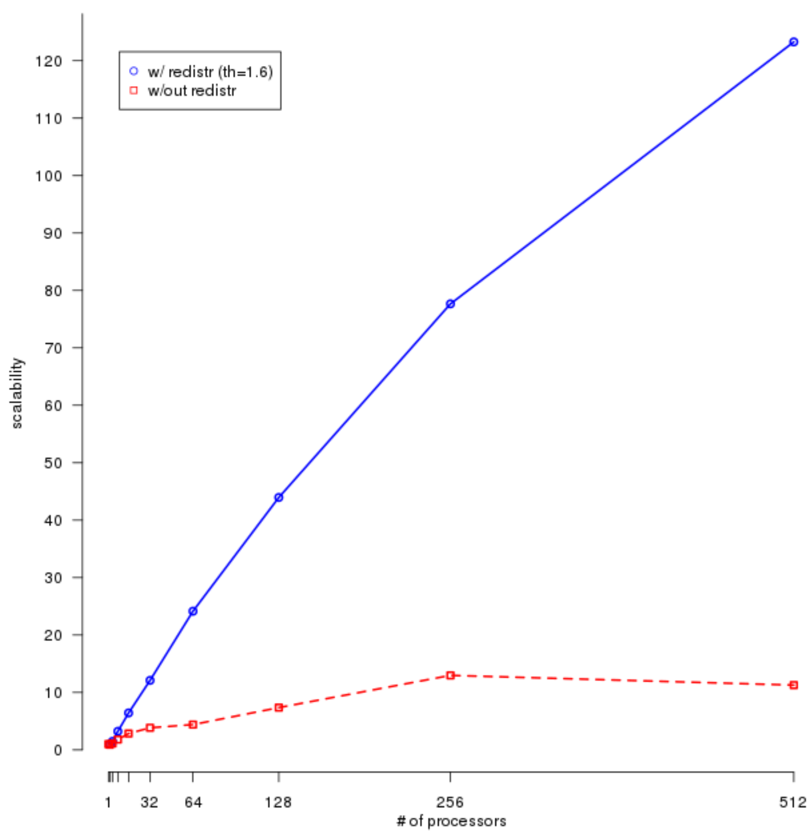


Figure 5.9: Scalability of the redistributed (th=1.6) and imbalanced version in Model 1 from 1 to 512 processors on Rain.

We now look into details in the distribution of time in `SRT`. Figure 5.10 shows that when we increase the number of processors the time taken by all the steps involved in redistributing the data increases. Since redistributing involves more communication than the wavefront propagation, this is expected.

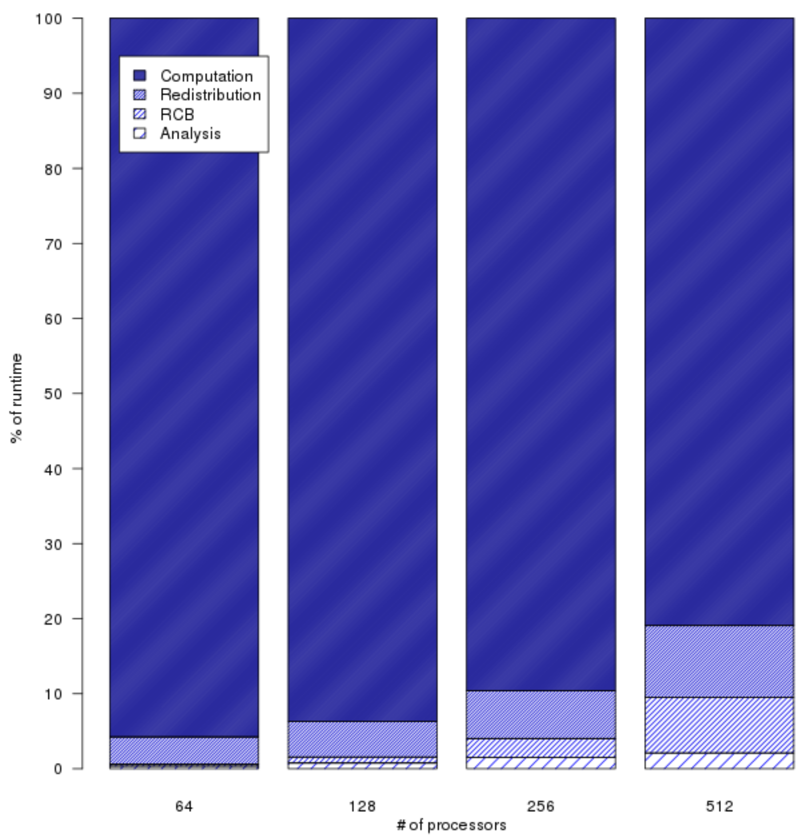


Figure 5.10: Distribution of time of the redistributed version(th=1.6) in Model 1 from 64 to 512 processors on Rain.

Figure 5.11 is a breakdown of the redistribution's steps, we notice that, as a percentage of the runtime, every step is becoming more and more important. When looking at the number of redistributions in Table 5.11, we can see that as we increase the number of processors, with the same fixed imbalance ratio threshold, the total number of redistribution increases.

