Technische Universität Hamburg-Harburg

Bachelor Thesis

A Data Partitioning Algorithm for Sound Particle Radiosity

Jan Winkelmann

August 6, 2012

advised by:
Alexander Pohl
HafenCity University Hamburg

and supervised by:
Prof. Dr. Sibylle Schupp

Technische Universität Hamburg-Harburg
Institute for Software Systems
Schwarzenbergstraße 95
21073 Hamburg

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 6. August 2012

Jan Winkelmann

Contents

1. Introduction **1**

2. Basics **3**
2.1. Geometrical Acoustics Simulation Methods . . . 3
2.1.1. Radiosity Method . . . 3
2.1.2. Sound Particle Method . . . 4
2.1.3. Sound Simulations . . . 4
2.2. Sound Particle Radiosity . . . 5
2.2.1. Sound Particle Reunification . . . 6
2.2.2. Sound Particle Radiosity Algorithm . . . 6
2.2.3. Example . . . 7
2.3. Parallel Computing . . . 8
2.3.1. Parallel Computing . . . 8
2.3.2. The Message Passing Interface . . . 11
2.4. Graph Partitioning . . . 11
2.4.1. Definition . . . 11
2.4.2. Data Partitioning using Graph Partitioning . . . 11
2.4.3. Software: ParMETIS . . . 12

3. A Data Parallel SPR Algorithm **13**
3.1. Parallelization Objectives . . . 13
3.2. Analysis of Reunification Buffer . . . 14
3.3. Decomposition of Reunification Buffer . . . 15
3.4. A Data Parallel SPR Algorithm . . . 17
3.4.1. Synchronization . . . 19
3.4.2. Requirement for Data Partitioning . . . 20
3.5. Memory Requirements . . . 20

4. Data Partitioning Algorithm for SPR **22**
4.1. Graph Model of Data Dependencies . . . 22
4.2. Using Graph Partitioning . . . 23
4.2.1. Existence of Good Decompositions . . . 23
4.2.2. Load Balancing . . . 25
4.3. Data Partitioning Algorithm . . . 25
4.3.1. Parallel Generation of the Directed Graph Model . . . 26
4.3.2. Parallel Generation of the Undirected Graph Model . . . 26
4.3.3. Description of the Algorithm . . . 27

4.3.4. Discussion 29
4.4. Memory Requirements 30

5. Evaluation **31**
5.1. Hypotheses and Variables 31
5.2. Experiment Design 32
5.3. Analysis of Experiment Results 33
5.4. Threats to Validity 39
5.4.1. Threats to Internal Validity 39
5.4.2. Threats to External Validity 41
5.5. Inferences 41
5.5.1. Parallelization Objectives 41
5.5.2. Effectiveness of the Data Partitioning Algorithm 43

6. Future Work **44**

7. Conclusions **45**

8. Bibliography **46**

A. Experiment Setup Data **47**

B. Complete Run Results **50**

1. Introduction

For the simulation of sound propagation in both room acoustics and urban settings, geometrical acoustics simulation methods are widely used. Due to their functional principle, these methods do not simulate wave effects, such as diffraction. Echograms obtained from the simulations can be converted to noise maps of urban settings. Reverberation time, also obtained from echograms, is an important measure in indoor acoustics.

Sound Particle Radiosity (SPR) is such a geometrical acoustic simulation technique, based on the sound particle method, that explicitly models diffraction. The introduction of diffraction requires a split-up of sound particles on impact with a patch, causing an exponential growth of the particles that need to be simulated. SPR compensates this growth by the reunification of sound particles using a two dimensional reunification buffer. The size of the reunification buffer is dependent on the size of the simulated scene and the desired accuracy. For large or highly accurate simulations the reunification buffer does not fit into the main memory of a single computer.

This thesis presents a version of SPR that uses distributed memory parallelism to make faster and more accurate simulations possible. The algorithm leverages data parallelism to distribute the reunification buffer among multiple processes.

As the parts of the buffer are interdependent, inter-process communication is required, which can grow to become a significant source of overhead. To reduce this source of overhead, the thesis presents a parallel data partitioning algorithm. The algorithm uses the SPR input data to decompose the reunification buffer in a way that reduces the required inter-process communication

This reduction is accomplished by constructing a graph of the data dependencies between the rows in the buffer and using graph partitioning to decompose the buffer. When used as a preprocessing step, the data partitioning algorithm reduces the data dependencies between the processes running the parallel version of SPR. The data partitioning algorithm increases the scalability of the parallel SPR algorithm, making even faster and more accurate simulations possible.

Finally, the performance of the parallel SPR algorithm and the data partitioning algorithm for the reunification buffer are tested on cluster computers. The performance of the data decomposition obtained with data partitioning algorithm is compared with a naïve data decomposition.

This thesis is structured as follows. Chapter 2 introduces basic concepts that establish the basis for the rest of the thesis. In Chapter 3, the parallel SPR algorithm is developed. The chapter also describes the goals for the parallel version of SPR and then presents a data parallel parallelization approach. Chapter 4 presents the data partitioning algorithm, used for decomposing the reunification buffer in a way that

reduces the inter-dependencies between the processes. These algorithms are evaluated in Chapter 5, using experiment data obtained by running parallel SPR simulations on cluster computers. Chapter 6 contains future directions for research, and Chapter 7 draws conclusions for the thesis.

2. Basics

This chapter contains the theoretical underpinnings for the rest of the thesis. In the first section the geometrical acoustics, a category of sound simulation methods, is introduced. Further, the second section explains Sound Particle Radiosity, the algorithm that is inspected in the thesis. The third section contains a very short introduction into parallel computing, including the Message Passing Interface. Lastly, the fourth section defines graph partitioning and introduces a library that partitions graphs in parallel.

2.1. Geometrical Acoustics Simulation Methods

This section includes an overview of two geometrical acoustics methods, the radiosity method and the sound particle method, because they form the basis of Sound Particle Radiosity. Furthermore, an explanation of the simulated scenes is given.

Geometrical acoustics is a simulation paradigm for methods that model sound propagation. Phase information of sound is not modeled by geometrical acoustics methods. Hence, simulations do not display wave phenomena, such as scattering or diffraction, unless a method models these effects explicitly. Geometrical Acoustics methods are primarily employed for the simulation of large, closed rooms and urban settings. Simulation methods that employ this paradigm are: the image source method, the sound particle method, the beam tracing method, the radiosity method, and others [10].

Other simulation methods such as wave-based simulation techniques are not inspected in this thesis.

2.1.1. Radiosity Method

The radiosity method models the exchange of sound energy between surfaces. This method was originally used in optics simulation. High accuracy is obtained by discretization into small sub-surfaces, called *patches*. For diffuse sound, the sound energy that each patch radiates to every other patch is calculated. Then, these "view factors" are used as coefficients in a system of linear equations that yield the brightness for each patch.

Unfortunately, the radiosity method can only simulate diffuse reflections, because the exchange of radiated energy is independent of the angle of incidence [14].

2.1.2. Sound Particle Method

The sound particle method approximates sound as particles that transport sound energy along some direction. Particles travel from a starting point along a direction, and are reflected from surfaces with the same emergent angle as the particles' angle of incidence, caused by the law of reflection, known from optics [14].

In the basic form of the sound particle method each particle that impacts a surface reflects back exactly one particle. One possible way to model diffraction and scattering using the sound particle method is by reflecting multiple particles for each particle that impacts a surface [12]. This recursive split-up of particles causes the number of existing particles to grow exponentially with the order of reflection. If high orders of reflection are simulated, the number of particles created is too large to simulate with the sound particle method.

The simulation scene is divided into convex sub-scenes, called *rooms*. Transparent wall transport sound energy from one room to another and split-up of particles from these walls can be used to simulate diffraction. Based on the uncertainty principle, particles that pass closer to an obstacle are more strongly deflected. Since convex rooms are free of obstacles, convex subdivision is an efficient method for the detection of simulation events.

2.1.3. Sound Simulations

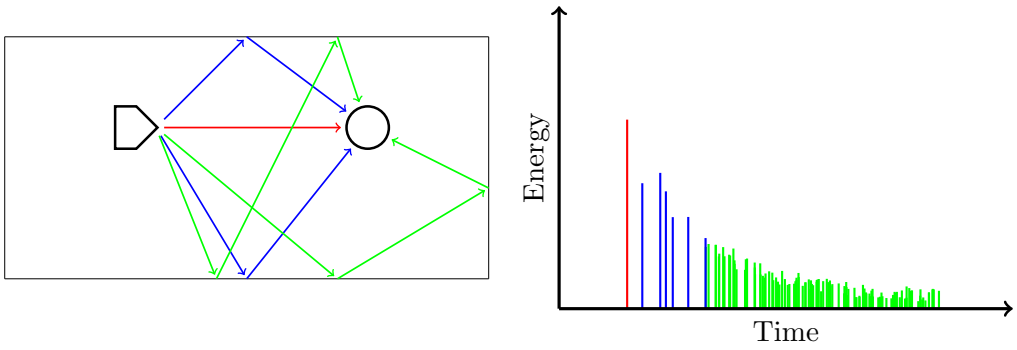
This section contains a description of the settings that are simulated by geometrical acoustics algorithms and the result of a sound propagation simulation, the echogram. For simplicity all simulation settings will be in 2 dimensions. However, all methods presented here also work for 3 dimensions.

In the simulated scenes, reflecting objects (usually walls) are represented by closed traverses. Line segments may be of different materials, which display different reflection properties. An impacting sound particle looses energy proportional to the *absorption factor* of the impacted material.

The remaining energy can be emitted in a number of ways, including geometrical reflection, scattering. Geometric reflections, as known from the law of reflection, are dependent on the angle of the impacting particle. Scattering is a kind of reflection that is independent of the angle of incidence. The ratio of energy emitted by geometrical reflection or scattering is dependent on the impacted material.

Patches on "transparent" walls do not model geometrical reflections or scattering; these walls enable the simulation of diffraction by a custom split-up method. Diffraction-based emission of sound energy occurs in non-convex rooms, only.

Sound is initially introduced uniformly around the *emitter* at a point into the scene. The sound particles propagate through the scene and are registered by, possibly multiple, *receivers*. Receivers are spatially extended, because the particles travel on infinitesimally thin lines. Sound particles are not altered in direction or energy level by passing through the receivers. For the purposes of this thesis the energy is recorded independent of the rays' direction. The information that is recorded by the receivers



(a) A simulation scene in which sound propagates from a sender to a single receiver.

(b) An example echogram, showing sound energy at the receiver over time.

Figure 2.1.: Figure taken from [12]

during the simulation is expressed in an echogram, which describes the reverberation behavior, the "fingerprint" of a scene.

Figure 2.1a shows an example simulation setup. Simulated is a rectangular room with a sound emitter and one receiver. The sound energy propagates from the emitter to the receiver directly, as well as indirectly by reflection from wall segments.

An echogram is a histogram, showing received sound energy over time. Figure 2.1b displays such an echogram. The echogram displays the registered energy over time. Direct sound is registered first, with the most energy, shown in red. Early reflections are shown in blue. The reverberation tail, shown in green, is caused by many particles with lower energies and high orders of reflection.

2.2. Sound Particle Radiosity

This section deals with the simulation method revised in this thesis, and is layed out as follows. After an explanation of sound particle reunification, the Sound Particle Radiosity (SPR) algorithm is explained. Then the basic data structure of the SPR algorithm is examined. Lastly, an example is given to illustrate the principle of SPR and its data structures.

As previously discussed, one way to simulate scattering and diffraction is the recursive split-up of particles on impact. Although the resulting growth of particles is exponential, many of the resulting particles are similar. That is to say their origin points, directions, and start times have little difference. *Sound Particle Radiosity* (SPR) [12] is an extension of the sound particle method that takes the idea of implicit reunification from the radiosity method and aims to handle the large number of particles by explicit reunification. A related method that also models wave phenomena using a variation of the radiosity method is Acoustic Radiance Transfer [13].

2.2.1. Sound Particle Reunification

To systematically and efficiently reunify similar particles, SPR discretized time and space. Surfaces are sectioned into patches, similar to the discretization required for the radiosity method. Time is sectioned into discrete time indices, linearly to the discretization of the line segments.

The discretization causes errors in the simulation, because sound travels in continuous time and space. The point of impact for sound particles are discretized to be the middle of patches. Discretization into smaller elements yields more accurate results at the cost of higher memory and runtime requirements.

The *reunification buffer* is the primary data structure of SPR. A sound particles' energy is stored in this buffer. The buffer has two dimension, one dimensions encodes the origin patch and the ending patch, the other the starting time of the stored particle. Thus, two particles with identical start and destination patches and the same starting time, having the same coordinates, can be reunified by adding their energies.

SPR simulations have various parameters. Discretization is adjustable by the number of patches per mean free path length [9]. The generation of particles is dependent on the number of initial particles emitted and multiple parameters defining the split-up of particles on impact on surfaces. Additionally, particles below a certain amount of energy are discarded.

2.2.2. Sound Particle Radiosity Algorithm

The SPR simulation algorithm works as follows. First, the scene is preprocessed by subdivision into convex sub-scenes and by discretization of all wall segments.

Once the preprocessing is done, the initial sound energy is produced by the emitter. The initial number of particles (a simulation parameter) are distributed deterministically around the emitter. For each particles' destination patch, the starting patch, and starting time is calculated given the position of the emitter. Then the resulting particles are written into the reunification buffer at the appropriate position.

SPR iterates over the buffer from lowest to highest time index. Each particle in the buffer at the current time index is processed as follows. Dependent on the patch material, the split-up of the particle at the destination patch is calculated, and reflected geometrically or via scattering. On reflection the particle is slit-up into a fixed number of particles, the reflection behavior of the patches is time invariant. The destination patch of the original particle is the origin of the reflected particles.

Strictly iterating over one time index after the other ensures that all reunification has been done on a certain particle, before that particle is processed. For a given time index all patch to patch combinations are inspected. Non zero entries in the buffer represent particles.

Each particle is processed only once, after the entry in the buffer was evaluated the particle is not inspected again. If reunification occurs a set of coordinates may be written to multiple times, by adding the energies of the particles. The energy of each reflected particle is checked, particles with an energy below a certain minimum are

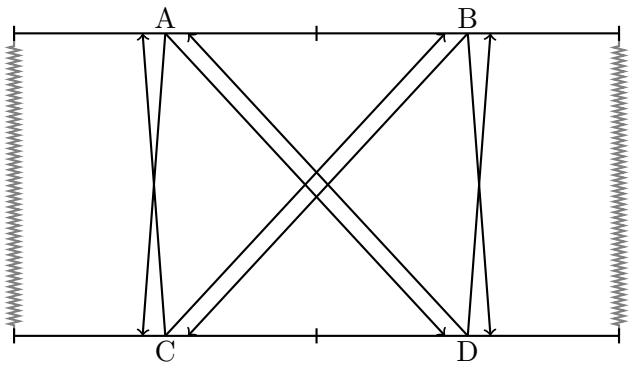


Figure 2.2.: An example scene for a simulation, consisting of one room with 2 full absorber walls and 2 walls having 2 patches each.

discarded. If the particle passes through a receiver while traveling from origin patch to destination patch, the energy is recorded by the receiver. Once all the reflected patches have been written to the buffer the original particle has been completely processed. Then, the next particle in the buffer can be simulated.

