Efficient Distributed Load Balancing for Parallel Algorithms

Biagio Cosenza

November 2010

Dottorato di Ricerca in Informatica
IX Ciclo Nuova Serie
Università degli Studi di Salerno

Supervisor
Prof. Vittorio Scarano

PhD Program Chair
Prof. Margherita Napoli

Supervisor
Prof. Vittorio Scarano, Dip. di Informatica,
Università degli Studi di Salerno, Italy

PhD Program Chair
Prof. Margherita Napoli, Dip. di Informatica,
Università degli Studi di Salerno, Italy

PhD Committee
Prof. Domenico Talia, Università della Calabria, Italy
Prof. Marco Faella, Università degli Studi di Napoli Federico II, Italy
Prof. Alfredo De Santis, Università degli Studi di Salerno, Italy

Dean
Prof. Alberto Negro, Dip. di Informatica,
Università degli Studi di Salerno, Italy

Thesis submitted on
November 30th, 2010

Date of defense
April 29th, 2011

Biagio Cosenza
ISISLab, Dip. di Informatica
Università degli Studi di Salerno
Via Ponte don Melillo, 84084 Fisciano (Salerno), Italy
cosenza@dia.unisa.it

Abstract

With the advent of massive parallel processing technology, exploiting the power offered by hundreds, or even thousands of processors is all but a trivial task. Computing by using multi-processor, multi-core or many-core adds a number of additional challenges related to the cooperation and communication of multiple processing units.

The uneven distribution of data among the various processors, i.e. the *load imbalance*, represents one of the major problems in data parallel applications. Without good load distribution strategies, we cannot reach good speedup, thus good efficiency.

Load balancing strategies can be classified in several ways, according to the methods used to balance workload. For instance, dynamic load balancing algorithms make scheduling decisions during the execution and commonly results in better performance compared to static approaches, where task assignment is done before the execution.

Even more important is the difference between *centralized* and *distributed* load balancing approaches. In fact, despite that centralized algorithms have a wider vision of the computation, hence may exploit smarter balancing techniques, they expose global synchronization and communication bottlenecks involving the master node. This definitely does not assure scalability with the number of processors.

This dissertation studies the impact of different load balancing strategies. In particular, one of the key observations driving our work is that distributed algorithms work better than centralized ones in the context of load balancing for multi-processors (alike for multi-cores and many-cores as well).

We first show a centralized approach for load balancing, then we propose several distributed approaches for problems having different parallelization, workload distribution and communication pattern. We try to efficiently combine several approaches to improve performance, in particular using predictive metrics to obtain a per task compute-time estimation, using adaptive subdivision, improving dynamic load balancing and addressing distributed balancing schemas. The main challenge tackled on this thesis has been to combine all these approaches together in new and efficient load balancing schemas.

We assess the proposed balancing techniques, starting from centralized approaches to distributed ones, in distinctive real case scenarios: Mesh-like computation, Parallel Ray Tracing, and Agent-based Simulations. Moreover, we test our algorithms with parallel hardware such has cluster of workstations, multi-core processors and exploiting SIMD vectorial instruction set.

Finally, we conclude the thesis with several remarks, about the impact of distributed techniques, the effect of the communication pattern and workload distribution, the use of cost estimation for adaptive partitioning, the trade-off *fast versus accuracy* in prediction-based approaches, the effectiveness of work stealing combined with sorting, and a non-trivial way to exploit hybrid CPU-GPU computations.

Keywords: parallel computing, load balancing, distributed algorithms, adaptive subdivision, dynamic load balancing, scheduling, scalability, task sorting, work stealing, agent-based simulations, Reynolds's behavioral model, mesh-like computation, ray tracing algorithms, parallel ray tracing, image space techniques, path tracing, Whitted ray tracing, SIMD parallelism, message passing, multi threading, GPU techniques

Acknowledgments

This dissertation could not have come about without the help and support of many people.