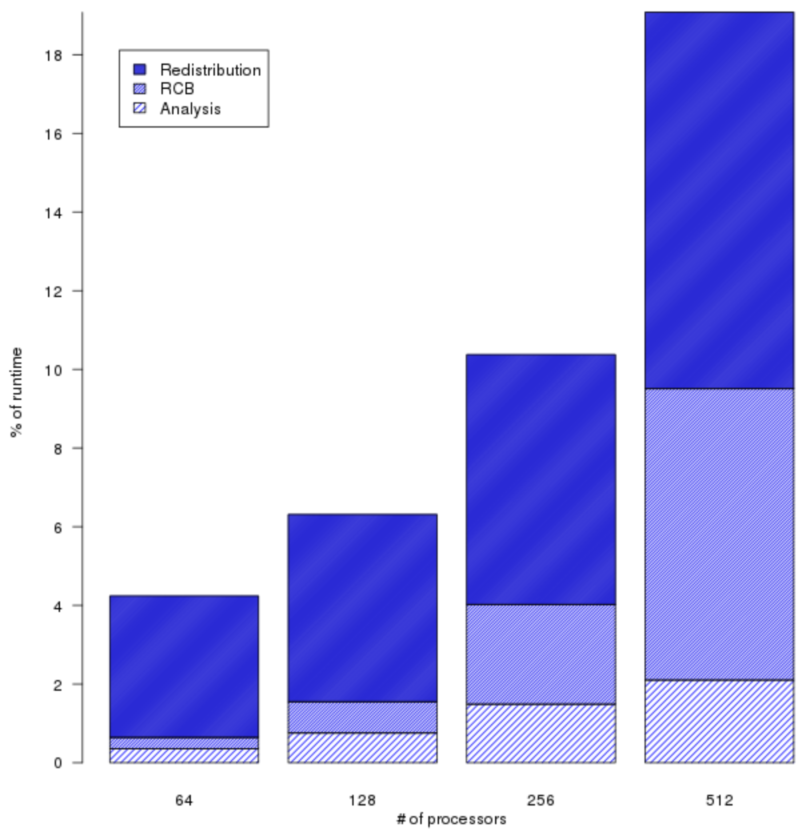


Figure 5.11: Distribution of redistribution phases as a percentage in Model 1 from 64 to 512 processors on Rain.

Figure 5.12 is the sane breakdown as Figure 5.11 but this time the y-axis is the actual runtime in seconds. We notice that the total time spent analyzing the imbalance remains flat. This is expected, as computing the imbalance ratio is a small reduction operation. On a higher processor count, this could become a bottleneck. The redistribution phase takes less time as we increase the number of processors from 64 to 256 processors, the reason is that every processor has less and less data to exchange. However, one of the steps of the redistribution algorithm is the all-to-all communication, which is likely to become a bottleneck at a higher processor count. We can see the time taken from 256 to 512 processors being almost the same. Finally, the Recursive Coordinate Bisection step takes more and more time, this is due to the fact that there is more redistributions performed, but if we look at the time taken for a single `RCB` decomposition, it also increases with the processor count. The reason is that we have more processors involved in the algorithm's communication phases.

Table 5.11: Frequency of redistribution in Model 1 (th=1.6).

Processor count	Number of redistributions
128	21
256	31
512	43

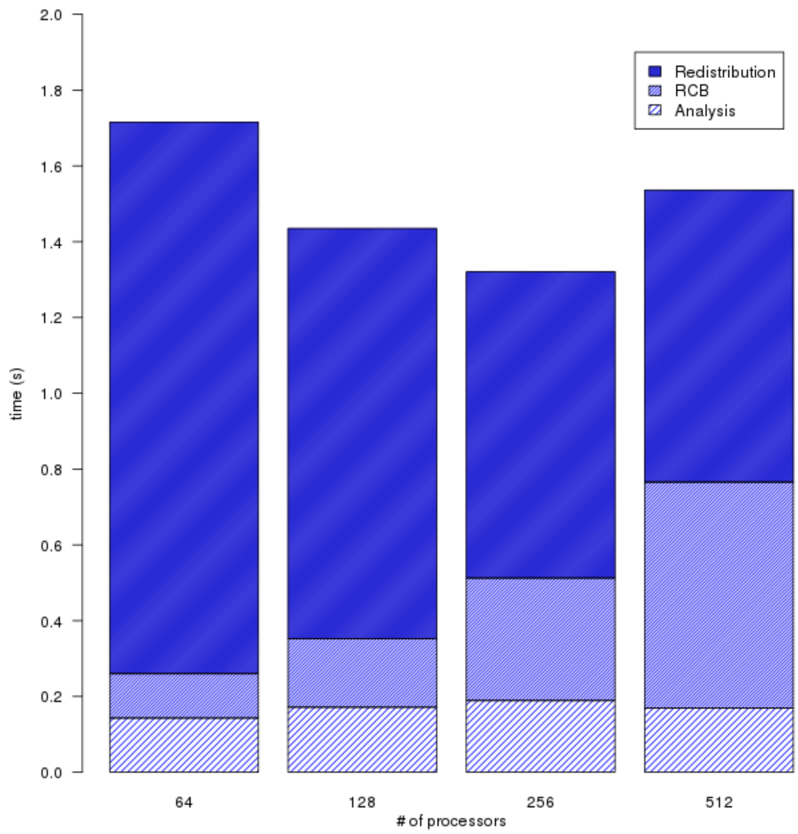


Figure 5.12: Distribution of redistribution phases in seconds in Model 1 from 64 to 512 processors on Rain.

We now look at the performance results obtained on Vulcan for the same Model. Figure 5.13 shows the running time from 128 to 1024 of the two same version we ran on Rain. We can see the same trend for runtime improvements. As on Rain, the non redistributed version initial partitioning is not favorable at 512 processors, it however gets better for 1024 processors, where the runtime is reduced by at least 200 seconds.

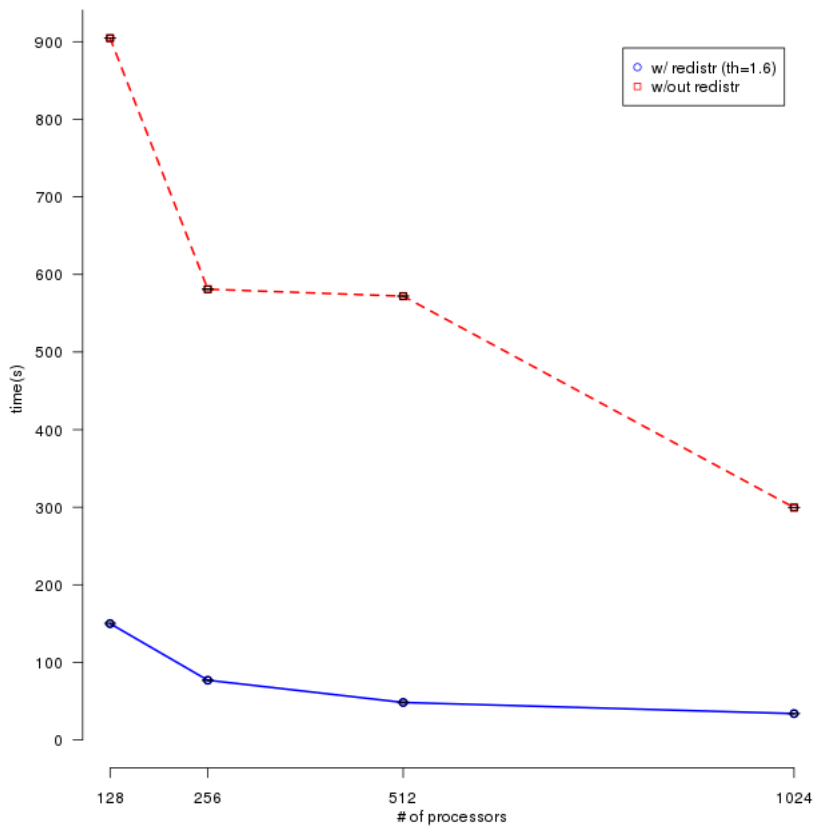


Figure 5.13: Strong scaling of the redistributed (th=1.6) and imbalanced version in Model 1 from 128 to 1024 processors on Vulcan. Confidence Intervals 95%.