Particles with a start time after the maximum simulation time are discarded. The algorithm terminates if no particles remain in the buffer.

A basic optimization is to make the reunification buffer a ring buffer with respect to the time index. The size of the time index is the travel time between the furthest two patches. Then old time indices, containing information that is not useful any longer, can be overwritten, decreasing the size of the reunification buffer significantly.

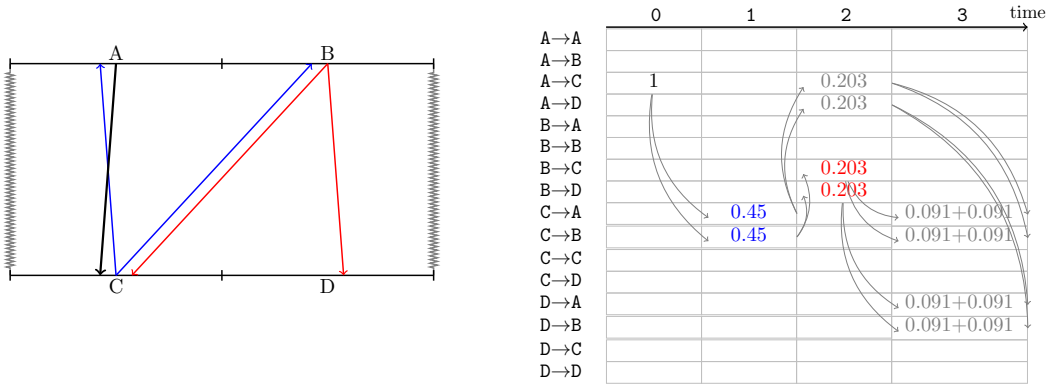
2.2.3. Example

Figure 2.2 is the scene that will be used as an example throughout the thesis. The scene contains one room with four walls. Two of the walls, with one patch each, are fully sound-absorbent. Fully absorbing walls, by definition, do not reflect any particles.

The other two walls are discretized into two patches each, labeled from A to D. These patches have an absorption factor of 0.1. Each patch reflects two new particles for an impacting particle.

Particles can travel from each patch to the other two patches on the opposite wall. The fully absorbing walls do not produce rays, as all sound energy is absorbed by the patches. For simplicity, particles to the full absorbing walls are not shown. The particles always originate from the middle of a patch.

Figure 2.3 is a basic example of how the reunification matrix works. Figure 2.3a is an instance of the example room. The initial particle, indicated in black, travels from patch A to patch C. On impact, the particle is split-up into two to new particles, one from C to A and another from C to B, indicated in blue. Not all particles for the second order of reflection are shown. One of the particles reflected at patch A would be a particle from A to C , which is already in the figure, hence the reflection from



(a) The example room with an initial ray from patch A to C. Some reflections are indicated.

(b) The reunification matrix for the example room. Rays that are not displayed in Figure 2.3a are grayed out.

Figure 2.3.: An example scene with the corresponding reunification buffer.

patch A are omitted. The second order reflection originating from patch B are shown in red.

Figure 2.3b is the corresponding reunification buffer for the example room. For simplicity, all particles require the same amount of time, regardless of the emergent angle; this is meant to simplify the example, the simulations model reflection accurately. The buffer is initialized with the particle from A to C with an energy of 1.0 and a time index zero. With each reflection, the original particle looses one tenth of its energy to absorption and spawns two new particles that equally divide the remaining energy among them.

In this example, the first particle spawns two new particles, one particle from C to A and one particle from C to B, each of which has $1 * 0.9/2 = 0.45$ energy units. The two new particles are written into the time index one. Recursive split-up of particles continues until time index three at which time a reunification occurs. Usually the particles at time index three should have $0.203 * 0.9/2 = 0.091$ energy units, but since two rays are reunified at that time their energies are added.

2.3. Parallel Computing

This section will give a short introduction into the basics of parallel computing and aims to provide the basics concepts required to understand parallel algorithms.

2.3.1. Parallel Computing

Parallel computing can be used to execute programs on computers with large amounts of resources, called clusters. Efficient execution of SPR on clusters is the goal of this thesis. To this end, the theory for using clusters to speed up SPR simulations is

provided.

The first two parts lay out formal machine and programming models, respectively. Further, the third part discusses problem decomposition for distributed execution. Lastly, the concepts of overhead and speedup are introduced.

Machine model

The machine model assumed in this thesis is the Multicomputer [2]. A Multicomputer consists of multiple, separate von Neumann machines, called *processes*. These processes have separate memory space and processing units, while collaborating via inter-process communication using an interconnect.

The inter-process communication cost between any two processes is assumed to be time invariant. Additionally, the cost of sending and receiving of messages is assumed to be the same. The cost of a communication is proportional to the message length, with a small constant factor that accounts for latency. Moreover, local memory access is assumed to be significantly cheaper than access to remote data via message passing.

With these assumptions, sending a large amount of data in one chunk requires the same time, independent of the sender and the receiver. However, sending the same amount of data in multiple chucks requires more time than sending all the data at once, because of the latency delays.

With these assumptions the model does not capture all aspects of the real world. Realistically estimating communication costs requires a more specific model, incorporating prior knowledge about the computer architecture, the interconnect setup, communication latency and transfer times, etc.

Developing a more complex model and optimizing algorithms for it is beyond the scope of this thesis. The machine model presented here is a good approximation and algorithms developed with it should perform as expected.

Programming model

The programming model employed in this thesis is known as Single Program Multiple Data (SPMD), an extension of Flynn's original taxonomy [2]. In SPMD, the available number of processes is fixed for each program execution, and each process runs the same binary. Processes are identified by a number, called *rank*, used to identify the processes. The rank enumerates the processes, starting by zero. Inter-process communication, over the interconnect, is called *message passing* and used for synchronization and for copying data from one address space to others.

The machine and programming models employed in this thesis are to be distinguished from shared memory models. In these models multiple threads share some or all of the memory but usually have separate processing units.

Problem Decomposition

Using a distributed memory machine model, such as the Multicomputer, requires algorithms to divide up problems among processes in order to solve them. There are

different degrees of decompositions, which vary in their granularity.

Domain decomposition is the highest level decomposition, it uses the application semantics for decomposing the problem. Task level parallelism divides a problem in multiple sub-problems that are then distributed to the processes. Sub-problems are generally not identical, and each process may receive more than one such sub-problem.

A lower level method for problem decomposition is data decomposition, also known as data level parallelism This approach divides up the data of the problem among the processes. As far as possible, each process only works on local data. The efficiency of data parallelism depends on how well local data can be worked on without requiring message passing.

A problem parallelizes well if it can be solved solely by working on data locally—these problems are in the class of embarrassingly parallel problems. If, however, a problem cannot be solved by inspecting one datum at a time, the problem's data is said to have *data dependencies*. These data dependencies hinder parallel execution, as they usually require message passing. For some problems, mappings of data to processes may minimize the data dependencies between processes. Thus, even not embarrassingly parallel problems can be parallelized efficiently.

Data partitioning is the process of decomposing data for later use in parallel programming. A data partitioning algorithm analyzes the problem or a specific instance of a problem and produces a data decomposition. In this thesis, data partitioning aims to reduce data dependencies between the processes, which reduces the required message passing. As message passing requires runtime, decreases the message passing amount also reduces runtime.

Overhead

Very few real world problem parallelize without data dependencies that require communication between the processes. The extra effort that is required to solve a problem in parallel is the *overhead* of the parallel algorithm.

Conceptually, the sequential execution of a program requires a certain number of instructions I. An optimal parallel algorithm distributes the workload evenly between P processes without requiring extra effort; each process then executes $\frac{I}{P}$ instructions. Thus, $\frac{1}{P}$ is the upper bound for parallelism. This upper bound is rarely reached in practice, because breaking down the problem and distributing the parts between the processes and ensuring that all processes get roughly the same amount of work often requires a considerable amount of instructions by itself.

To quantify the quality of parallelism that algorithms display, several metrics have been developed. Let T_P be the runtime of a program running with P processes in parallel. The *speedup* is defined as $S_P = \frac{T_1}{T_P}$.

While high speedup indicates that a parallel program runs faster than its serial version, speedup does not give any information about how efficient the program is. *Efficiency* sets the obtained speedup in relation to the $\frac{1}{P}$ upper bound. It is defined as $E_N = \frac{S_P}{P}$.

2.3.2. The Message Passing Interface

The Message Passing Interface (MPI) is a specification for a message passing interface library. It is the de-facto standard for cluster computing. The MPI standard, released by the MPI Forum, that will be used in this thesis is the 2.0 Standard [11].

Execution environments that implement the MPI standard provide functionality for basic message passing, thus providing a framework for robust programming in the message passing paradigm. Although it is technically possible to use different programs that communicate with each other, each process uses the exact same one. The number of processes to be spawned is specified at runtime.

MPI aims to provide ease of use and portability for programs on cluster computing platforms. Portability is an essential part in cluster computing, as hardware can vary widely. For instance the nodes in the clusters may vary in the interconnect medium (Ethernet or Infiniband), endianness, and word size (32 or 64 bit). Compliant execution environments provide message passing services between processes.

MPI messages can be synchronization between processes or the movement of data from one address space to the other. The standard guarantees complete and correct delivery of messages or notification of failure.

2.4. Graph Partitioning

This last section of the current chapter introduces graph partitioning. First, the term is formally defined. Then the usage of graph partitioning as a data decomposition method is discussed. Finally, ParMETIS a library for parallel graph partitioning is presented.

2.4.1. Definition

A $k-$way partitioning of an undirected graph (V, E) is a partitioning of the set of vertices V into k sets, $V_1, V_2, ..., V_k$ such that

- $|V_i| = n/k$, for $|V| = n, i \in \{1, .., k\}$
- $\bigcup_{i=1}^{k} V_i = V$
- $V_i \cap V_j = \emptyset$, for $1 \leq i, j \leq k, i \neq j$.

The *edge cut* of a partition is the number of edges incident to vertices of different subsets. $k-$way graph partitioning with minimal edge cut is NP-hard, but as number of heuristics with good runtime have been developed [6].

2.4.2. Data Partitioning using Graph Partitioning

Decomposing problems with data parallelism often results in dependencies between the processes. When modeled as a graph, partitioning can be employed to reduce these data dependencies.

Reducing the edge cut of the partition reduces the data dependencies between the processes and thus the communication between the processes. A reduction of the communication also decreases runtime, since than accessing data locally is faster than communication over the interconnect.

Communication is directed, as one process sends data and another receives it. Graph partitioning, however, is performed on undirected graphs, because the direction of the communication is irrelevant to the communication volume. Assuming that incoming and outgoing communication have the same cost, it is sufficient to model the communication arising from data dependencies with an undirected graph. Thus, graph partitioning is suitable for data partitioning in data parallel algorithms. The graph that requires partitioning may be too large to be held in the memory of a single machine. To partition even large graphs parallel graph partitioning libraries have been developed.

2.4.3. Software: ParMETIS

A number of tool kits have been developed for parallel graph partitioning, including JOSTLE [15], ParMETIS [7], and PT-Scotch [1].

ParMETIS and PT-Scotch are still under active development, but JOSTLE is not. PT-Scotch is slower but yields better results than ParMETIS. For both libraries the edge cut worsens as a function of the processes used, but for PT-Scotch less so than for ParMETIS [1]. In this thesis ParMETIS will be used for graph partitioning, because runtime of the graph partitioning is important.

ParMETIS employs a multi-level method for graph partitioning. Multi-level methods "coarsen" the target graph, by collapsing vertices, thus obtaining a series of smaller graphs. The smallest graph is then partitioned using traditional methods. Lastly, the partitioning of the coarsest graph is successively "refined", by mapping the partitioning onto the bigger graphs, finally yielding a partition for the original graph.

ParMETIS is a parallel library, all processes call the graph partitioning routine at the same time, and each process passes a part of the graph to the routine. The graphs vertices to be encoded as integers, numbered contiguously starting from zero. Each process owns a number of graphs vertices, and its adjacency list. Since the graph is undirected, each edge is encoded on the adjacency list of both incident vertices. Each process passes ParMETIS the adjacency lists of the vertices it owns.

The result of the graph partitioning is a mapping of vertices to subsets. ParMETIS returns this mapping, however, every process receives only the mapping for the vertices it owned when calling the graph partitioning function. The vertices are not automatically redistributed, assembling the entire mapping of vertices to processes or redistribution of the vertices must be done manually. For a complete explanation of the ParMETIS library and its functions see the ParMETIS manual [8].

3. A Data Parallel SPR Algorithm

In this chapter a parallel SPR algorithm is presented that divides the reunification buffer and the workload among multiple processes. Before the algorithm is presented, some additional concepts are introduced in previous sections. First, Section 3.1 defines the objectives for parallelization. These objectives form a basis for the algorithm design, and will later be used to evaluate the quality of the algorithm design. Section 3.2 contains an analysis of the data behavior the SPR simulations exhibit. In Section 3.3 the insights into the data behavior are used then to propose a decomposition of the reunification buffer. Further, Section 3.4 introduces a data parallel algorithm for SPR. Last, Section 3.5 analyzes the parallel SPR algorithm with respect to the parallelization objectives.

3.1. Parallelization Objectives

This section sets the design priorities for the parallelization algorithm, and eventually in Chapter 4, the data partitioning algorithm. These objectives will decide which trade-offs are made in the design of the algorithms, and set the bar for a successful algorithm design.

Parallelization in cluster computing can decrease the runtime of programs. In the case of SPR, however, the runtime of the simulations are not the primary problem. Rather, the limiting factor is the size of the reunification buffer, as it quickly becomes too large to be held in the random access memory (RAM) of a single machine.

The first parallelization objective is, therefore, leveraging the RAM gained by executing a SPR simulation on multiple computers. Hence, the memory overhead of the parallel SPR and the data partitioning algorithm should be kept small. The memory requirements of sequential SPR simulations are in $\mathcal{O}(N^3)$, for the number of patches in the scene N. Memory overhead is not necessarily only a function of the problem size, but also of the number of processes P. An efficient algorithm should decrease the memory overhead with the number of processes. Ideally, the algorithm would scale in $\mathcal{O}(N^2/P)$, but a flexible data decomposition of the reunification buffer, for example a mapping of rows to processes, has a size in $\mathcal{O}(N^2)$. Unfortunately, this requires a memory overhead in $\mathcal{O}(N^2)$ to be acceptable. The memory usage of the parallel SPR algorithm is discussed in Section 3.5, and of the data partitioning algorithms in Section 4.4.