First, I would like to thank my advisor, *Prof. Vittorio Scarano*.
He has taught me how to articulate my ideas and how to step back to see the bigger picture. Particularly, I appreciate the freedom he gave me to explore my ideas; while this freedom has taken me down some garden paths, it has taught me a lot about the research process and strengthened my skills as a researcher. Moreover, the most important contribution that he has made in my life is that he has taught me how important is to have fun in my job.

I would like to express my gratitude to all members of the ISIS Research Lab: *Gennaro, Rosario, Delfina, Raffaele, Bernardo, Ugo, Ilaria* and *Pina* – and *Prof. Alberto Negro* and *Filomena De Santis*. A special mention is to *Gennaro Cordasco*, for his useful feedback and suggestions on my thesis, and to *Rosario De Chiara*, with his hints and fancy graphs.

There are several friends who I should thank for making my stay in Salerno enjoyable. Is a long list, comprising *Edoardo, Raffaele, Pina, Nazario, Terry*, my PhD mates, and many others countless friends.

A *Dankeschön* is to *Prof. Carsten Dachsbacher*, for patiently teaching me most of the bells and whistles of computer graphics. I entered the field of graphics with few prior experience, and he took me on and was patient as I learned the ropes. Working with him has been a wonderful learning experience, and I appreciate the time and effort he has taken to improve my work.

I should greatly appreciate people at VISUS in Stuttgart, especially *Prof. Thomas Ertl* for hosting me in several projects and thereby introducing me into his group, *Guido Reina* for letting me feel at home, *Gregor, Filip, Sven, Michael* and *Thomas* for many helpful discussions, and all other people at VISUS that have certainly broadened my view and made my stay there a great experience.

I would like to thank the PhD Committee members for kindly accepting to review this thesis.

I have to mention many other people (in random order): *Alexander Schultz* and *Rainer Keller* from HLRS Stuttgart, for always helping out once help was needed during the two HPC-Europa Projects; Manta team from University of Utah for sharing their code on ray tracing and for their feedbacks; *Björn Knafla* from Universität Kassel, for its insightful thoughts on Agent-based Simulation; *Ursula Habel* from IZ Stuttgart, for being helpful during the DAAD Scholarship; *Elena Orsini* from Universität Hohenheim, for her invaluable friendship; *Giovanni Erbacci* from CINECA Bologna for his helpful support with both ISCRA and HPC-Europa grants; *Roberto Ciavarella* from ENEA Portici for his quick solutions to my complex setup problems at ENEA Grid; the *Italians* group in Stuttgart for their nice *Stammtisch*.

My work has been funded by several grants: HPC-Europa++, HPC-Europa2, a DAAD (Deutscher Akademischer Austausch Dienst) Scholarship, and a ISCRA Cineca. I want to thank people behind these organizations for offering similar opportunities.

Last but not least, I would like to thank my family: my father *Fabio*, my mother *Caterina*, and my little brother *Giuseppe*.

Beyond people, I have a final thanksgiving for a place. A place where I had inspiring thoughts and ideas, where friendly people always have something to tell and ask, and their smiles made my time there valuable. This place is my hometown, *la Calabria*.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In 1965 Gordon Moore made a prediction about the semiconductor industry that has become the stuff of legend. Moore's Law predicted the incredibly growth that the semiconductor industry has experienced over the last 50 years [70]. Starting from nothing, it has now passed $200 billion in annual revenue and Moore's Law has become the foundation of a trillion-dollar electronics industry. However, many parameters relating to the industry have changed almost exponentially with time, including chip complexity, chip performance, feature size, and the numbers of transistors produced each year. Several times in the past, it appeared that technological barriers such as power consumption would slow or even stop the growth trends.