Figure 5.14 shows the scalability of the two versions. The baseline is with 128 processors, which is represented as a scalability value of 1 one the plot. The redistributed version scales more consistently as we increase the processor count.

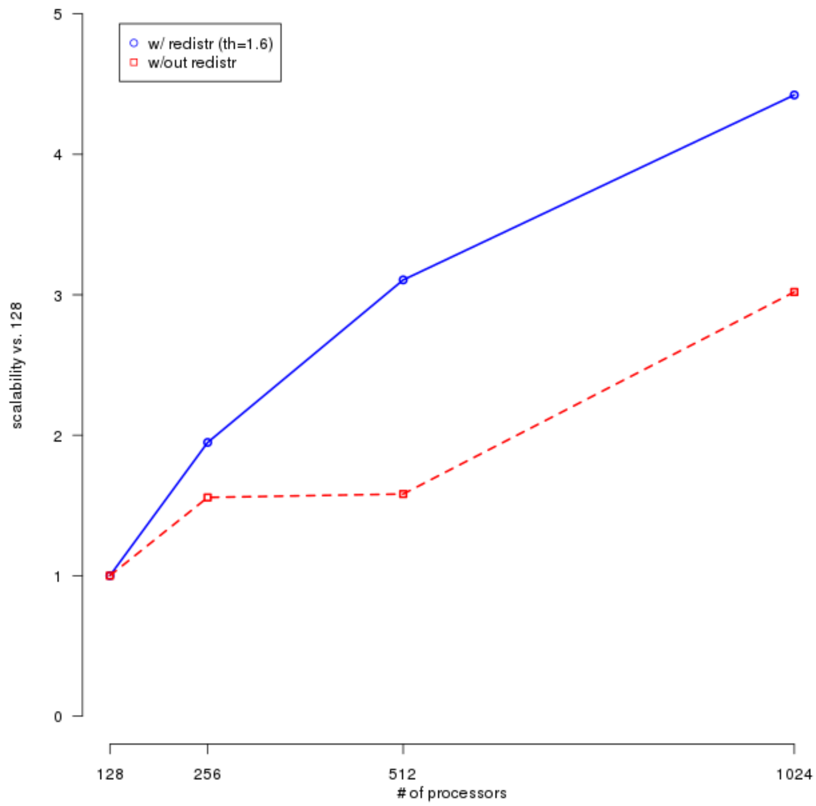


Figure 5.14: Scalability of the redistributed (th=1.6) and imbalanced version in Model 1 from 128 to 1024 processors on Vulcan.

Whenever the non redistributed version has a poor distribution, we can notice the best speedups. For instance in Figure 5.15, we notice a speedup of almost 12 at 512 processors. This speedup is smaller at 1024 processors, but the redistributed version is always faster.

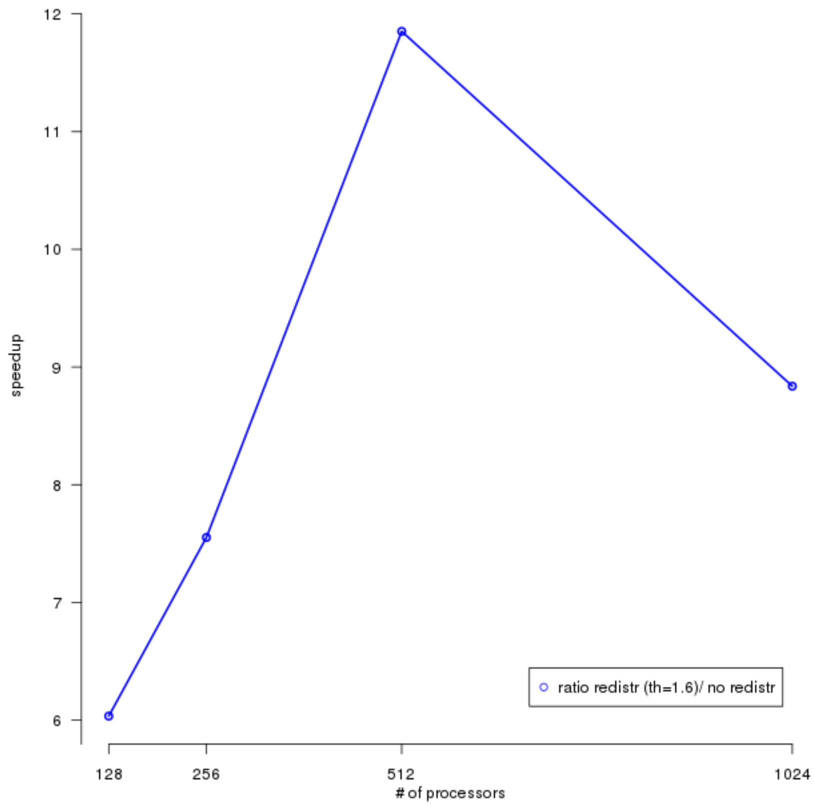


Figure 5.15: Speedup of redistributed version (th=1.6) versus imbalanced version in Model 1 from 128 to 1024 processors on Vulcan.

The breakdown of SRT's running time on Vulcan presented from Figures 5.16, 5.17, and 5.18 shows the same trends as the results on Rain. However, for the same processor count, the redistribution steps on Vulcan take less time than on Rain. This can be explained with the higher quality Interconnect from the BlueGene/Q cluster.