While the runtimes of the serial version of SPR are relatively short, runtime speedup is still desirable. Under no circumstances should the parallel algorithm attain speedups of less than one for realistic amounts of processes (1–64 processes). To keep runtime

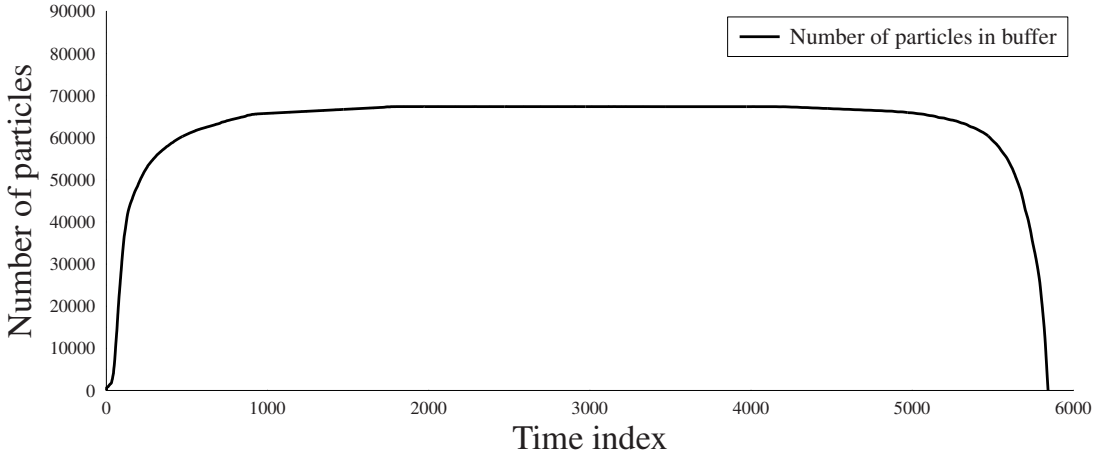


Figure 3.1.: Number of particles in the reunification buffer over time

overhead to a minimum, data partitioning should not require excessive runtime, and remain a small fraction of the execution time, independent of the number of processes.

3.2. Analysis of Reunification Buffer

The reunification buffer is the core data structure of SPR. It stores sound particles by origin patch, destination patch, and starting time index. The SPR algorithm populates the buffer with initial rays, and then iterates over the buffer from lowest to highest time index. Each retrieved particle is simulated and deleted from the buffer. The simulation of a particle results in a number of new particles, which are written back into the reunification buffer. These writes may reunify two particles or create a new particle.

An interesting metric is the number of particles in the buffer at a given time. The length of a particles' path from one patch to the other is limited by the scene, effectively limiting the time indices that can be written to when simulating particles at a given time index. The number of patches in the scene is also finite. Therefore, the number of particles that can be in the buffer at one time is also limited. The rest of this section will analyze exactly how close the buffer is to full occupation during a simulation.

At the beginning of the simulation only very few particles are stored in the buffer. With each simulation step, the number of particles increases. Given a typical set of simulation parameters, the number of particles grows, until either the maximum number of particles or the maximum simulation time is reached. For very large absorptions of the scene the number of particles decreases before the maximum simulation time is reached.

Given a buffer with a high number of particles stored in it, the reunification rate of particles is very high, and SPR becomes an effective simulation method. For a scene with low absorption, and a high maximum simulation time, the number of particles

in the buffer is plotted in Figure 3.1[1]. The graph clearly displays three phases: The beginning phase, with a low, but rising number of elements. A plateau phase, where the increase in particles decreases until the maximum of possible particles in the buffer is reached. Finally, the end phase when the maximum simulation time is reached, and particles are discarded, resulting in a decreasing number of particles. In the figure the beginning phase is from time index 0 up to circa time index 1000. The end phase starts at about time index 5000.

In the rest of the thesis the plateau phase will be called the *dense* case, as the buffer is densely populated. The beginning phase and the end phase will be called the *sparse* case. The data partitioning will be done for the dense case, as this is the case SPR works best at, and it is where the most writes per time index take place.

The occupation of the reunification buffer does not always progress as shown in Figure 3.1. Usually the maximum simulation time is shorter than the 10 seconds used in that example. The maximum simulation time may well be reached before the buffer occupation plateaus.

Even if the maximum number of particles in the buffer is reached, the buffer is not completely occupied. A maximum for the number of particles always exists, but is dependent on the simulation parameters. The maximum will always be reached given low absorption and high simulation time. If, for example, one of the scenes' walls is fully absorbent, no particles will originate from that wall, which decreases the possible number of particles in the buffer. In case all walls are fully absorbent, no particles will be reflected, and the buffer will never fill.

The occupation of the buffer is also influenced by the scene layout. It is physically impossible for a particle to have the same origin and destination wall. Therefore, many rows of the buffer cannot contain any particles, because the simulation cannot write to them.

3.3. Decomposition of Reunification Buffer

This section deals with the decomposition of the reunification buffer. Decomposing the buffer is an important step towards leveraging the RAM of processes for simulation.

This section disregards the simulation of the particles and focuses on the decomposition of the buffer and how particles are stored and retrieved. Decomposition of the buffer is done in a way that allows the storing and receiving of particles to be transparent from the SPR simulation routines. The entire buffer, as it is presented to the SPR simulation routines, is called the *global buffer*. The parts of the buffer a process owns form what is called the *local buffer*.

The decomposition of the buffer has two requirements in order for reads and writes to the global buffer to be possible. First of all, every process needs to be able to determine the owner of a given particle. This can be done via a mapping of particles to processes $\mathcal{P}$. Second, given a particle that is owned by a process, that process needs to be able to store that particle in the local buffer, and be able to retrieve it.

[1]For the simulation setup see Section 5.2

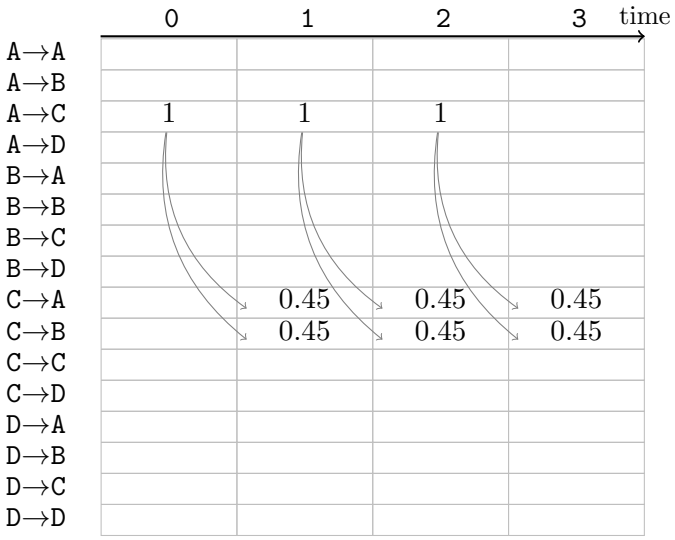


Figure 3.2.: The reflection behavior of patches is time invariant.

Read and write operation to the local buffer require a mapping $\mathcal{C}$ of a processes own particles in the global buffer to particles in the local buffer. This mapping is partial for process numbers larger than two, as each process will only own parts of the global buffer. Each process has its own version of this mapping, because each process owns different parts of the global reunification buffer.

If one were to map each particle to a process explicitly, the mapping of particles to processes alone would be of the same size as the reunification buffer. Then, the memory overhead would be larger that the reunification buffer itself. The mapping needs to be smaller, sacrificing granularity of the mapping for size. A number of ways to generalize the mapping are thinkable, for example mapping based on columns or rows of the reunification buffer.

A data decomposition based on columns, for example, would be a very inefficient mapping. Only particles from a few columns can be simulated at each time, because the data dependencies of each particle must be respected. Many processes could not work on their part of the buffer. Also, every single write would be remote, because the particles generated by the simulation step always have a higher time index.

The split-up of particles on impact with patches is time invariant. Hence, the simulation of a particle yields the same particles, expecting only the time offset, for the entire simulation, see Figure 3.2. Therefore, the loss of granularity for row-based mappings is very small, but the reduction in size is significant. This thesis will continue to use row-based mappings.

For a data decomposition to be complete and correct, a $\mathcal{P}$ and a $\mathcal{C}$ mapping for every process is required. $\mathcal{P}$ must be total, that is, it must assign every row to a process. Every $\mathcal{C}$ mapping must map all the rows that are assigned to that process via the $\mathcal{P}$ mapping to rows in the local buffer.

$\mathcal{P}$

Row	Process ID
A→A	0
A→B	0
A→C	0
A→D	0
B→A	0
B→B	0
B→C	0
B→D	0
C→A	1
C→B	1
C→C	1
C→D	1
D→A	1
D→B	1
D→C	1
D→D	1

$\mathcal{C}$

Process 0		Process 1	
Global	Local	Global	Local
A→A	1	C→A	1
A→B	2	C→B	2
A→C	3	C→C	3
A→D	4	C→D	4
B→A	5	D→A	5
B→B	6	D→B	6
B→C	7	D→C	7
B→D	8	D→D	8

Figure 3.3.: Mapping of rows to processes (left) and mappings from local to global buffer coordinates (right). This forms a data decomposition for reunification buffer of the example room for two processes.

A simple way to obtain a data decomposition is to evenly distribute the rows among the processes with no regard to data dependencies, whatsoever. This will be called a naïve data decomposition. Naïve mappings preserve correctness of the algorithm, but since the data dependencies of the particles in the buffer are not taken into account the mappings are likely inefficient.

Figure 3.3 is an example for a naïve data decomposition of the example room and two processes. This mapping takes the reunification buffer from Figure 3.2 and splits it between B→D and C→A. Any decomposition for two processes assigns each process half the rows of the global buffer, the number of columns remains the same for the local and the global buffer. The global buffer has 16 rows, from A→A to D→D, while the two local buffers have 8 rows each, numbered from 1 to 8. The $\mathcal{P}$ mapping of rows to process identifier assigns the first eight rows to process zero and the latter eight to process one. While $\mathcal{C}$, the mappings from local to global buffer coordinates, assigns the rows on the global buffer to coordinates in the local buffer.

3.4. A Data Parallel SPR Algorithm

The largest memory structure of SPR is by far the reunification buffer, and SPR spends most of the runtime simulating particles from the buffer. Thus, parallelization of SPR using a data parallel approach seems appropriate.

Given the parallelization objectives a distributed memory approach must be taken. Shared memory parallelism, such as Open MP, can only increase the amount of processing power available, but not the amount of available RAM. Only distributed memory parallelism results in a gain of available RAM.

The development of a data parallel approach for SPR requires an understanding of how the algorithm operates on its data. The data dependencies between the elements in the reunification buffer is key for data parallelism.

To this end, Algorithm 3.1 illustrates the SPR algorithm from a data-centric point of view. The algorithm provides a sketch of how particles are read and written to the reunification buffer. The sound propagation logic takes place in the called functions.

Algorithm 3.1 The SPR from a data-centric point of view

```
function SPR(SimulationParameters sp)
    allocateBuffer( sp )
    List<Echogram> es = initEchograms( sp )
    while simulationNotCompleted() do
        Particle p = getNextParticleFromBuffer()
        List<Particle> ps = simulateParticle( p, es, sp )
        for all Particle q in ps do
            storeInBuffer( q )
    end while
    return es
```

The algorithm operates as follows. First, the buffer and the echograms are initialized. Then, in the algorithms' main loop, the next sound particle is read from the buffer and simulated. The simulation returns the split-up particles and those are written back to the buffer at the appropriate coordinates. `simulationNotCompleted()` checks whether the simulation is complete, by checking whether the global buffer is empty, or if the maximum simulation time is reached.

`getNextParticle()` reads particles from the reunification buffer. To ensure correct results, the particles are retrieved ordered by time index. All the particles returned by the simulation of the particle have a higher time index as the original particle. The ordering can be realized by iterating over the time indices in a strictly increasing sequence.

SPR can achieve parallelization by dividing the reunification buffer among the processes. Each process calls the simulation routine for the particles in its part of the buffer. Write operations to non-local parts of the buffer are realized via message passing. Hence, this approach divides up the workload along with the data.

A data parallel concept for SPR is given in Algorithm 3.2. The buffer is naïvely distributed among the processes, as discussed earlier. Every particle is uniquely assigned to one process. Each process reads particles from its own part of the buffer and performs the required simulation steps, which generates new particles. The new particles are sent to the respective processes and written to the buffer there. Finally, the

Algorithm 3.2 A sketch of the parallelization approach for SPR

```
function PARELLEL_SPR(SimulationParameters sp)
    List<Echograms> es = initLocalEchograms( sp )
    allocateLocalBuffer( sp )
    while simulationNotCompleted() do
        Particle p = getNextParticleFromLocalBuffer()
        List<Particle> ps = simulateParticle( p, es, sp )
        for all Particle q in ps do
            sendParticleToOwningProcess( q )
        List<Particle> ps' = receiveParticlesFromProcesses()
        for all Particle q in ps' do
            storeInLocalBuffer( q )
    end while
    synchronizeEchograms( es )
    return es
```

echograms are synchronized between the processes, as each process keeps echograms for its own particles.

Algorithm 3.2 is a straight forward implementation of the data parallel paradigm. The reunification buffer is distributed among the processes and each process simulates local particles. Remote write operations are handled transparently for the simulation routines.

Given a correct decomposition, the parallel SPR algorithm yields the same result as the sequential one. The parallel version is correct, because it simulates the same particles as the serial version. In the parallel version the buffer is distributed, but a correct decomposition can hide the message passing from the simulation routines. The parallel version also simulates multiple particles at the same time, which preserves correctness, as long as the processes synchronize.

The next section discusses this synchronization, and the section after that discusses why data partitioning is helpful for this algorithm.

3.4.1. Synchronization

The data parallel approach requires that the data dependencies are accounted for when simulating sound particles. SPR assumes that only completely reunified particles are evaluated, i.e., no particle needs to be evaluated twice. A process may, therefore, not simulate a particle from its buffer, that a process might still write to. For the parallel algorithm, this constraint necessitates synchronization between the processes, independent of the communication of the sound particles.