Lately, the same Moore [71] admitted that a new, more fundamental barrier is emerging that the technology is approaching atomic dimensions, raising all sorts of new challenges. He said:

> *No exponential change of a physical quantity can, however, continue forever. For one reason or another something limits continued growth. For our industry many of the exponential trends are approaching limits that require new means for circumvention if we are to continue the historic rate of progress.*

Nevertheless, Moore's Law is not dead yet. The number of transistors on modern processors continues to double every 18 months, but those transistors are now just manifesting themselves as additional processing cores.

In *single-core processor*[1], one way to increase performance is to increase clock rates, but with heating and energy concerns, that only goes so far. The increased density of *multi-core processors* [2] allows each core to be clocked well below its theoretical maximum, which assists with heat dissipation and power management.

Despite, in the past, the promise of parallelism has fascinated researchers, for at least three decades single-processor computing always prevailed. Such a

[1] Here the term indicates a single processor with a single core

[2] An architecture that supports multiple cores in a single processor package that replicates the cache coherent, shared address space architecture common to traditional multi-processor computers. Typically, multi-core processors put multiple cores in a given thermal envelope and emphasize the best-possible single-thread (or single-core) performance [59]

switch to parallel microprocessors is a milestone in the history of computing and the new belief now is that the number of cores on a chip doubles with each silicon generation.

In this new era, programmers who care about performance must get up off their recliners and start making their programs parallel. Today, increasing parallelism is the primary method of improving performance.

A major reason slowing deployment of parallel programs is that efficient parallel programs are difficult to write. Parallel programming adds a second dimension to programming: not just *when* will a particular operation be executed, but *where*, i.e. what processor will perform it. A vast number of parallelizable applications do not have a regular structure for efficient parallelization. Such applications require load balancing to perform efficiently in parallel. The load in these applications may also change over time, requiring rebalancing. The programmer is left with the choice of either distributing computation naively, producing poorly-performing programs, or spending more development time including load balancing code in the application.

1.1 Introducing Parallel Computing

1.1.1 The need of Parallel Computing

Nowadays, there is a strong need for computational power.

CFD (Computational Fluid Dynamics) applications require millions of calculations to simulate the interaction of liquids and gases with surfaces defined by boundary conditions. Even with high-speed supercomputers, only approximate solutions can be achieved in many cases. Numerical weather prediction uses current weather conditions as input into mathematical models of the atmosphere to predict the weather. In order to handle such as huge datasets and to perform complex calculations on a resolution fine enough to make the results useful, some of the most powerful supercomputers in the world are required. In chemistry, MD (Molecular Dynamics) simulations, in which atoms and molecules are allowed to interact for a period of time by approximations of known physics, give a view of the motion of the particles. This kind of simulation may involve from few thousand to millions of atoms. The huge datasets used by bioinformatics and astrophysics need time-consuming algorithms for their analysis. Computational finance utilizes various computationally intensive methods like Monte Carlo simulations, in order to understand financial risk of a specific financial instrument.

These represent just a fast outlook to a wide and growing area of applications for high performance computing, all having a strong need of computational power.

Simultaneously, the only way to assure higher compute availability is towards parallel computing. Hence, future high performance hardware will be inherently parallel, exposing together several forms of parallelism, such as (well known) *multi-processors* and (new) *many-cores*.

Luckily, real world applications are naturally parallel. But unluckily, parallel programming is hard.

Computing with multiple processors involves the same effort we had when computing single processor, but yet adds a number of new challenges related to the cooperation and communication of multiple processors. None of these new factors are trivial, giving a good reason of why scientist and programmers find parallel computing so challenging.

1.1.2 Why Now?

Researchers are discussing in deep this step toward parallel architectures and its impact in hardware and software [3]. In fact, today three walls are forcing microprocessor manufacturers to bet their futures on parallel microprocessors: the Power wall, the Memory wall [108] and the Instruction Level Parallelism (ILP) wall.

In the past, we stated that power is free but transistors are expensive. Now a *Power wall* let us deduce the opposite: power is expensive, but transistors are *free*. Hence, we can put more transistors on a chip than we have the power to turn on. This forces us to concede the battle for maximum performance of individual processing elements, in order to win the war for application efficiency through optimizing total system performance.