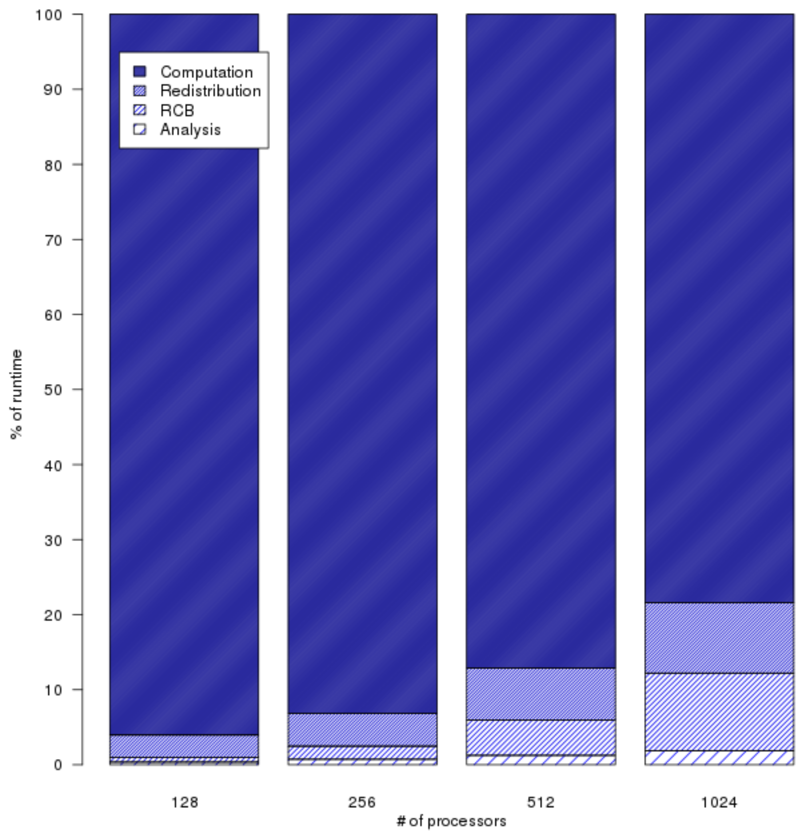


Figure 5.16: Distribution of time of the redistributed version(th=1.6) in Model 1 from 128 to 1024 processors on Vulcan.

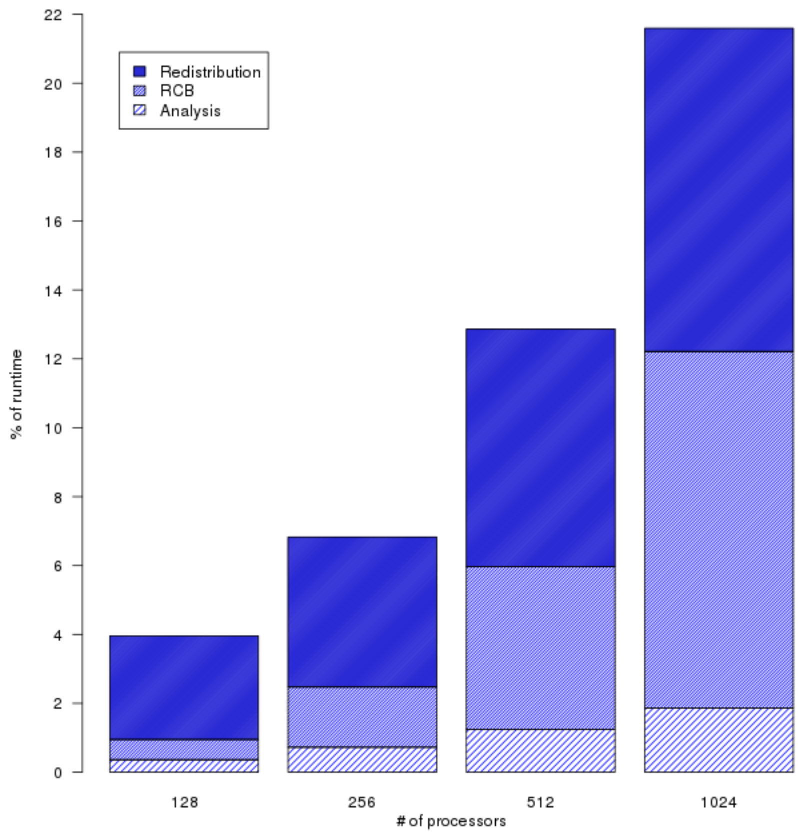


Figure 5.17: Distribution of redistribution phases as a percentage in Model 1 from 128 to 1024 processors on Vulcan.

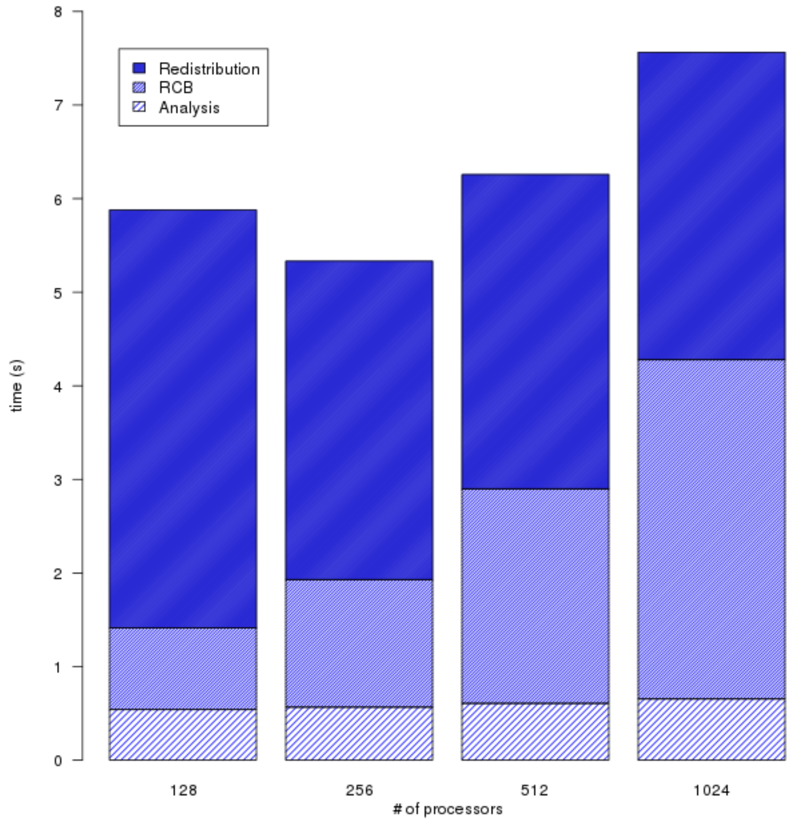


Figure 5.18: Distribution of redistribution phases in seconds in Model 1 from 128 to 1024 processors on Vulcan.