Forbidding the simulation of particles that are not fully reunified induces a time index interval that particles might be read from for simulation. The size of the interval is the minimum travel distance between any two patches located on different processes. Reading particles from this interval ensures that no particle is read from the buffer

and simulated while another process might still write to that coordinate in the buffer. Due to technical limitations, see Section 4.4, the time index particles are read from is the same for all processes; all processes synchronize after simulating all particles stored in one time index.

Throughout the rest of the thesis it is assumed that the data dependencies of all particles are properly observed using synchronization.

3.4.2. Requirement for Data Partitioning

When a particle is simulated it is split-up into S particles, which are written back into the buffer. S is the split-up of a particle on simulation, it is a simulation parameter of the SPR simulation. Reunification of sound particles is achieved by repeated adding of the particles energy to the appropriate coordinates in the buffer. If particles are reunified, they are written to multiple times, but always read from only once, at the time of simulation.

In the worst case one read to the buffer causes S writes to it. The only instances where less than S are written is, if the energy of a resulting particle is below the specified minimum, or the starting time of the particle above the maximum simulation time. Both cases are rare, therefore, most often a read to the buffer causes S writes.

The data parallel algorithm reads the particles for simulation from the local buffer and writes the generated particles back to the buffer. Reads to the buffer are guaranteed to be local, but writes are remote, if the resulting particle is owned by another process. For naïve decompositions the probability that a write is remote increases with the number of processes, since the buffer is evenly distributed among the processes.

For P processes, each process simulates up to N^2 particles per time index, which cause as many reads to the buffer. Every simulated particle is split into S particles, so $N^2 \cdot S$ particles have to be written back into the buffer. If a naïve decomposition is used, only $\frac{N^2}{P} \cdot S$ of these writes are local, the rest require message passing. For large P there are S remote accesses to the buffer for every local access, which is a major source of overhead.

One way to mitigate the problem, is to make ensure that many writes to the buffer are local. By definition naïve decompositions do not consider the behavior of the simulation, and are therefore unlikely to reduce the number of remote writes. A decomposition is required that reduces the remote writes, by analyzing the data dependencies of the rows in the buffer. Chapter 4 introduces a data partitioning algorithm that produces such data decompositions.

3.5. Memory Requirements

Keeping the memory overhead to a minimum is an important part of the parallelization objectives. Therefore, this section examines the memory requirements of the parallel SPR algorithm and evaluates the resulting overhead.

There are three objects that take up significant memory in the parallel SPR algorithm: the reunification buffer, the mappings required for the decomposition of the

buffer, and the communication buffers. This section will examine the memory requirements of all three in turn. Let N be the number of patches in the scene and P the number of processes.

The global reunification buffer has a size in $\mathcal{O}(N^3)$. Evenly dividing the rows of the global buffer among the processes yields size of the local buffer in $\mathcal{O}(N^3/P)$.

The mapping $\mathcal{C}$, from local to global buffer coordinates, is of size N^2/P as each process owns that many rows. This mapping is different for each processor, and each processor requires only its own mapping. However, constant time look-up in both directions requires a separate data structure for each direction. The look-up from local to global buffers has a size of N^2/P, but the look-up from global to local has a size of N^2.

The mapping of rows to processors is of size N^2. Each processor requires this mapping, and constant time look-up is only required in the direction of rows to processors.

The algorithm does not specify how remote writes to the buffer are to be implemented. Sending a single large communication is generally more efficient than sending multiple smaller ones. When sending large communications, however, the sizes of the sending and receiving buffers can become quite large.

It would be possible to send each particle that requires a remote write directly and as a single particle without buffering. For this option, the memory requirements are practically none, however, a very inefficient one, as there will be a latency delay for every particle.

The other extreme would be to simulate multiple time indices before synchronization, buffering remote particles all the while. As part of the synchronization all particles would be sent to the owning processes and written to the local buffers. Unfortunately, this approach would require very large buffers, and complex synchronization between the processes to ensure the data dependencies are respected.

A good trade-off between communication time and memory overhead is to simulate all particles in one time index. Particles that require remote writes are sent to their owner processes after all the particles in a time index have been simulated. One time index holds N^2/P particles per process in the worst case. Given a split-up of S, the maximum number of particles that can be written while simulating one time index is $S \cdot N^2/P$. Thus, the send buffers are quite large, but still in $\mathcal{O}(N^2)$.

To summarize, the proposed parallel SPR algorithm divides the reunification buffer, and with that the workload, evenly among the processes. The memory overhead of each process for the mappings is in $\mathcal{O}(N^2+N^2+N^2/P) = \mathcal{O}(N^2)$. When buffering particles from one time index before exchanging particles, the size of the communication buffers is in $\mathcal{O}(N^2)$, also for each process. Therefore, the memory overhead of each process is in $\mathcal{O}(N^2)$, which fulfills the parallelization requirements. The total memory overhead is the memory overhead of all processes combined, and thus in $\mathcal{O}(N^2 \cdot P)$.

4. Data Partitioning Algorithm for SPR

Chapter 3 introduced a parallel SPR algorithm that decomposes the buffer by rows. The message passing requirements of the algorithm depend strongly on the buffer decomposition used. In this chapter the data dependencies between the rows are examined, and formalized in a graph. The resulting graph is then partitioned to obtain a data decomposition that reduces message passing requirements.

4.1. Graph Model of Data Dependencies

With the work of the previous chapter, the term data dependency can be refined for the purpose of this thesis: The data dependencies of a datum are the data that must be processed before that datum can be safely evaluated. In the context of SPR and the reunification buffer, a particle must be fully reunified before it can safely be evaluated. Thus, the data dependencies of a particle are all the particles that generate that particular particle on simulation.

One way to approximate the data dependencies of a particle is to assume it is dependent on all particles that could still write to that particle, see Section 3.4.1. While this is a correct approximation, it is a very broad one. A better approximation is desirable, because the model of the data dependencies will be used for the data partitioning, and a more accurate model will result in a better decomposition.

The rows of the reunification buffer encode the origin and destination patch of a ray, while the column encodes the starting time of a ray. Section 3.3 introduced a row-based decomposition of the buffer for distribution among the processes. A model of the data dependencies at a row level is appropriate, because the decomposition is also row-based. Let R be the set of rows in the reunification buffer. Data dependencies between rows can be expressed by a "on simulation writes to" relation $\mathcal{R} \subseteq R \times R$. $(p, q) \in \mathcal{R}$ if the simulation of a particle in the row p results in write of a particle into the row q. It is possible to generate the information for this model by inspecting only one time index without loss of information, because the reflection properties of patches are time invariant.

Using the rows as vertices and $\mathcal{R}$ as the set of edges, the data dependencies between rows can be expressed as a graph $(R, \mathcal{R})$. Throughout the thesis this graph will be called *graph model*.

Figure 4.1 is the graph model for the example room in Figure 2.2. The graph model is generated from the reunification buffer, which has all patch to patch combinations as rows. Patch combinations which do not occur in the example room are present as vertices but do not have incoming or outgoing edges. Given a particle in the row `A→C`,

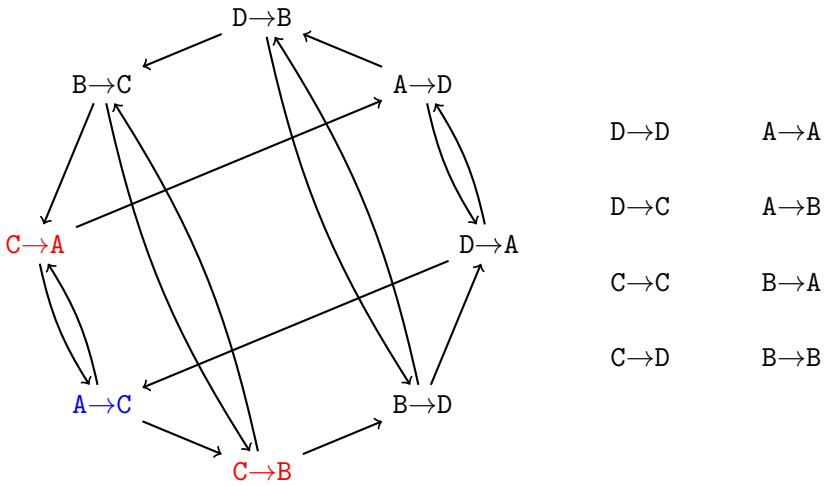


Figure 4.1.: Graph model of the data dependencies of the rows in the example room

indicated in blue, the graph model predicts that the simulation of that particle will cause writes to the rows C→A and C→B, indicated in red.

4.2. Using Graph Partitioning

This section takes a look at the graph model developed in the previous section and argues why graph partitioning of that model produces an efficient data decomposition.

As a preparation step, the graph model can be used to measure the quality of existing decompositions. Assume a row mapped to a processor has an outgoing edge in the graph model that is incident to a row that is mapped to another process. Then the simulation of a sound particle from that row requires massage passing to write the resulting particle to the buffer at owner process. Application of the graph model to a data decomposition of the buffer yields all possible remote writes.

An efficient decomposition is one that has few edges that are incident to vertices mapped to different processes, because fewer remote writes can occur. This is exactly the definition of an edge cut. Thus, partitioning the graph model should yield a mapping of rows to processes that reduces message passing requirements.

Such a mapping does not influence the workings of the algorithm. All writes to the buffer are still done, but the buffer is distributed in such a way that as many of the expected writes as possible are local.

4.2.1. Existence of Good Decompositions

Conceptually, for a dense reunification buffer, sound particles go from every part of the room to every other part. It is, therefore, not obvious that data decompositions exist that reduce the data dependencies between processes. Indeed, if the decomposition would be on a patch level all decompositions would be equally bad, as statistically

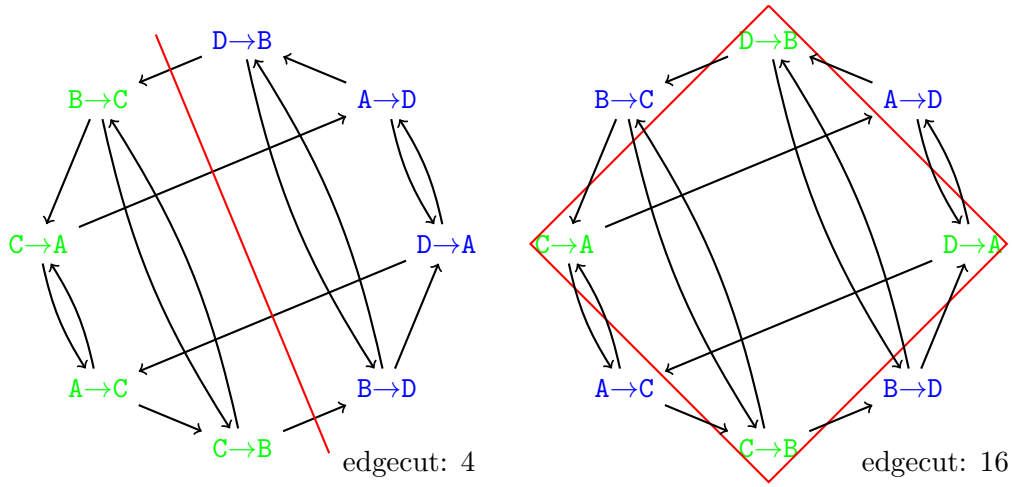


Figure 4.2.: Two different edge cuts for the graph model of the example room

every patch communicates with every other. This section establishes, that the graph model can be used to evaluate the quality of a decomposition.

For a partitioning on a row basis, however, there are good decompositions, because the data dependencies between the rows are time invariant and fairly few in number. Indeed, as split-up is a constant factor S, the number of edges is $N^2 \cdot S$. The linear growth indicates a sparse graph.

For complete graphs, every partition has the same edge cut. For sparse graphs, partitions with different edge cuts exist. Sparseness of a graph does not guarantee good decompositions, as the edge cut might only be slightly better, or the heuristics might not actually find the best solution.

Figure 4.2 illustrates the existence of good and bad decompositions for the graph model of the example scene. For the sake of simplicity the figure does not show the vertices that have no incident edges. A red line indicates the edge cut that partitions the set of vertices into two subsets, with the vertices marked in green and blue. Shown on the left is a partitioning with an edge cut of four, which is the minimum edge cut for four partitions. The minimal edge cut for two partitions is zero and can be obtained by partitioning the vertices without incident vertices in one partition and the vertices shown in the figure in the other. The partitioning shown on the right, with an edge cut of 16, is an application of the graph model to the naïve decomposition from Figure 3.3.

Good row-based decompositions of the reunification buffer exist. Decompositions that yield a low edge cut when the graph model is applied to it are good partitions. Moreover, the graph model can be used to obtain good partitions by applying graph partitioning to the graph model.

4.2.2. Load Balancing

The communication behavior of a parallel program may change as the simulation progresses [4]. If these changes are significant and happen often, the algorithm may need to change the data decomposition on-the-fly. This is known as dynamic load balancing. Performing data partitioning on a problem once to obtain a data decomposition, and maintaining that decomposition throughout the program execution, is known as static load balancing.

SPR undergoes such a change in communication behavior, because the reunification buffer changes its degree of occupation. The reunification buffer starts out sparse, changes to dense and, in the end phase, back to sparse. As the model focuses on the dense case the obtained mapping is likely not to be optimal for the entire simulation.

Once the buffer is dense, it will remain dense until the end phase of the simulation. In the entire dense phase the communication behavior will not change. For every time index, every possible particle will be simulated, generating the maximum number of particles, all of which need to be written to the buffer. Hence, a mapping that optimizes for the dense case does not need to change mid-simulation. The entire data decomposition can be computed in a preprocessing step and, from then on, does not need to change.

It is possible to calculate efficient mappings for early steps of the simulation, if the starting rays are taken into account. Then, the propagation of the starting rays through the rows can be modeled; such an approach will likely yield a mapping that assigns all rows active in the early steps of the simulation to one process.

As discussed, obtaining a mapping for sparse reunification buffers is more complex than for dense reunification buffers. Switching partitioning schemes is even more complicated, because the optimal time to switch from one mapping to the other is hard to determine. Therefore, this thesis concentrates on static load balancing.

4.3. Data Partitioning Algorithm

k-way graph partitioning of the graph model returns k partitions that minimize the edge cut. If the k partitions are used to assign the rows to processes, the resulting decomposition reduces the data dependencies between the processes. The idea of the data partitioning algorithm is, therefore, to generate the graph model and to partition it to obtain a data decomposition.

The data partitioning algorithm must also be parallel, in order to scale well with the number of processes. Section 2.4 introduced ParMETIS, a parallel graph partitioning library. The generation of graph model in parallel is treated in the next subsection. As previously mentioned, graph partitioning is generally done on undirected graph. The conversion of the directed graph model to an undirected one in parallel is discussed in the second subsection. This section concludes with a full presentation of the algorithm, that uses graph partitioning of the undirected model to obtain a data decomposition.