The *Memory wall*, i.e. the growing disparity of speed between CPU and memory outside the CPU chip [108], would become an overwhelming bottleneck in computer performance and it will change the way to optimize programs. Thus, *multiply* is no longer considered a harming slow operation, if compared to *load* and *store*.

Similarly, there are diminishing returns on finding more ILP [55].

Furthermore, performance improvements do not yield both lower latency and higher bandwidth, this because (across many technologies) bandwidth improves by at least the square of the improvement in latency [82].

It is clear that *many-core* [3] is the future of computing. Furthermore, it is unwise to presume that multi-core architectures and programming models suitable for 2 to 32 processors (or cores) can incrementally evolve to serve systems equipped with more than 1,000 processors. The elaborate tuning of next generation hardware let us suppose that auto-tuner tools should be more important than conventional compilers in translating parallel programs.

1.1.3 Parallel Architectures

In this thesis, we exploit several kinds of parallel architectures. In Appendix A, we show a list of hardware platforms used. We tried to exploit parallelism in

[3]The term *many-core* indicates to an architecture that supports multiple cores in a single processor package where the supporting infrastructure (interconnect, memory hierarchy, etc) is designed to support high levels of scalability, going well beyond that encountered in multiprocessor computer. Many-core processors put more cores in a given thermal envelop than the corresponding multi-core processors, consciously compromising single-core performance in favor of parallel performance [58].

different ways, using different parallel programming paradigms (e.g. message passing, multi threading and vectorial instructions) and hardware architectures (e.g. cluster of workstations with distribute memory and multi-core with shared memory).

Processing element and processor We often use the word processor. However, sometime is more indicated the generic term *processing element*. A processing element refers to a hardware element that executes a stream of instructions. The context defines what unit of hardware is considered a processing element (e.g. core, processor, or computer). Note that the context depends on both hardware and software platform configuration. For example, let us consider a cluster of SMP workstations. In some programming environments, each workstation is viewed as executing a single instruction stream; in this case, a processing element is a workstation. A different programming environment running on the same hardware, however, may view each processor or core of the individual workstation as executing an individual instruction stream; in this case, the processing element is the processor or core rather than the workstation.

Data processing Following Flynn's classification of parallel architectures, an architecture can be classified by whether it processes a single instruction at a time or multiple instructions simultaneously, and whether it operates on one or multiple data sets [47].

SISD (Single Instruction Single Data) machines are conventional serial computers that process only one stream of instructions and one stream of data. Instructions are executed sequentially but may be overlapped by pipelining. As long as everything can be regarded as a single processing element, the architecture remains SISD.

SIMD (Single Instruction Multiple Data) encompasses computers that have many identical interconnected processing elements under the supervision of a single control unit. The control unit transmits the same instruction, simultaneously, to all processing elements. Processing elements simultaneously execute the same instruction and are said to be *lock-stepped* together. Each processing element works on data from its own memory and hence on distinct data streams. The execution of instructions is said to be synchronous, because every processing element must be allowed to complete its instruction before the next instruction is taken for execution. For example, array processors and GPU are SIMD machines (for GPU, is often preferred the acronym SIMT, Single Instruction Multiple Threads). Another well-known example of SIMD is the Intel SSE vectorial instruction set (such instructions are explored in Chapter 3).

In MISD (Multiple Instruction Single Data), multiple instructions operate on a single data stream. It is an uncommon architecture, which is generally used for fault tolerance. There are few examples of such computer.

When using MIMD (Multiple Instruction Multiple Data), multiple autonomous processing elements simultaneously execute different instructions on different data. Computers have many interconnected processing elements, each of which

has their own control unit. The processing elements work on their own data with their own instructions. Tasks executed by different processing elements can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. Distributed systems are usually MIMD architectures, exploiting either a single shared memory space or a distributed one. Examples of such platforms are cluster of workstations, multi-processors PCs, and IBM SP. In this dissertation, all our works exploit MIMD parallelism, in particular cluster of workstations (Chapters 2, 3 and 4) and multi-threading (Chapter 3).