In Figure 5.18 we notice the same trend for actual seconds spent on each phase, and we actually see that the redistribution almost stay the same from 256 to 1024 processors. As discussed previously, the redistribution is likely to take more time with more processors.

5.3.3.2 Model 2 - Salt canopy

Model 2, as described in section 5.2, is not as complex as Model 1. This decrease in complexity makes the rays less likely to interpolate, which is good for the original non redistributed version. Figures 5.19 and 5.20 shows the redistributed version is still faster, but not too far from the original version.

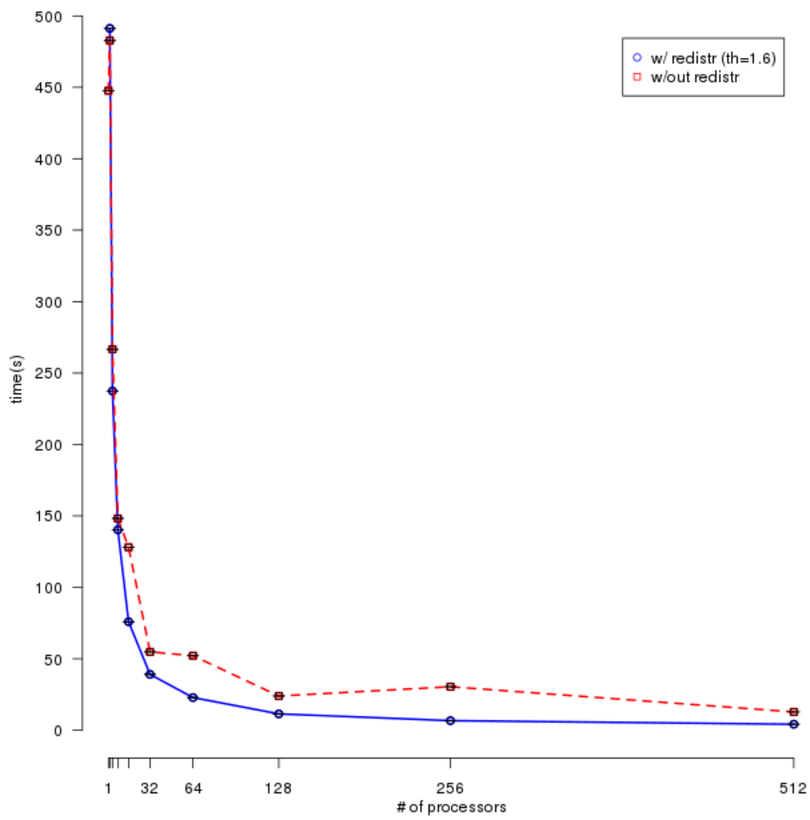


Figure 5.19: Strong scaling of the redistributed (th=1.6) and imbalanced version in Model 2 from 1 to 512 processors on Rain. Confidence Intervals 95%.

However, we can see in Figure 5.21 that the redistributed version scales better and more consistently than the non-redistributed version.

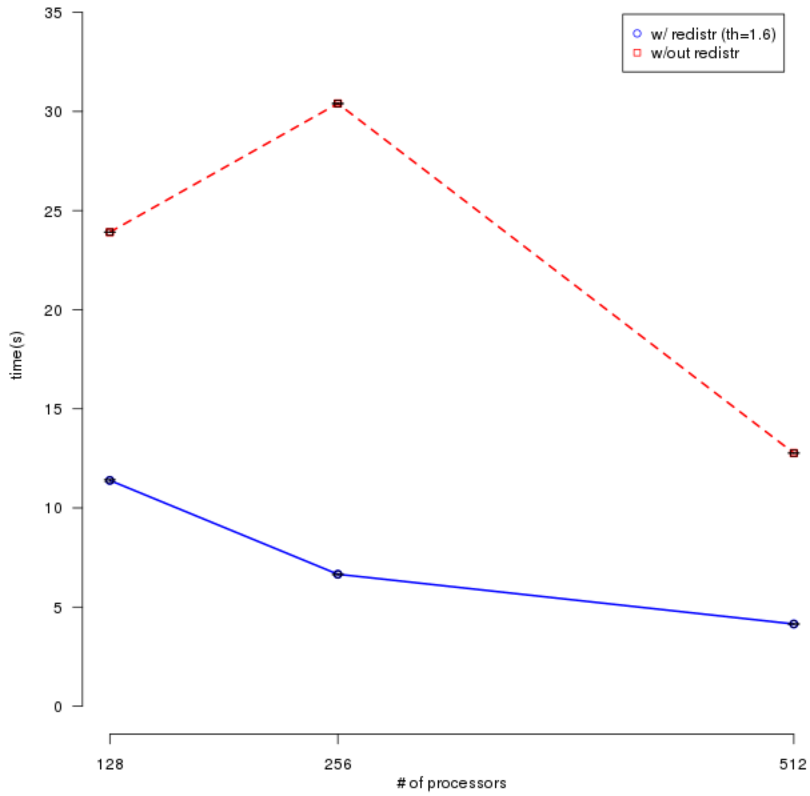


Figure 5.20: Strong scaling of the redistributed (th=1.6) and imbalanced version in Model 2 from 128 to 512 processors on Rain. Confidence Intervals 95%.

Figure 5.22 show that we manage to get up to 350% improvement over the original version on 256 processors, and a 200% improvement on 512 processors.