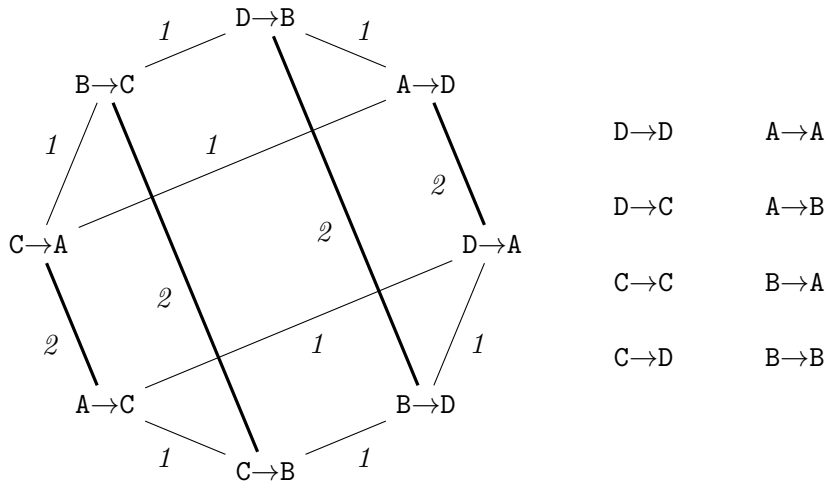


Figure 4.3.: Undirected graph model of example room, with weighted edges

4.3.1. Parallel Generation of the Directed Graph Model

For sequential SPR the directed model can be generated by simulating a particle in every row of the buffer and observing the rows that are written to. Because the algorithm uses the simulation routines to observe the data dependencies between rows, which ensures the data dependencies are recorded correct. The simulation of a single particle in a row suffices, because the reflection behavior is time invariant.

The distributed algorithm for generating the directed model is data parallel, just as the parallel SPR algorithm. First, the buffer is distributed using a naïve decomposition. Then, each process generates the graph model of its own rows, using the sequential algorithm just described. No synchronization is required, because the particles do not need to be written to the buffer, only the row coordinated need to be observed. Thus, each process generates the outgoing edges for its own rows. Since every row is owned by one of the processes, the entire graph is generated.

4.3.2. Parallel Generation of the Undirected Graph Model

The graph partitioning software takes the undirected graph as an distributed adjacency list. Each edge in the graph must be represented in the adjacency lists for both incident vertices. The undirected graph can be generated from the distributed directed graph in the following manner.

First, the directed model has to be generated in parallel. For each edge (e, f) a process has in its local directed graph model, it adds the flip edge (f, e). Now, each process has a sub-graph of the undirected graph model, which contains all edges that are incident to a vertex it owns.

Then, the processes send edges that are not outgoing from vertices it owns to the owner process. Each process receives its new edges and adds them to the adjacency lists. Now each process has the complete adjacency lists for all vertices it owns.

A special case occurs if the directed graph model contains an edge and its flip edge, so that the graph contains cycles of size 2. This is the case in the example graph model in Figure 4.1 between, for example, the vertices `C`→`A` and `A`→`C`. Both edges are in the directed model already, so adding the flip edge to the model introduces duplicates to the adjacency lists. As the undirected model is not a multigraph, duplicates in the adjacency lists are illegal.

In the directed model an edge indicated that exactly one sound particle may be written to the row that is the head of that edge. If duplicates in the undirected model are deleted, the model predicts only one particle written, when both rows may be written to. Fortunately, ParMETIS allows graph to be weighted. To compensate for the duplicate edge, the remaining edge is assigned the weight 2, all other edges are assigned a weight of one.

In the end, the graph model is undirected, weighted, and each process owns all outgoing edges for its own vertices.

Figure 4.3 shows the undirected graph model for the example room, with weights on the edges. Wherever there was a loss in edges in the conversion from the directed graph, the remaining edge was assigned a weight of 2.

4.3.3. Description of the Algorithm

Now everything is in place for presenting the actual graph partitioning algorithm. Algorithm 4.1 provides the pseudo-code of the algorithm.

First, the reunification buffer is distributed using the naïve decomposition, and all possible fields on the lowest time index are filled. Then, the adjacency lists for the graph model are initialized as a graph, the default value for edge weights is one. Vertices are not weighted.

All particles in the buffer are simulated. For each simulated particle, the resulting particles are saved as edges in the graph. The particles that result from the simulation are not written back to the buffer, as only one time step is to be simulated. Once all particles have been simulated, and the appropriate edges added to the graph, the distributed directed graph is completed. Each process has generated all outgoing outgoing for the vertices it owns.

Next, for each edge in the graph, the flip edge is added, effectively making the local graph undirected. At this point the distributed graph model can contain outgoing edges that do not belong to the process that owns them. Thus, all edges are iterated over, and the "foreign" edges are sent to the appropriate processes and deleted from the local graph model.

Then, the processes receive the edges that were sent to them in the previous steps. The newly arriving edges are added to the graph. Duplicates are then detected and deleted, while the weights of the remaining edges is set to two. The default for edge weights is one.

Now, the distributed undirected graph model and its weights are complete and graph partitioning can begin. The resulting mapping is returned only for the vertices a process owns. To obtain a complete $\mathcal{P}$ mapping, the local mappings have to be

Algorithm 4.1 The Data Partitioning Algorithm

```
function DATAPARTITIONING
    allocateLocalBuffer()
    fillLowestTimeIndexOfLocalBuffer()
    LocalGraph g ← {}
    initWeights( g, 1 )
    for all Particle p in localBuffer do
        List<Particle> ps = simulateParticle( p )
        for all Particle q in ps do
            Row e = getRowCoordiante( p )
            Row f = getRowCoordinate( f )
            addEdge( e, f, g )
    for all Edge (e, f) in g do
        addEdge( f, e, g )
        if isNotLocal( f ) then
            sendEdgeToOwningProcess( f, e )
            removeEdge( f, e, g )
    List<Edge> es = receiveEdgesFromProcesses()
    for all Edge (e, f) in es do
        addEdge( e, f, g )
    List<Edge> es = retrieveDuplicates( g )
    for all Edge (e, f) in es do
        removeEdge( e, f, g )
        setWeight( (e, f), 2, g )
    LocalMapping<Row,Process> lm = partitionGraph( g )
    return Mapping<Row, Process> assembleMapping( lm )
```

concatenated. Generating the $\mathcal{C}$ mappings from a given $\mathcal{P}$ mapping is trivial and not part of the pseudo-code.

The graph partitioning can be done as a preprocessing step, and from the resulting mapping of rows to processes, each process can easily calculate a mapping of the distributed coordinate it owns to local coordinates of the buffer.

Figure 4.4 gives the mappings produced by the data partitioning algorithm for four processes. The mappings returned by the graph partitioning library for each process is indicated on the $\mathcal{P}$ table. For example, the first process own the first four rows, and thus receives the mapping for all the rows of the form $\texttt{A}\rightarrow *$. This is a partitioning shown in Figure 4.2 on the left. The $\mathcal{C}$ mappings are only given for processes 0 and 1. In this example the mappings for the processes 2 and 3 are only unusable rows.

$\mathcal{P}$

Row	Process ID
A→A	2
A→B	3
A→C	0
A→D	1
B→A	2
B→B	3
B→C	0
B→D	1
C→A	0
C→B	0
C→C	2
C→D	3
D→A	1
D→B	1
D→C	2
D→D	3

$\mathcal{C}$

Process 0		Process 1	
Global	Local	Global	Local
A→A	1	A→B	1
A→C	2	A→D	2
B→A	3	B→B	3
B→C	4	B→D	4
C→A	5	C→D	5
C→B	6	D→A	6
C→C	7	D→B	7
D→C	8	D→D	8

Figure 4.4.: Mapping of rows to processes (left) and mappings from local to global buffer coordinates (right) for the example room

4.3.4. Discussion

A data decomposition can only be as good as the model it is based on. The developed graph model makes some assumptions that influence the resulting decomposition.

First, the graph model assumes a dense reunification buffer. The model assumes, and the partitioning optimizes for, every possible communication taking place for each time index. As was previously discussed, this is not true for the entire simulation, but it is a safe over-approximation.

The second potentially problematic assumption is that the graph model predicts actual communication costs correctly. Assuming a dense reunification buffer, the graph model makes accurate predictions about communication amounts, because the number of particles to be sent is modeled, and the size of a particle is constant and known. The cost of communication is, however, not only a function of communication amount, but also factors such as congestion of the inter-connect. As discussed in Section 2.3.1, a model that predicts actual communication costs is out of scope.

4.4. Memory Requirements

Just as in Section 3.5 this chapter discusses the memory overhead of the data partitioning algorithm.

The directed model contains $N^2 \cdot S$ edges, and each process owns its outgoing edges, so the total number of edges is $N^2 \cdot S/P$. The undirected model adds flip edges for all edges in the undirected model. If the newly generated edge is foreign, it is sent to the owner process. In the worst case every process sends all its edges to one process. Hence, the size of the undirected model is $(1 + 1/P) \cdot S \cdot N^2$ in the worst case.

The $\mathcal{P}$ mapping has a size of N^2, because—by definition—the mapping assigns a process to every patch to patch combination Therefore, the memory requirements of the data partitioning algorithm is in $\mathcal{O}(N^2)$ in addition to the memory requirements for the graph partitioning.

5. Evaluation

This chapter conducts an analysis of the parallel SPR and the data partitioning algorithm presented in this theses. The evaluation is based on empirical data gathered by executing an implementation of the algorithm on a cluster with multiple parameter sets. Some hints as to the procedure for empirical analysis in computer science were taken from Jedlitschka, Ciolkowski, and Pfahl [5].

The first section contains an explanation of the hypotheses tested in this evaluation as well as the variables of the test runs. The second section lays out the test setup for the test runs. The third section contains the analysis of the data obtained from the test runs. In the fourth section threats to validity are discussed. Section 5 draws inferences from the data.

5.1. Hypotheses and Variables

There are two questions of interest regarding the algorithms presented in this thesis, which are now formed into two hypotheses.

The first hypothesis that is to be checked in this chapter is, whether the parallelization objectives are met by the implementation. In order for this hypothesis to hold, the program must show modest memory overhead, and scale well in memory and runtime to large number of processes. A large number of processes is 64 in this case. Thus, the program must not allocate excessive memory for communication buffers or mapping tables, and attain speedups up larger than one for multiple processes. Where applicable, preprocessing time must be a small fraction of the runtime.

The second hypothesis is: The presented data partitioning algorithm lowers the amount of message passing and runtime compared to a naïve data decomposition. Message passing requirements can be measures by the total communication amount of the processes in bytes.

The experiment has four independent variables: First, the patches per mean free path length is a simulation parameter that controls the accuracy of the simulation. SPR scales in $\mathcal{O}(N^3)$ in both space and runtime for this variable. All runs have 200 patches per mean free path length.

The second independent variable controls the data decomposition used for the reunification buffer. The experiments compare the naïve data decomposition with the data partitioning algorithm from Chapter 4.

The third independent variable controls whether main memory usage should be measured during execution. These measurements increase the runtime of the program and must, therefore, be made separately. Observing memory usage is an indication for memory overhead.

The fourth independent variable is the number of processes that execute the program in parallel. By fixing all other independent variables and varying this one, the scaling behavior of the implementation can be observed.

The dependent variables are the runtime of the implementation, the total amount of communication between the processes, the total amount of RAM used, the time preprocessing time, and the edge cut of the graph partitioning.

The first variable is the overall runtime of the program. If the processes have different runtimes, the longest runtime is taken. The runtime of the data partitioning algorithm if it has been used, is measured explicitly. The overall runtime includes the runtime of the data partitioning algorithm. Measurement of all time variables is in seconds.

The third dependent variable is the total amount of communication between the processes, which is measured by the MPI library directly. These measurements are given in megabytes. As a third variable, the edge cut of the graph model is given, if the graph partitioning algorithm is used.

The RAM usage of the program is the fourth variable, measurements are only taken if the ram measurement variable is set accordingly, The result of the measurement is not only the peak memory usage, but a detailed profile of allocated heap memory over time.

For multiple executions with the same independent variable set, the runtime and the preprocessing time vary. The main memory usage, communication amount, and edge cut do not vary for multiple executions.

Table 5.1 shows all runs done for evaluation. A line separates the independent variables on the left from the dependent variables on the right.

5.2. Experiment Design

This section contains an description of the experiment design. First the experiment setup, then the test process is described.

Experiment setup

The implementation under test is developed from a reference implementation of the SPR algorithm. Because the implementation is written in C++, the reunification buffer, the communication buffers, and the mapping tables are STL vectors. The implementation supports only scenes with one room at this point, so only convex scenes can be tested. Non-convex scenes are broken down into convex rooms by SPR, so non-convex rooms are not supported at this point.

For all test runs the same scene with one set of parameters are used. ParMETIS, version 4.02, is used as the graph partitioning library.

The implementation is compiled with the Intel compiler 11, run with Intel MPI version 3. Resource allocation and scheduling is done by the clusters' batch system, PBS Pro version 10.

Allocated heap memory usage is measured with Valgrinds' massif tool. In the Appendix A the version numbers and the program flags used are described in more detail.

The simulations were run on the cluster "Apis" at the Hamburg University of Technology [3]. The cluster is an SGI Altix XE with a Gigabit Ethernet interconnect. Cluster nodes used for the runs are equipped with 2 Intel X5560 CPUs (Quad Core, Nehalem (2.8GHz)) and 48GB RAM each. A cluster node may run as many processes as it has core, in this case a cluster node may run up to 8 processes. Running a simulation with eight processes requires one cluster node, while 64 processes require eight cluster nodes.

Test process

Let a run be an execution of the program on a cluster, with one fixed set of independent variables. The results of each test run are recorded. All runs have 200 patches per mean free path length. Let a test batch be a set of test runs with 1,2,4,8,16,32 and 64 processes. To accurately observe the variation on the dependent variables, each test run is repeated five times. The rest of the independent variables remain fixed.

Hypothesis testing, for both hypotheses, requires three test batches. The first batch uses the naïve data partitioning, and does not measure memory usage. The second batch uses the data partitioning algorithm, and also does not measure memory usage. The third and last batch uses the data partitioning algorithm, but this time does measure memory usage.

A test batch for measuring the memory usage of the the naïve partitioning is not required. The used memory mapping buffers are identical; the only difference will be the memory requirements of the data partitioning algorithm. No additional insight over the memory overhead can be gained by measuring the memory usage by the naïve decomposition.