Memory Flynn classification is limited to data processing. However, with next generation parallel hardware, memory will be more important than processing. By a memory-centric viewpoint, we roughly classify parallel platforms in three categories: distributed memory, shared memory and shared address space.

Distributed memory machines are considered those in which each processor has a local memory with its own address space. A processor's memory cannot be accessed directly by another processor, requiring both processors to be involved when communicating values from one memory to another. An example of distributed memory machines is a cluster of workstations. All hardware architectures used in this dissertation and enlisted in Appendix A are clusters of workstations with distributed memory system.

Shared memory machines are those in which a single address space and global memory are shared between multiple processors. Each processor owns a local cache, and its values are kept coherent with the global memory by the operating system. Data exchange between processors happens simply by placing the values, or pointers to values, in a predefined location and synchronizing appropriately

In this dissertation, we developed some specific techniques for shared memory machines (in particular in Chapter 3).

Shared address space architectures are those in which each processor has its own local memory, but a single shared address space is mapped across distinct memories. Such architectures allow a processor to access other processors' memory without their direct involvement, but they differ from shared memory machines in that there is no implicit caching of values located on remote machines.

Many modern machines are also built using a combination of these technologies in a hierarchical fashion. For instance, most clusters consist of a number of shared memory machines connected by a network, resulting in a hybrid of shared and distributed memory characteristics. IBM's large-scale SP machines are an example of this design (Appendix A.5).

We may further distinguish UMA (Uniform Memory Access) from NUMA (Non-Uniform Memory Access) machines: In the first, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data; in the latter, memory access time depends

on the memory location relative to a processor.

1.1.4 Scalability and Efficiency

As in the sequential world, many metrics from program execution provide hints to the overall efficiency and effectiveness of a running program. These metrics are crucial in order to understand and evaluate implementations running on such new parallel architectures. We introduce some measures of the effectiveness of a parallel program commonly used in parallel computing: speedup, scalability and efficiency. Thus, we discuss about two tools used to predict and estimate parallel speedup: The Amdahl and Gustafson's Laws.

Speedup In parallel computing, speedup refers to how much a parallel algorithm is faster than a corresponding sequential one. It is defined as the single-processor execution time divided by the execution time on p processors:

$$speedup_p = \frac{T_1}{T_p}$$

where p is the number of processors, T_1 is the execution time of the sequential algorithm, T_p is the execution time of the parallel algorithm with p processors.

When running an algorithm with *linear speedup*, doubling the number of processors doubles the speed. As this is ideal, it is considered very good scalability.

There are two ways to indicate T_1. If we consider the execution time of the best sequential algorithm, then we have an *absolute speedup*. Instead, if we consider the execution time of the same parallel algorithm on one processor, we have a *relative speedup*. Of course, the best serial implementation is faster than the parallel one with one processor. Relative speedup is usually implied if the type of speedup is not specified, because it does not require implementation of the sequential algorithm.

It is a challenging task to achieve a good speedup. This because the parallel implementation of most interesting programs requires work beyond, which is not required for the sequential algorithm (e.g. synchronization and communication between processors).

In some rare cases, a *super linear speedup* may happen. The rationale behind a super linear speedup is that the parallelization of many algorithms requires, on each processor, allocating approximately $1/p$ of the sequential program's memory. This causes the working set of each processor to decrease as p increases, allowing it to make better use of the memory hierarchy. If this effect overcomes the overhead of communication, we can reach a linear, or even a super linear speedup. An example of problem where super linear speedup can occur is a problem performing backtracking in parallel: One processor can prune a branch of the exhaustive search that another processor would have taken otherwise.