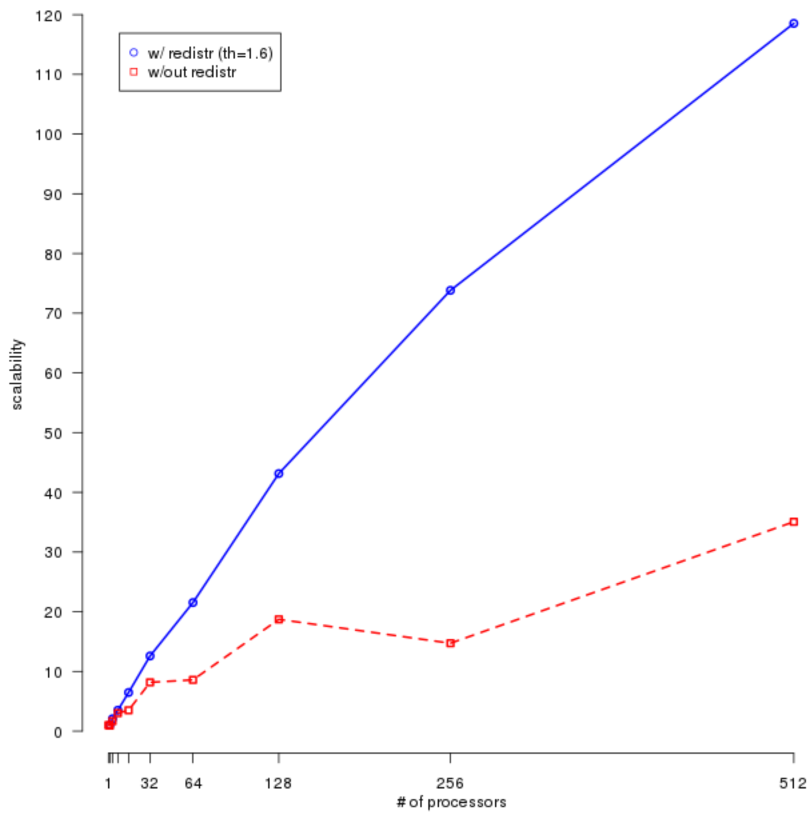


Figure 5.21: Scalability of the redistributed (th=1.6) and imbalanced version in Model 2 from 1 to 512 processors on Rain.

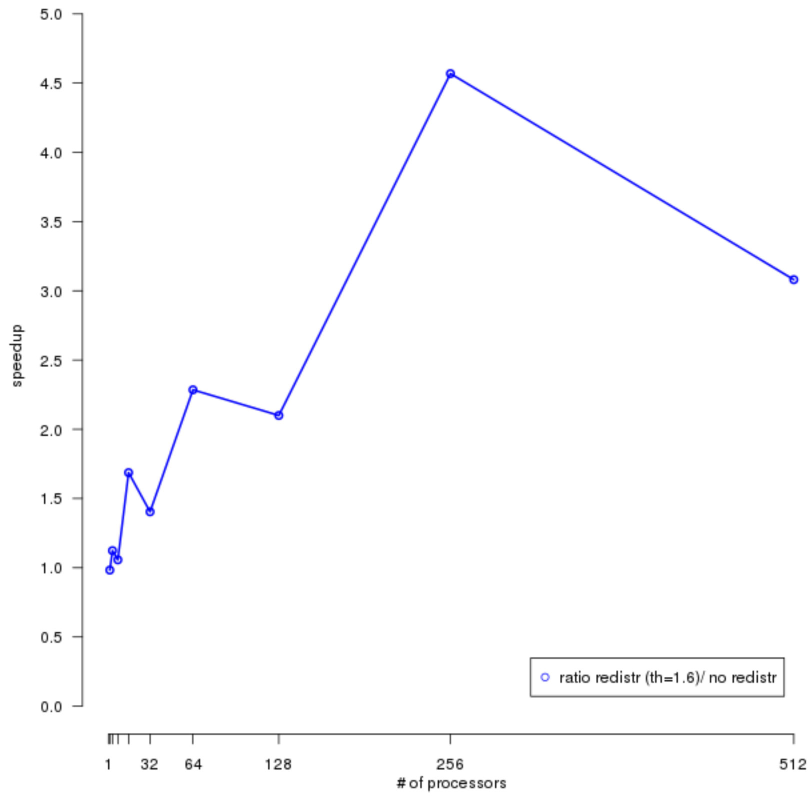


Figure 5.22: Speedup of redistributed version (th=1.6) versus imbalanced version in Model 2 from 1 to 512 processors on Rain.

5.3.3.3 Model 3 - Cylindrical inclusions

The last Model we present, is one of the simplest models, as we can see in Figure 5.3, this model only has 3 regions. Figures 5.23 and 5.24 show how small the improvements are. However, the redistributed version is still faster as we increase the processor count.

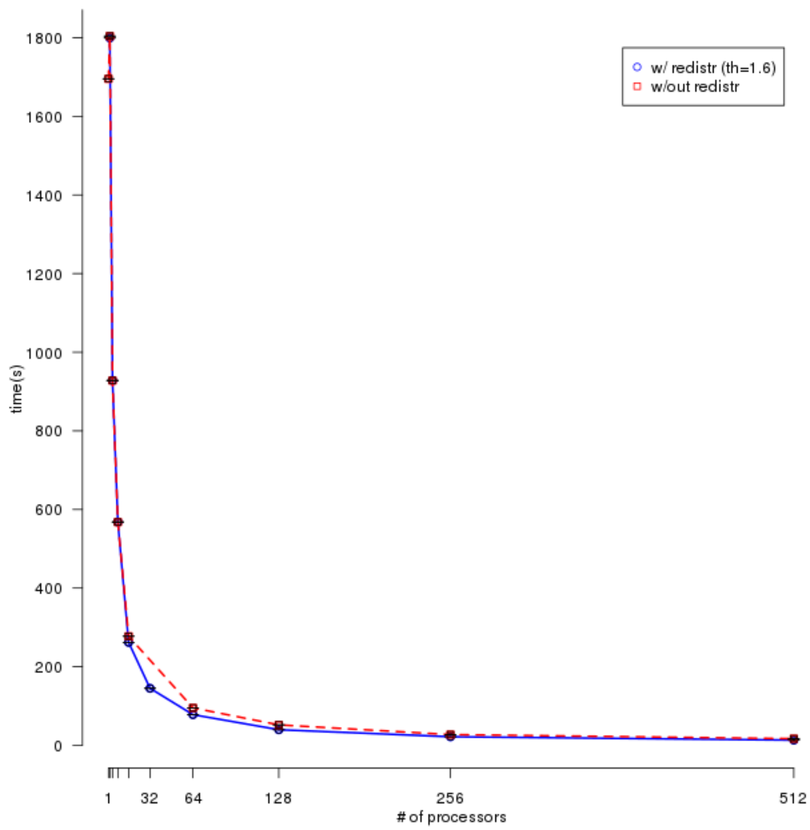


Figure 5.23: Strong scaling of the redistributed (th=1.6) and imbalanced version in Model 3 from 1 to 512 processors on Rain. Confidence Intervals 95%.