5.3. Analysis of Experiment Results

An overview of the results are given in Table 5.1. A full table of all runs is B.2. The first column indicates the number of processes in the run, the second column the data decomposition used. "Decomp." indicates use of the data partitioning algorithm, while "naïve" indicates use of the naïve data decomposition. The third row, labeled "Data Prt" gives the time required for the data partitioning algorithm in seconds. The next column "Runtime" gives the runtime of the run. "Runtime" is median values of five runs, and "Data Prt." is the data partitioning time from the same run. The fifth column gives the edge cut of the graph model, if graph partitioning was used. The next column, "Comm. Amnt" contains the total communication volume of the run. The last column, "RAM usage", gives of main memory used in the run in gigabytes.

The test results are now analyzed in turn.

# Proc	Decomp.	Data Prt. (S)	Runtime (H:MM:SS)	Edge Cut	Comm. Amnt (MiB)	Ram usage (GiB)
1	naïve	n.a.	0:18:39	n.a.	0	n.a.
2	naïve	n.a.	0:10:44	n.a.	11184	n.a.
4	naïve	n.a.	0:06:21	n.a.	15520	n.a.
8	naïve	n.a.	0:03:30	n.a.	15520	n.a.
16	naïve	n.a.	0:03:06	n.a.	15521	n.a.
32	naïve	n.a.	0:20:32	n.a.	15523	n.a.
64	naïve	n.a.	0:16:40	n.a.	15528	n.a.
1	data part.	35	0:18:49	0	0	n.a.
2	data part.	36	0:11:09	230219	2377	n.a.
4	data part.	28	0:06:37	327014	3557	n.a.
8	data part.	26	0:03:52	395961	4501	n.a.
16	data part.	37	0:02:24	431881	5721	n.a.
32	data part.	56	0:02:30	702507	8026	n.a.
64	data part.	70	0:02:01	906848	11598	n.a.
1	data part.	654	2:42:28	0	0	25.530
2	data part.	538	1:29:18	230219	2377	28.080
4	data part.	481	0:58:37	327014	3557	30.159
8	data part.	509	0:36:51	395961	4501	33.915
16	data part.	567	0:24:07	431881	5721	40.889
32	data part.	658	0:19:14	702507	8026	55.580
64	data part.	762	0:17:43	906848	11598	84.457

Table 5.1.: An overview of the test results, test batches are separated by lines

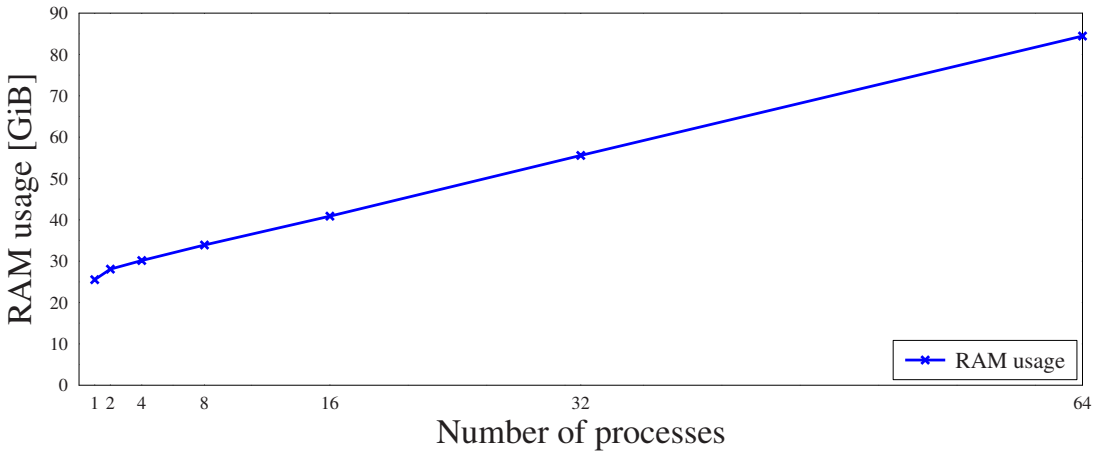


Figure 5.1.: Accumulated maximum RAM usage vs. number of processes

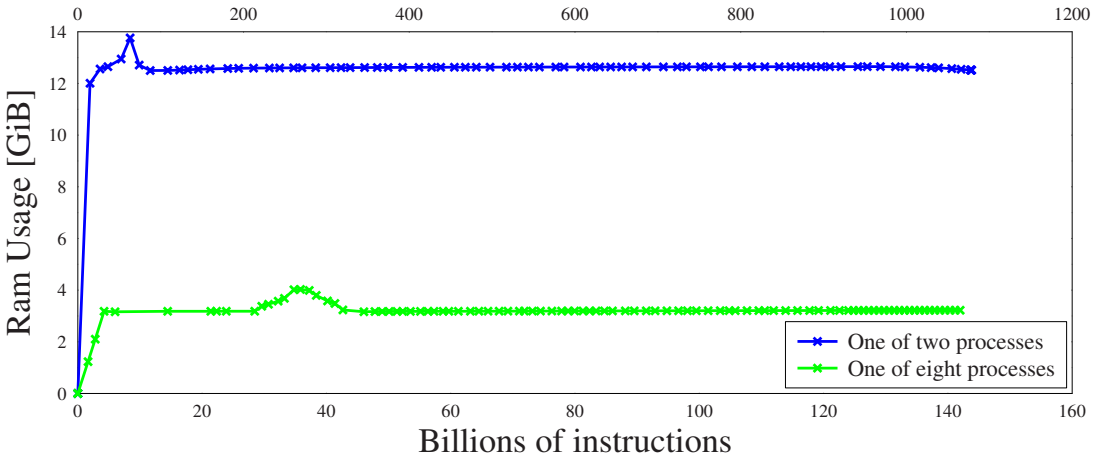


Figure 5.2.: Allocated memory over time of two processes, one from a run with two processes (blue) and the other with a run of eight processes (green).

Analysis of Memory Usage

Figure 5.1 plots RAM usage against the number of processes. Clearly, the required memory is proportional to the number of processes. All the runs indicated in the figure have the same number of patches per mean free path length, so the size of the global reunification buffer is the same for all these processes. A run with one process requires 25.5 gigabytes of memory, while a run with 64 processes requires 84 gigabytes. Thus, for a run with 64 processes has at least 59 gigabytes of memory overhead. However, the amount of memory gained by additional processes is higher than the memory overhead. The run with 64 processes runs on 8 cluster nodes with 384 gigabytes of RAM total.

Figure 5.2 shows the output of the massif profiler for test runs with two and eight processes. The graph shows the allocated memory on the heap against the number of instructions executed for one of the processes in the run. Let p_2 be the process of the run with two processes, and p_8 be the process of the run with eight processes. The upper x axis belongs to the blue graph, representing p_2 The green graph, for p_8, is drawn relative to the lower x axis.

As the reunification buffer, and with it the workload is evenly divided among processes, processes that collaborate with fewer other processors must hold more of the reunification buffer and execute more instructions. This can be seen in the Figure 5.2, the process that collaborates with only one other process, p_2, requires 12 gigabytes of main memory and executes 10 trillion instructions. The green graph indicates, that p_8 only requires 3 GiB of memory on average and 140 billion instructions.

The RAM usage shows the same pattern for both processes. Both processes allocate memory up to a certain point, 3 GiB for the process in the run with eight processes, and 12GiB for the other process. After the initial initialization follows a short peak in RAM usage for both processes. Then, the allocated memory falls to the previous

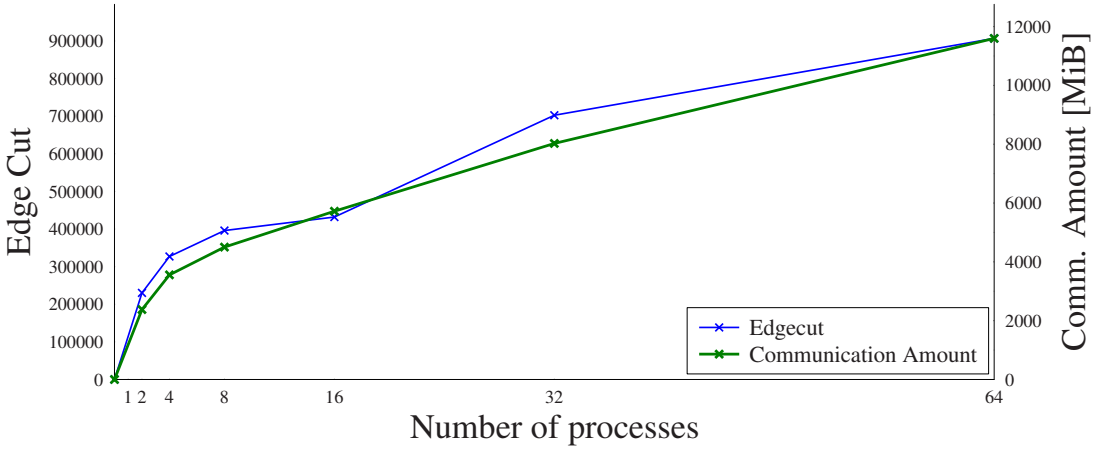


Figure 5.3.: Communication amount and edge cut on a proportional scale vs. number of processors

point and remains there until the end of the execution.

The memory profiles indicates, that the height of the initial plateau is the memory requirement of the reunification buffer, while the peak at the beginning is the memory allocated during the graph partitioning. For both processes the peak has a height of about one gigabyte. The memory usage stagnates after the preprocessing is done, as the simulation routines mainly read and write from the buffer.

All test runs run with 200 patches per mean free path length. Increasing this simulation parameter increases the accuracy of the results, but the memory overhead scales in N^2 for this parameter. At some point the overhead per process will grow larger than the amount of RAM gained by adding a cluster node. Thus, this approach does not scale to arbitrarily accurate simulations.

Communication Amount and Edge Cut

Unfortunately, the graphs produced by the data partitioning algorithm are too large for convenient visualization. The graph quickly contains millions of vertices. Therefore, no figures of the partitioned graphs are shown, and the edge cut must suffice.

Figure 5.3 is a graph that shows the edge cut and total communication volume plotted against the number of processes. Edge cut and communication volume are drawn on proportional axis scales. The plot shows that the amount of communication occurring during the simulation correlate.

Strictly speaking, the correlation between communication amount and edge cut, if there is any, is not proportional. Most edges have a weight of one, but Section 4.3.2 discussed a case where edges get a weight of two. So the number of edges cut cannot be a proportional predictor of communication amount.

The edges with a weight of two are, however, only a small fraction of the edges. Additionally, the graph partitioning will avoid cutting these edges. Edges with a

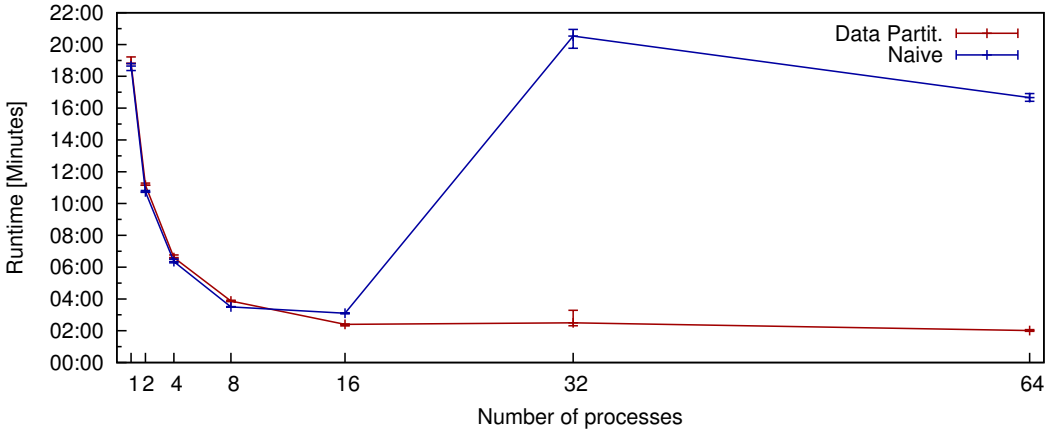


Figure 5.4.: Runtime vs. number of processors

weight of two should be the exception.

Unfortunately, there is no edge cut for the naïve data decompositions. As seen in Table 5.1, the runs with the naïve data decomposition have significantly higher communication amounts.

Analysis of Runtime

One question of interest is, how well the original parallelization algorithm attains passable speedups for multiple processes. Figure 5.4 plots the runtime in relation to the number of processes, for both the test runs with the graph partitioning algorithm and the naïve data decomposition. Each run was repeated five times and the variation is indicated as error bars.

Both algorithms display steady speedups for up to 16 processors. The runs with the naïve data decomposition are faster than the runs with the data partitioning algorithm. The variation of the runtimes is very low.

For the runs with 32 processes the two graphs show very different runtimes. The run with the data partitioning displays a slight increase of runtime, while the naïve decomposition causes the runtime to increase to a median value of 21 minutes, which is double the runtime of the run with one processor. Both of these runs show large relative variation in runtime.

For 64 processes the runtime of the runs with the data partitioning algorithm decrease slightly in comparison to 32 processors. The run with the naïve data decomposition require about 10 minutes of runtime, which is faster than for 32 processes, but still slower than for 1 process.

As seen in Table 5.1, the naïve data decomposition produces 15 GiB of traffic for all runs from four to 64 processes. The amount of communication does increase with the number of processes, but only a by megabytes in the single digits. The only case where further subdivision of decompositions does not increase the amount of commu-

nication is, if the decomposition is maximally bad to start with. As demonstrated by the example in Section 3.3, naïve decompositions tend to produce bad results. In the experimental runs, a naïve decomposition into four parts appears to produce a decomposition that is very close to the worst one.

Given that the required communication amount is close to constant for the runs with four to 64 processes, the runtime behavior is surprising. The runs with 4 and 8 processes are executed on one cluster node. The processes sharing a cluster node appear to communicate faster than over the Ethernet interconnect between nodes.

Even so, the the run with 16 processes requires two cluster nodes but still requires only 3 minutes runtime. 32 processes take place on four cluster nodes, and require 20 minutes runtime. The runtimes is then reduced for the run with 64 processes. All these have the same memory requirements, and—of course—the same simulation parameters. This phenomenon could be caused by the interconnect. If the interconnect is a torus, the two nodes running the 16 processes may be connected directly. If then, some of the nodes that run the 32 processes are not directly connected communication would be very inefficient. For 64 processes this would be less of a problem, because each process communicates less that in a run with 32 processes.

Moreover, Table 5.1 shows that runtime of the data partitioning algorithm seems to increase with the number of processes. For runs with 64 processes, that only have a complete runtime of 2 minutes, the preprocessing alone requires a minute. This is a limit to the parallelism of the parallel SPR algorithm.