Scalability Generally speaking, scalability indicates the ability of a parallel system to handle a growing amount of processors. In the context of high performance computing there are two common notions of scalability: *strong scalability* and *weak scalability*. The first defines how the solution time varies with the number of processors, for a fixed total problem size. The latter defines how the solution time varies with the problem size, for a fixed number of processors. In this dissertation, if not specified, we indicate with *scalability* the *strong scalability*.

When we discuss about scalability, we often refer to the parallel speedup of a program. In fact, parallel performance scalability is typically reported using a graph showing speedup versus the number of processors.

Efficiency A further metric used to measure parallel performance is efficiency. Parallel efficiency is so defined:

$$efficiency_p = \frac{speedup_p}{p} = \frac{T_1}{pT_p}$$

Efficiency is a value between zero and one (or a percentage value), indicating how much the processors are utilized in solving the problem. Algorithms with linear speedup show an efficiency of 1, while algorithms difficult to parallelize have efficiency that approaches zero as the number of processors increases (e.g. $1/log_p$).

Amdahl's Law As Amdahl observed 40 years ago, the less parallel portion of a program can limit performance on a parallel computer [2]. Amdahl's law states that, if p is the number of processors, α is the amount of time spent (by a serial processor) on serial parts of a program and $(1 - \alpha)$ is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then speedup is given by

$$speedup_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

This law is used to find the maximum expected improvement to an overall system when only part of the system is improved. For instance, it can be used to predict the theoretical maximum speedup using multiple processors

Gustafson's Law Amdahl's law has been criticized for several reasons, e.g. it does not scale the availability of computing power as the number of machines increases. In 1988, Gustafson introduced a new law [52]. He proposed that the programmer sets the size of problems in order to use the available equipment to solve problems within a practical fixed time. According to the Gustafson's Law, the speedup is defined as:

$$speedup_p = p - \alpha(p - 1)$$

Gustafson's Law says that more processors are available, larger problems can be solved in the same time. Here, the target is to reformulate problems so that solving a larger problem in the same amount of time would be possible.

Both Amdahl and Gustafson's Laws are yet useful tools to predict the speedup using multiple processors.

1.1.5 Challenges on Parallel Programming

The development of parallel system will focus on new aspects.

On the parallel hardware front, things seem to be clearer. Hardware architects working on many-core systems seem to have a deep understanding about troubles and issues of designing future multi-core and multi-processor system. They know that there will be many-cores, and these cores in a single processor may be different. They know there will be a scalable on-die interconnect and caches will need to adapt to the workloads to maximize locality.

Conversely, on the software front, it's chaos. Do we need new programming languages? Alternatively, is it enough to extend existing languages?
New programming models and languages, to be successful, should be independent of the number of processors. We cannot think that languages designed for 16-32 threads can even work well with thousands of threads.

An aspect like synchronization is critical. The latency of synchronization is high and so it is advantageous to synchronize as little as possible. By converse, most modern networks perform best with large data transfers, hence using a higher granularity of data movement.

Data can be accessed by computational tasks that are spread over different processing elements. Thus, we need to optimize data placement so that communication is minimized, and to minimize remote accesses.

Efficient parallel programs should support together several models of parallelism: task-level parallelism, word-level parallelism, and bit-level parallelism.

Finally, the mapping of computational tasks to processing elements must be performed in such a way that the elements are idle (waiting for data or synchronization) as little as possible. This well-known problem, the *load balancing* problem, is probably one of the harder challenge for parallel programming.

Reconsidering metrics The advent of these new parallel architectures introduces new way to consider performance. In particular, several old beliefs about parallel performance metric should be reconsidered.

An old belief was that less than linear scaling for a multi-processor application is failure. Nowadays, with the current trends in parallel computing, any speedup via parallelism is a success.

Another old belief was that scalability is almost free. To build scalable applications requires to have care of load balance, locality, and resource contention for shared resources. Moreover, reaching scalability for architectures having a huge amount of processors is all but easy.