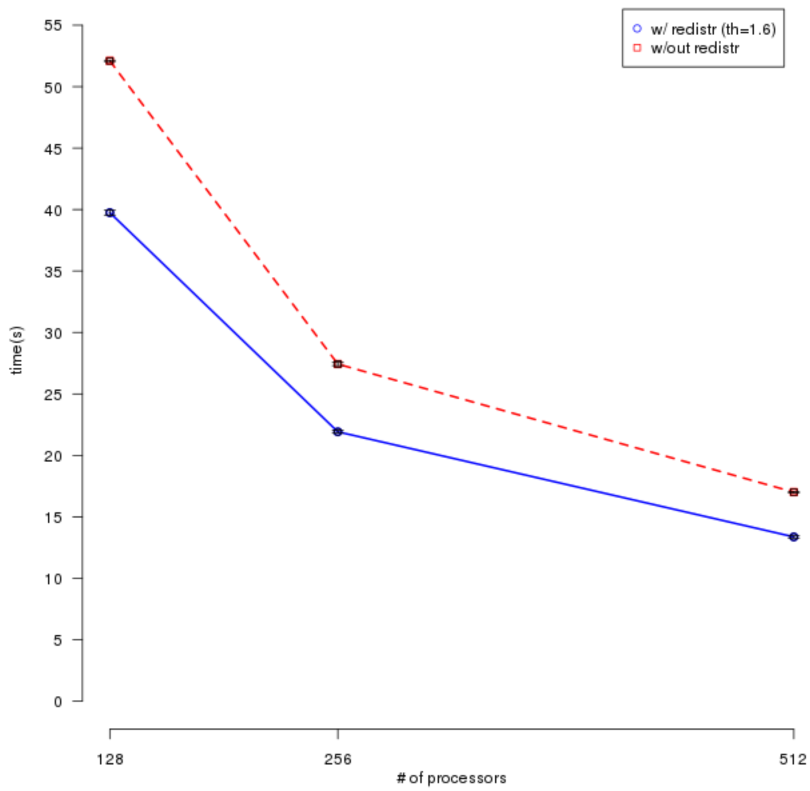


Figure 5.24: Strong scaling of the redistributed (th=1.6) and imbalanced version in Model 3 from 128 to 512 processors on Rain. Confidence Intervals 95%.

Figure 5.25 shows that the non redistributed version scales more consistently. Since the model is simpler, not as much interpolation of the raytubes is required. However, we can notice a better scalability of the redistributed version

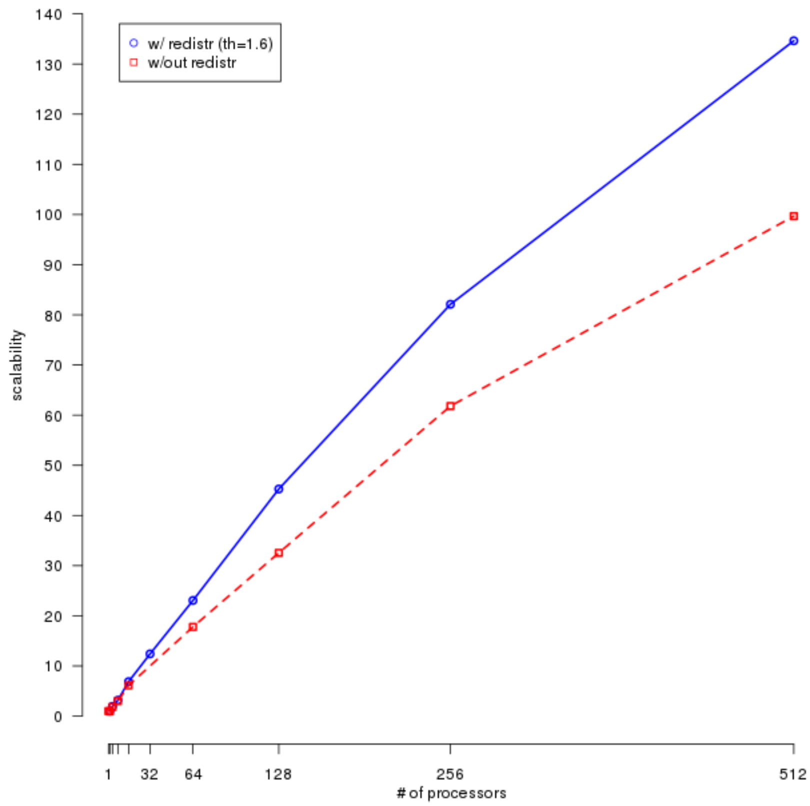


Figure 5.25: Scalability of the redistributed (th=1.6) and imbalanced version in Model 3 from 1 to 512 processors on Rain.

In Figure 5.26, we see that he speedup of the redistributed version over the non redistributed version are more modest than the realistic models. We can get a speedup up to 120% on 16 processors, but after 64 processors we see a speedup of approximatively 30%.

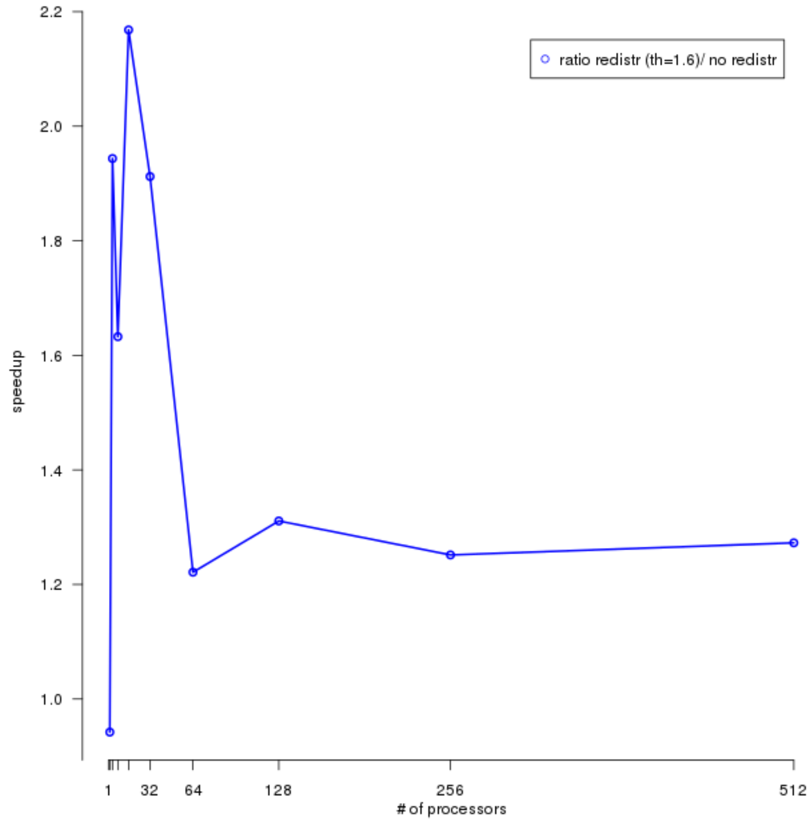


Figure 5.26: Speedup of redistributed version (th=1.6) versus imbalanced version in Model 3 from 1 to 512 processors on Rain.