Analysis of Resulting Echograms

During the simulation each process generates its own echograms. After the simulation is completed, the echograms are added to form the complete echogram. Figure 5.5 shows these cumulative echograms for runs with eight processes. Both cumulative echograms add up to the reference echogram.

The processes with the ranks 1 and 5 do not contribute any energy to the echograms. There are two reasons why a process may not contribute energy to the echogram. As already discussed, parts of the buffer are empty, and entire processes might receive parts of the buffer that cannot be filled with particles. The second reason is, that a process only simulates particles that are not registered by receivers.

If a processor simulates a sound particle and a receiver is in the line segment between the origin and the destination patch, the process will register the energy of that particle in its histogram. Therefore, the echograms of the individual processes are dependent on the data decomposition. The data decomposition produced by the data partitioning algorithm and the naïve decomposition assign different rows of the reunification buffer to the processes. Figure 5.5 shows that the different decompositions produce different cumulative echograms.

For a simulation with the same parameter the echogram is always the same, because sound propagation through scenes is deterministic. Thus the cumulative echograms always adds up to the same values, independent of the data decomposition.

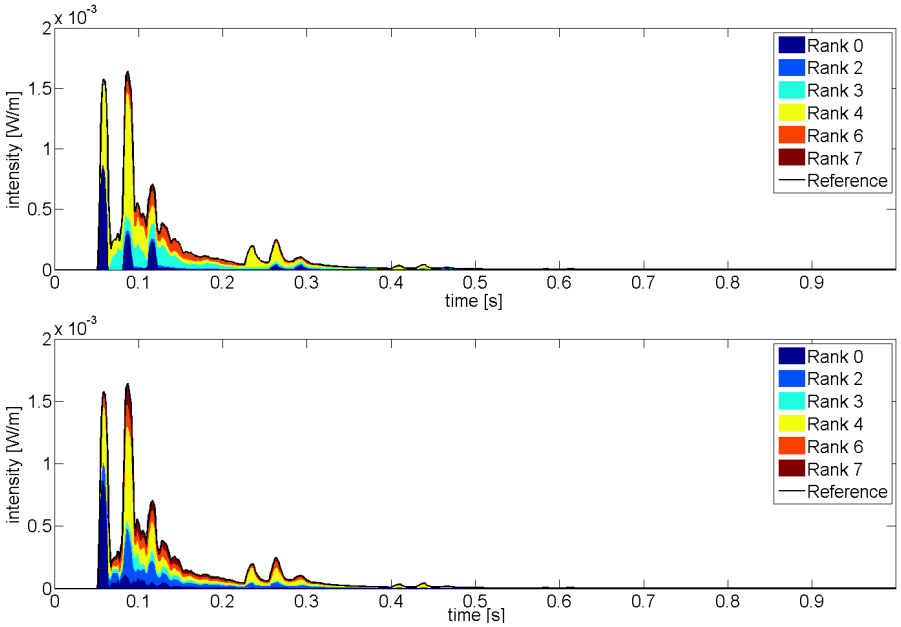


Figure 5.5.: Cumulative plot of the echograms predicted by each process from a run with eight processes. The upper echogram is produced by the naïve data decomposition, the lower one by the data decomposition.

These echogram can be used as a test of the implementation. Both, the parallel version with the naïve decomposition and the version with the data partitioning algorithm yield the reference echogram.

5.4. Threats to Validity

This section mentions threats to the validity of the simulations results or the conclusion drawn therefrom. Threats to internal validity will be addressed first, and after that threats to external validity.

5.4.1. Threats to Internal Validity

Most threats to internal validity for experiments on cluster computers concern the use of the hardware. This section will address hardware issues that may influence runtime and a threat caused by the implementation of the algorithms.

The nodes that make up the cluster are not identical. If the processes run on machines with different amounts of RAM or different CPUs it will influence the runtimes. Runs from cluster nodes with different hardware are not comparable. To produce results that are comparable with each other, only all runs were made with cluster nodes with identical hardware.

Indeed, even of the used cluster nodes have the same hardware, the interconnect

Processes	Cluster Nodes
1	1×n001
2	2×n001
4	4×n001
8	8×n001
16	8×n001, 8×n002
32	8×n001, 8×n002, 8×n003, 8×n004
64	8×n001, 8×n002, 8×n003, 8×n004 8×n005, 8×n006, 8×n007, 8×n010

Table 5.2.: The mapping which nodes are used for a given number of processes

between nodes might be different. Most clusters do not have a fully switched interconnect, so some cluster nodes have direct connections, but others do not. Hence, the available bandwidth between any two cluster nodes are not identical. Every run is repeated five times to determine runtime variance. If these runs occur on different cluster nodes, the nodes might have different available bandwidth, which would cause different runtimes. This threat was controlled by using the same cluster nodes for every run with the same number of processes. Table 5.2 contains the cluster nodes used for runs with a given number of processes, a complete specification of the cluster is available [3]. Every runs uses the cluster node "n001". Runs with eight processes require two cluster nodes so "n001" is used and "n002" is added. For the other nodes further nodes are added until the required number is met.

The cluster nodes run a Linux, which us in essence a time sharing system. In this case time sharing is undesirable, because other running processes might block the CPU or poison the case, which would falsify the runtimes. To ensure nothing interferes with the running of the program the batch system is instructed to schedule all used nodes exclusively to that task. No other dedicated task is scheduled on the cluster nodes in involved in a run, so the SPR implementation can make full usage of the available RAM, CPU, and available bandwidth. Thus, free resources, such as CPU cores that are not used by the simulation, remain unused.

The exclusive scheduling of the cluster nodes for all runs is a best effort to ensure accurate runtime results. However, the cluster nodes and the traffic on the interconnect from other nodes cannot be fully controlled for.

Another threat to validity is the implementation of the algorithms. A faulty implementation of the algorithms would invalidate the data. The results of the implementations and of all test runs were checked against a reference. Section 5.5 contains echograms of two runs, compared to the reference. The same compiler and libraries were used for the reference echogram.

All the experiment data is saved and available for review. Some information, such as exact version numbers and the command line flags used are in the appendix.

5.4.2. Threats to External Validity

Only one test scene with one set of test parameters has been used for all runs. One might think that this threatens external validity, because other scenes or simulation parameters might cause the implementation to show different behavior than the scene and parameters chosen for this run. This different behavior could then manifest in different efficiencies of the data partitioning algorithm or the parallel SPR implementation itself.

Simulation parameters, used in the test runs, are realistic and not particularly chosen for their influence on the program execution. The choice of scene or simulation parameters does not influence functionality of the parallel SPR algorithm, or significantly alters its efficiency compared to the sequential version. The scene and simulation parameters only influence the size and the occupancy behavior of the buffer, none of which impacts the functionality of the parallel SPR algorithm. The only simulation parameter that has a significant influence on the efficiency of the SPR algorithm in comparison to the serial version is the split-up factor. Very large split-up factors lead to many writes operations to the buffer, which are potentially remote, and thus slow.

The graph model is not influenced by the simulation parameters, either. Independent of the split-up factor, the graph remains sparse for large numbers of vertices. The simulated scene is a convex room, thus a non-convex room is a significant change not analyzed by the experiment. For dense buffers, sound propagates from each patch to each other possible patch. If the simulated scene is non-convex some patches do not have a line of sight. The data partitioning algorithm models the fact that no sound can travel between those patches, thus the graph model should be even more efficient.

5.5. Inferences

This describes inferences drawn from the data; whether and why the data uphold the hypotheses. We will examine the hypotheses in turn.

5.5.1. Parallelization Objectives

The first hypothesis is the fulfillment of the parallelization requirements, both on the memory usage and the runtime behavior. First the memory usage, and then the runtime behavior, of the parallel SPR algorithm and the data partitioning algorithm will be examined.

Memory Usage Requirements

The data parallel approach to SPR is successful in lowering the memory requirements for individual processes by distributing the buffer among multiple processes.

Although the memory usage test runs include the data partitioning algorithm, the memory usage pattern without the data partitioning can be inferred from the data. Since parallel SPR with a naïve data decomposition does not have memory intensive

preprocessing steps, such as graph partitioning, the memory usage is dominated by the reunification buffer. The size of the communication buffers and the mapping tables is less than 1% per process. For 32 and 64 processes the size of the local reunification buffer becomes less than one gigabyte per process, and is not the largest memory structure.

The graph partitioning requires a significant amount of memory per process. This is problematic, especially because the data partitioning was supposed to help scaling to large number of processes. In that respect the data partitioning algorithm fails the memory aspect of the parallelization requirements, at the moment.

In the implementation the entire reunification buffer is allocated for the generation of the graph model, and remains allocated during the graph partitioning. The generation of the graph model only requires the buffer to have one time index. The observed memory peak can effectively be eliminated by allocating a buffer with one time index for the data partitioning, and extending the buffer after the data partitioning algorithm completed.

For both data decompositions, the naïve one, and the decomposition obtained from the data partitioning algorithm, some processes store no particles throughout the entire simulation. This is to be expected for both decompositions. As already discussed, some rows of the buffer can never hold particles, because the physics of the simulation forbids them. Due to technical issues, the reunification buffer allocated by the SPR introduces even more overhead. Thus, for the scene and parameter set used in the simulation 70% of the reunification buffer can not be occupied.

A naïve partitioning is likely to decompose the buffer in a way that leaves some processes with permanently empty buffer rows.

The data partitioning algorithm actually proactively produces decompositions, because the algorithm aims to reduce the edge cut. If, for example, all occupy-able rows can be mapped to one process, the graph partitioning should do so, as it will reduce the edge cut to zero.

Runtime Requirements

The runtime requirement part of the parallelization objectives is concerned with the scaling of the algorithms to large process numbers. First, the runtime of the parallel SPR algorithm, then of the data partitioning algorithm will be examined.

Parallel SPR simulations with naïve data decomposition on multiple nodes have mixed runtimes. For eight processes on two cluster nodes SPR still attains speedups, but for more processes the test runs take longer than the runs with one process. The test runs with using the data partitioning algorithm on the other hand, display speedups even for the runs with higher number of processors.

The parallelization objectives demand a speedup for large numbers of processes. Data partitioning computes a data decomposition that, as required, produce these speedups even for 64 processes.

Technically, the data partitioning fails the requirements set by the parallelization objectives, because the algorithm requires a significant portion of the runtime. How-

ever, the runtime of the data partitioning algorithm seems to pay off, as the naïve data decomposition requires 16 minutes for 64 processes.

The runtimes of the data partitioning algorithm do not scale well. The data is not conclusive on whether there is a speedup at all, but if there is one it is too small. Thus, the data partitioning algorithm also fails the runtime requirement part of the parallelization objective.

5.5.2. Effectiveness of the Data Partitioning Algorithm

The second hypothesis is, that the data partitioning decreases the required message passing and the runtime of the parallel SPR algorithm in comparison to a naïve data decomposition.

It was previously established, that the occurring traffic correlated with the edge cut of the graph model. The runs with the naïve data decomposition have significantly higher communication volumes, than the runs with the data partitioning algorithm. Thus, the data partitioning algorithm does what it was supposed to, namely to produce data decompositions that reduces communication requirements.

The data decomposition computed by the data partitioning algorithm does not improve the runtime of the parallel SPR algorithm for the runs with eight or less processes, although it reduces the communication amount. These runs take place on a single cluster node, and the communication costs are very low.

The runs with more than eight processes show that the data partitioning algorithm works very much better than the naïve data decomposition. Although the runtimes are not significantly decreasing after 16 processes, the runs using the data partitioning sustain the low runtimes. The reunification buffer is too small to distribute well to 64 processes. A more accurate simulation would require a larger buffer and would probably show better speedups for processes with very high numbers of processes.

The experimental data supports the claim, that the data partitioning algorithm reduces the required communication amount, and with it, the runtime.

One anomaly remains: The communication amount for the runs with 64 processes are 11.5 GiB for the version with the data partition algorithm, and 15.5 GiB for the naïve data decomposition. This is an indicator that the buffer does not really decompose well into 64 parts. Indeed, if one considers the small difference in the occurring traffic it is surprising that the runtime for the two runs are so different. The data from the runs do not indicate anything that could explain this phenomenon.

6. Future Work

First, the implementation must be extended to provide support for non-convex scenes. The simulation non-convex scenes is important, because SPR explicitly models diffraction, which only occurs in multiple rooms.

More research into the runtime behavior of the parallel SPR algorithm is required. Since RAM usage scales with the number of processes the number of processes should be kept as small as possible. The available RAM can be maximized by placing each process on a dedicated cluster node, instead of having multiple processes per node, which was done for the evaluation.

The maximum number of cluster nodes used in the evaluation was 8 nodes for 64 processes. The runtime for the parallel SPR version with naïve data decomposition exemplifies that scaling the number of nodes can yield unexpected results.

Running only one process on a cluster node that has 8 or even more CPU cores is a large waste of resources. Runtimes can be further improved by using shared memory parallelism, such as OpenMP.

Memory overhead is another problem that requires more attention. The memory peak during the execution of the data partitioning algorithm can be eliminated by running the preprocessing before allocating the entire reunification buffer.

Even with the memory peak, the memory overhead increases linearly with the number of processes, and quadratically with the number of patches. Indeed, memory overhead always must scale linearly with the number of processes, because every process requires the entire mapping of rows to processes.

A possible way to mitigate the memory overhead is to use the graph model that is constructed as part of the data partitioning algorithm, to identify the rows that cannot hold particles. Rows with no incoming edges can be completely discarded, because they will never be reached. Rows with incoming but no outgoing edges cannot be discarded, as there might be a receiver on the path of the particle.

The existence of unused rows also causes workload imbalance. Because the number of unusable rows is possibly larger than the usable rows, can cause entire processes to be rendered useless because they only own unusable rows. The data partitioning algorithm can further aggravated this issue. Graph partitioning can decrease the edge cut, by assigning a process only unusable rows. Elimination of unusable rows during the data partitioning would eliminate this issue.

Adding cluster nodes increases RAM linearly, but the memory overhead per process is in $\mathcal{O}(N^2)$. Thus, for high accuracies the overhead for a process may be higher than the RAM available on a cluster node. To scale SPR to arbitrary accuracies, the memory overhead will need to scale linearly.

7. Conclusions

The thesis presented a data parallel SPR algorithm, that decomposes the reunification buffer by distributing the rows among the processes This approach lowers the RAM and computation requirements for processes running the SPR simulation. Lower requirement on processes make bigger and more accurate simulations feasible.

Additionally, the thesis introduced a data partitioning algorithm that minimizes message passing requirements. The algorithm generates a graph that models the data dependencies between the rows of the reunification buffer. Partitioning the graph yields a decomposition of the reunification buffer that reduces inter-process communication.

Empiric measurements show that the parallel SPR algorithm scales well as long as communication costs are cheap. For large numbers of processes parallel SPR with naïve decompositions require up to twice the sequential runtime. With the data partitioning algorithm, SPR simulations consistently attain speedups for all tested number of processes.

The runtime of the data partitioning algorithm does not display speedup, indeed the runtimes become linger for larger number of processes. 64 processes the data partitioning requires 71 seconds, while the runtime of the entire execution is only 2 minutes. This is a significant portion of the runtime, but it is still a consistent speedup, while the version with the naïve data decomposition does not attain speedup. Thus, the long runtimes of the data partitioning algorithm pay of for many processors.

Measurements of the memory usage indicate linear growth of RAM overhead with the number of processes. For the simulation run on the experiments an additional processes required one gigabyte of memory overhead.

All in all, the data parallel approach to SPR increases overall available RAM. When used with a data partitioning algorithm that takes the inter-dependencies between the parts of the reunification buffer into account, the runtime can be reduced even for large numbers of processes.

8. Bibliography

[1] Chevalier, C., and Pellegrini, F. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing 34*, 6–8 (2009), 6–8.

[2] Foster, I. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[3] Hamburg University of Technology. Linux Cluster "Apis". `http://www.tu-harburg.de/rzt/tuinfo/ausorg/para/apis/index.html`. Accessed: 2012/07/20.

[4] Hendrickson, B. Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling 25*, 2 (2000), 99–108.

[5] Jedlitschka, A., Ciolkowski, M., and Pfahl, D. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. Springer London, 2008, pp. 201–228.

[6] Karypis, G., and Kumar, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing 20*, 1 (1998), 359–392.

[7] Karypis, G., and Kumar, V. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distribtributed Computing 48*, 1 (1998), 96–129.

[8] Karypis, G., and Schloegel, K. ParMETIS, Parallel Graph Partitioning and Sparse Matrix Ordering Library. `http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf`. Acessed: 2012/07/30.

[9] Kosten, C. W. The mean free path in room acoustics. *Acustica 10* (1960), 245–250.

[10] Kuttruff, H. *Akustik: Eine Einführung.* S. Hirzel, 2004.

[11] Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications 12*, 1–2 (1998), 1–299.

[12] Pohl, A., and Stephenson, U. A Combination of the Sound Particle Simulation Method and the Radiosity Method. *Building Acoustics 18*, 1 (2011), 97–122.

[13] Siltanen, S., Lokki, T., and Savioja, L. Room Acoustics Modeling with Acoustic Radiance Transfer. *International Symposium on Room Acoustics 18*, 1 (2011), pages not available.

[14] Vorländer, M. *Auralization: Fundamentals of Acoustics, Modelling, Simulation, Algorithms and Acoustic Virtual Reality (RWTHedition).* Springer, 2007.

[15] Walshaw, C., and Cross, M. JOSTLE: Parallel Multilevel Graph-Partitioning Software—An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, F. Magoules, Ed. Civil-Comp Ltd., 2007, pp. 27–58.

A. Experiment Setup Data

Problem File

Figure A.1 is the problem file that was used for all experiment runs. The figure gives the simulation parameters, such as the patches per mean free patch length. Also included is the simulated scene; two of the walls in the room are made of the material "Holzboden" with has an absorption factor of 0.1. The "Holzboden" relfects all sound particles geometrically. The two other walls are made of the material "Ziegelwand", with an absoprtion factor of 0.5. "Ziegelwand" refelcts 50% of the impacted sound energy via scattering and the other 50% geometrically.

Figure A.2 is a rendering of the scene by the GUI version of the SPR implementation. The walls are indicated by a black frame and the receiver is the small red dot, while the big blue circle is the receiver.

```
[Global]
PatchesPerMfpl = 200
SplitScattering = 10
SplitDiffraction = 0
TMax = 1
TSample = 0.001
NumberOfRays = 1000
FrequencyMode = 0
FrequencyBand = 17
Dapdf = 0
Eds = 2
Normalization = 1
Lambda = true

[Polygons]
NumberOfPolygons = 1

[Geometry]
PointsX_0 = 0, 0, 30, 30
PointsY_0 = 10, 0, 0, 10
Material_0 = Holzboden, Holzboden, Ziegelwand, Ziegelwand

[Sources]
NumberOfSources = 1
PointsX = 5.0704
PointsY = 2.434
Power = 1

[Receivers]
NumberOfReceivers = 1
PointsX = 25.1338
PointsY = 3.45477
Radius = 2
```

Figure A.1.: The SPR problem file used for all runs

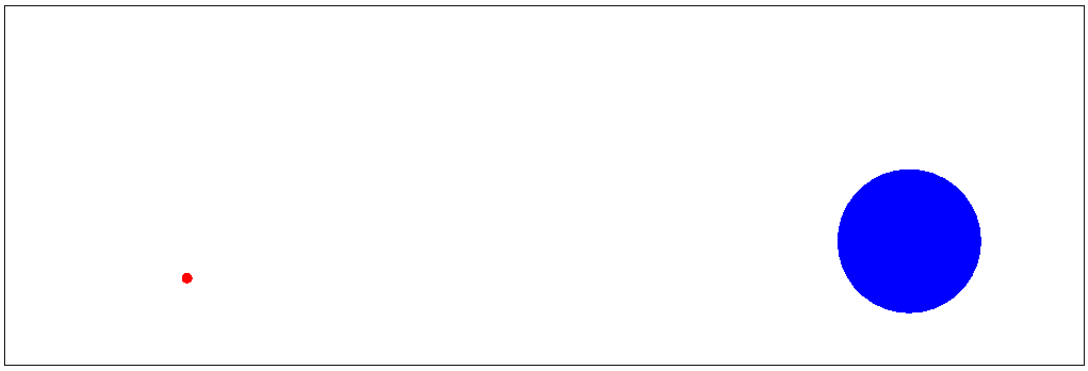


Figure A.2.: Scene used in test runs, as rendered by GUI version of SPR.
The red dot is the sound emitter, and the blue cricle is a sound receiver.

Program Versions and Flags

Operating System

Operating System: SUSE Linux Enterprise Server 10 (x86_64)
OS version: VERSION = 10 PATCHLEVEL = 2
Kernel version: Linux kernel version 2.6.16.60, gcc

Compiler

Compiler used: Intel Compiler version intel/11.0.069 64 bit
Compiler arguments: -O2 -falign-functions=16 -ansi-alias -fstrict-aliasing -w1 -Wcheck -wd654,1572,411,873,1125,2259

MPI Library

MPI library used: Intel MPI
MPI version: mpi-intel/3.2.0.011-run 64 bit
MPI arguments: `I_MPI_DEVICE sock` causes MPI to use socket interfaces for message passing (i.e., the shared memory interface is not used if multiple processes share a cluster node), `I_MPI_STATS 2` the MPI library will collect the occuring communication traffic, and `-n` with the number of processes specified

Batch System

PBS version: PBSPro_10.1.0.91350
PBS options: `place=free:excl`

Measurements

Runtime Measurements: GNU time 1.7
Memory profiles: Valgrind/massif (version: 3.7.0)
Memory profile options: `--tool=massif --detailed-freq=1 --max-snapshots=150`

B. Complete Run Results

Run ID	Runtime (MM.SS.ss)	# Proc.	Comm Amount (MiB)
B8BEDEF4	18:21.72	1	0
C418F250	18:22.29	1	0
6B56C1D3	18:39.19	1	0
3CCD592B	18:39.62	1	0
2CB70FCA	18:48.45	1	0
8AC18E55	10:42.23	2	11183.88500
11AC575E	10:42.53	2	11183.88500
2286E224	10:44.44	2	11183.88500
8838DF9A	10:46.78	2	11183.88500
D0CF9C54	10:48.78	2	11183.88500
F9316FED	6:15.62	4	15519.72280
49768736	6:18.96	4	15519.72280
7A8A7D9B	6:21.25	4	15519.72280
1F0371A2	6:24.68	4	15519.72280
A2805263	6:29.90	4	15519.72280
01DBB0E3	3:28.51	8	15520.09644
64D93846	3:29.31	8	15520.09644
C32A00A6	3:29.68	8	15520.09644
78B366A1	3:29.88	8	15520.09644
AA5DEF7D	3:30.26	8	15520.09644
87CC7AB1	3:04.23	16	15521.02268
911E4337	3:05.10	16	15521.02268
556B2594	3:06.31	16	15521.02268
17BFA5F6	3:07.08	16	15521.02268
1F407EB1	3:08.06	16	15521.02268
F8E3721E	19:46.37	32	15523.23025
5EEC40C3	22:20.22	32	15523.23025
13A7EC29	20:31.50	32	15523.23025
D9E015DF	20:43.15	32	15523.23025
CA63397C	20:57.23	32	15523.23025
8A4ADCA0	16:25.92	64	15528.35245
F71F0CEE	16:32.95	64	15528.35245
27CB80D1	16:39.66	64	15528.35245
7C40041D	16:41.57	64	15528.35245
A3DAB999	16:54.71	64	15528.35245

Table B.1.: Results of test batch with naïve data decomposition

Run ID	Runtime (MM:SS.ss)	# Proc.	Edge Cut	Comm. Amount (MiB)	Data Part. SS.ssss
93197DF4	18:47.91	1	0	0	34.8626
8E8B1A29	18:48.91	1	0	0	34.9695
DF82005C	18:49.15	1	0	0	34.8413
78688710	18:50.73	1	0	0	35.0530
24995857	19:13.28	1	0	0	35.0915
A8FE5D74	11:08.71	2	230219	2376.60500	35.6890
6834DC18	11:08.89	2	230219	2376.60500	35.6816
B360C8CB	11:08.93	2	230219	2376.60500	35.7127
063DF954	11:10.65	2	230219	2376.60500	35.6578
77540CA0	11:16.72	2	230219	2376.60500	35.8029
692DEFE0	6:31.42	4	327014	3557.1709	27.7493
6D2BB532	6:33.20	4	327014	3557.17090	27.7468
62AFE2DE	6:36.28	4	327014	3557.17090	27.7591
6F84A6CC	6:36.33	4	327014	3557.17090	27.7492
466DC0E9	6:46.04	4	327014	3557.17090	27.7979
C7B5381D	3:50.15	8	395961	4500.71732	25.7310
096A937D	3:50.46	8	395961	4500.71732	25.8040
91DC8016	3:51.52	8	395961	4500.71732	25.6873
B37E0FD4	3:54.61	8	395961	4500.71732	25.6936
AF26C3B4	3:55.49	8	395961	4500.71732	25.7199
6C940993	2:18.10	16	431881	5720.83372	30.7317
6C05B8E9	2:19.53	16	431881	5720.83372	30.8090
423AC7BC	2:23.55	16	431881	5720.83372	32.6441
B6C89AC6	2:30.27	16	431881	5720.83372	30.7745
48C43C90	2:23.97	16	431881	5720.83372	30.9646
61A7F87F	2:18.52	32	702507	8025.70155	57.3009
AA6C2CBB	2:22.04	32	702507	8025.70155	57.4523
A5F05083	2:30.24	32	702507	8025.70155	55.6954
9D769B3F	3:13.78	32	702507	8025.70155	56.8929
BBB8D82D	3:16.86	32	702507	8025.70155	56.0789
78FE53A3	1:57.84	64	906848	11598.31652	71.2071
0B25B95A	1:58.60	64	906848	11598.31652	69.4990
4A415420	2:00.91	64	906848	11598.31652	69.9213
2F804E86	2:03.64	64	906848	11598.31652	70.9078
6E0F6C20	2:04.04	64	906848	11598.31652	69.3577

Table B.2.: Test batch for data partitioning algorithm

Run ID	Runtime (H:MM:SS.ss)	# Proc.	Edge Cut	Comm. Amount (MiB)	Ram (GiB)	Data Part. (S.sss)
B59B1C33	2:41:56.00	1	0	0	25.530	653.737
706648B8	2:42:16.00	1	0	0	25.530	653.533
7450F406	2:42:28.00	1	0	0	25.530	653.933
40BFD122	2:42:40.00	1	0	0	25.530	653.326
2BE1F889	2:43:09.00	1	0	0	25.530	656.706
CA1DF5CD	1:28:54.00	2	230219	2376.60500	28.080	537.946
1E440ABA	1:29:12.00	2	230219	2376.60500	28.080	538.634
BB3C5975	1:29:18.00	2	230219	2376.60500	28.080	537.609
A3368527	1:29:19.00	2	230219	2376.60500	28.080	537.459
DF5DB319	1:29:26.00	2	230219	2376.60500	28.080	538.334
41F773F6	0:58:25.42	4	327014	3557.17090	30.225	479.505
3A76921A	0:58:31.28	4	327014	3557.17090	30.292	481.025
1EAAEDA1	0:58:37.02	4	327014	3557.17090	30.159	480.565
9CD77A6F	0:58:52.83	4	327014	3557.17090	30.159	480.399
A59E5667	0:58:58.90	4	327014	3557.17090	30.159	478.776
954B29C8	0:37:13.28	8	395961	4500.71732	33.915	508.806
FCFE3298	0:36:44.68	8	395961	4500.71732	33.904	507.253
B2862C35	0:36:51.00	8	395961	4500.71732	33.929	509.382
AC5E9992	0:36:53.33	8	395961	4500.71732	33.954	508.972
D5D17B79	0:36:55.17	8	395961	4500.71732	33.904	508.684
3A57793D	0:24:03.23	16	431881	5720.83372	40.889	567.965
12BCDC65	0:24:03.32	16	431881	5720.83372	40.889	567.166
DBBC42B1	0:24:06.62	16	431881	5720.83372	40.889	567.232
A79A188F	0:24:07.70	16	431881	5720.83372	40.889	564.988
1E15A4A1	0:24:13.24	16	431881	5720.83372	40.889	569.989
AD50AE75	0:19:09.27	32	702507	8025.70155	55.580	656.612
4050FF4B	0:19:10.02	32	702507	8025.70155	55.580	657.110
691F3CBE	0:19:14.19	32	702507	8025.70155	55.580	657.758
4E7AD6A4	0:19:30.89	32	702507	8025.70155	55.580	660.339
135EF9C3	0:19:44.61	32	702507	8025.70155	55.580	661.362
E2E5DCF4	0:17:21.12	64	906848	11598.31652	84.457	755.939
95B98C4D	0:17:39.58	64	906848	11598.31652	84.457	761.918
BCF90DC1	0:17:43.39	64	906848	11598.31652	84.457	762.245
E8344836	0:17:44.82	64	906848	11598.31652	84.457	761.388
82B8AE55	0:17:54.61	64	906848	11598.31652	84.457	760.240

Table B.3.: Test batch for measuring RAM. This batch uses the data partitioning algorithm