6. CONCLUSION AND FUTURE WORK

In this thesis we have described our work to improve STAPL's existing view-based distribution and redistribution framework to be more generic. We also described our work on the SRT application to support utilizing STAPL's latest version and use a Recursive Coordinate Bisection framework on the two main data structures in order to generate a decomposition that is used to redistribute these containers. We describe a method to reduce the overhead of redistribution in SRT and only perform it when necessary, and provide performance results.

The redistribution performed here is done using global information, requiring all locations to communicate to share that information. In the future, we could use a decentralized approach, such as *Neighborhood load balancing* [18], where locations share load information only with its immediate neighbors. This decentralized method should be careful about keeping a low count of remote reads for the raytubes if we do not want performance to suffer. The current global redistribution could also be improved by reducing the amount of global communication required. The goal would be to eliminate the expensive all-to-all global communication step, especially when locations want to redistribute only with their closest neighbors, as this kind of communication pattern might not scale well as we increase the number of processors.

Finally, the original distribution requires redistribution because of its too naive partitioning but recently Alyabes [1] presented a method to improve on that initial partitioning, improving execution time up to 35%. It could be interesting to see if this method combined with the method presented in this thesis could improve our parallel seismic ray tracing application.

REFERENCES

[1] A. F. Alyabes. Static Load Balancing using Non-Uniform Mesh Partitioning based on Ray Density Prediction for the Parallel Wavefront Construction Method. Master's thesis, Department of Geology and Geophysics, Texas A&M University, 2014.

[2] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimizations for distributed-memory x10 programs. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1101–1113, May 2011.

[3] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *Computers, IEEE Transactions on*, C-36(5):570–580, May 1987.

[4] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[5] A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pView. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Houston, TX, USA, September 2010.

[6] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: Standard template adaptive parallel library. In *Proc. Annual Haifa Experimen-*

tal Systems Conference (SYSTOR), pages 1–10, New York, NY, USA, 2010. ACM.

[7] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *The Ninth Int. Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 26, pages 52–60, Los Alamitos, CA, USA, 2004.

[8] N. Castet. A Parallel Graph Partitioner for STAPL. Master's thesis, Department of Computer Science, Texas A&M University, May 2013.

[9] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1 –11, March 2007.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[11] N. Ettrich and D. Gajewski. Wave front construction in smooth media for prestack depth migration. *Pure and Applied Geophysics*, 148:481–502, 1996.

[12] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[13] D Hilbert. ber die stetige abbildung einer linie auf flchenstck. In *Mathematische Annalen*, volume 38, pages 459–460, 1891.

[14] Intel. *Reference Manual for Intel Threading Building Blocks, version 1.13*, 2009.

[15] T. K. Jain. Parallel Seismic Ray Tracing. Master's thesis, Department of Computer Science, Texas A&M University, May 2013.

[16] R. L. Gibson Jr. Ray tracing by wavefront construction in 3-D, anisotropic media. *Eos Transactions, American Geophysical Union*, 80:F696, 1999.

[17] R. L. Gibson Jr., V. D. Durussel, and K. J. Lee. Modeling and velocity analysis with a wavefront construction algorithm for anisotropic media. *Geophysics*, 70:T63–T74, 2005.

[18] L. V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. *ACM SIGPLAN Notices*, 28:91–108, 1993.

[19] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[20] G. Lambaré, P. S. Lucio, and A. Hanyga. Two-dimensional multivalued traveltime and amplitude maps by uniform sampling of a ray field. *Geophysical Journal International*, 125:584–598, 1996.

[21] K. J. Lee. *Efficient ray tracing algorithms based on wavefront construction and model based interpolation method*. PhD thesis, Texas A.& M. University, 2005.

[22] A. R. Levander. Fourth-order finite-difference p-sv seismograms. *Geophysics*, 53:1425–1436, 1988.

[23] P. S. Lucio, G. Lambaré, and A. Hanyga. 3D multidimensional travel time and amplitude maps. *Pure and Applied Geophysics*, 148:449–479, 1996.

[24] R. Nishtala, G. Almasi, and C. Cascaval. Performance without pain = productivity: Data layout and collective communication in upc. In *Proceedings*

of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, pages 99–110, New York, NY, USA, 2008. ACM.

[25] I. Papadopoulos, N. Thomas, A. Fidel, N. M. Amato, and L. Rauchwerger. In *Proceedings of the 29th International Conference on Supercomputing(ICS)*, page to appear, Newport Beach, California, USA, 2015.

[26] A. Sanz, R. Asenjo, J. Lopez, R. Larrosa, An. Navarro, V. Litvinov, S. Choi, and B. L. Chamberlain. Global data re-allocation via communication aggregation in chapel. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pages 235–242, Washington, DC, USA, 2012. IEEE Computer Society.

[27] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 235–246, San Antonio, Texas, USA, 2011.

[28] V. Červený. *Seismic Ray Theory*. Cambridge University Press, 2001.

[29] V. Vinje, E. Iversen, and H. Gjøystdal. Traveltime and amplitude estimation using wavefront construction. *Geophysics*, 58:1157–1166, 1993.

[30] V. Vinje, K. Åstebøl, E. Iversen, and H. Gjøystdal. 3-d ray modeling by wavefront construction in open models. *Geophysics*, 64:1912–1919, 1999.

[31] J. Virieux. P-sv wave propagation in heterogeneous media: Velocity-stress finite-difference method. *Geophysics*, 51:889–901, 1986.

[32] H. Gjøystdal, E. Iversen, I. Lecomte, V. Vinje, and K. Åstebøl. Review of ray theory applications in modeling and imaging of seismic data. *Studia Geophysica*

et Geodaetica, 46:113–164, 2002.