1.2 Why do we care about Load Balancing?

A way to introduce the *load balancing* problem is by using an analogy.

A construction company having hundreds of workers has to build a new district in a city. The district comprises several houses, each of which having several floors. Supposing that more are the worker, shorter is the time to build a house, the aim of the company manager is to build the whole amount of house as soon as possible. This seems a classical easy to parallelize problem: If we have w workers and h houses, the best way to assign job is to assign about w/h workers to each house. However, if we suppose houses are not equal, the problem increases of complexity: Some houses have more floors, or more rooms, or they need extra-time for furniture and details. This means that the time need by a house to be built is not the same for each house. This inequality in house workload affects the overall finishing time: if half the house requires the double of the time need by the remaining house, half the worker will be idle for half the working time. As a result, the total time need by using a naïve worker strategy is +33% higher than time need by the optimal one. Depending by the problem, things may be even worse. The workload can be extremely various (e.g. house A need $10x$ the time required by house B), with task arranged in a more complex way (e.g. that house A should be built before house B), having thousands of workers, or the building time cannot be exactly estimated *a priori*.

This analogy is helpful to understand how important is load balancing in parallel computing. To build houses in shorter time means to solve problem effectively. To have idle workers means to waste computational resources (i.e. processors). To have tasks not well balanced means to lose efficiency.

1.2.1 The Problem of Load Balancing

The execution time of a parallel algorithm *on a given processor* is determined by the time required to perform its portion of the computation plus the overhead of any time spent performing communication or waiting for remote data values to arrive. Instead, the execution time of the algorithm *as a whole* is determined by the longest execution time of any of the processors. For this reason, it is advisable to balance the computation and communication between processors in such a way that the maximum per-processor execution time is minimal.

This is referred to as load balancing, since the conventional wisdom is that dividing work between the processors as evenly as possible will minimize idle time on each processor, thereby reducing the total execution time.

For some applications with constant workloads, using static load balancing is sufficient. However, a wide range of applications has workload that are unpredictable and/or change during the computation; such applications require dynamic load balancers that adjust the decomposition as the computation proceeds.

Load imbalance is one of the major problems in data parallel applications. In fact, a common source of load imbalance is the uneven distribution of data

among the various processors in the system. Without good load distribution strategies, we cannot aims to reach good speedup, thus good efficiency.

The combination of both irregular and dynamic parallel applications with large-scale multi-core clusters poses significant challenges to achieving scalable performance. New scalable dynamic load balancing strategies are need that is why today it is yet a challenging problem.

1.2.2 Aspects of Load Balancing

Load balancing and Mapping Dividing a computation (henceforth, *decomposition*) into smaller computations (tasks) and assigning them to different processors for parallel executions (named *mapping*), represent two key steps in the design of parallel algorithms [65]. The whole computation is usually represented via a directed acyclic graph (DAG) $\mathcal{G} = \{\mathcal{N}, \mathcal{E}\}$ which consists of a set of nodes $\mathcal{N}$ representing the tasks and a set of edges $\mathcal{E}$ representing interactions and/or dependencies among tasks. The number and, consequently, the size of tasks into which a given computation is decomposed determines the granularity of the decomposition. It may appear that the time required to solve a problem can be easily reduced, by simply increasing the granularity of decomposition, in order to perform more and more tasks in parallel, but this is not always true. Typically, interactions between tasks, and/or other important factors, limit our choice to coarse-grained granularity. Indeed, when the tasks being computed are mutually independent, the granularity of the decomposition does not affect the performances. However, dependencies among tasks incur inevitable communication overhead when tasks are assigned to different processors. Moreover, the finer is the adopted granularity by the system, the more is the generated inter-tasks communication. The interaction between tasks is a direct consequence of the fact that exchanging information (e.g. input, output, or intermediate data) is usually needed.

Load balancing and mapping are two closely related problems. In fact, a good mapping strategy should strive to achieve two conflicting goals: (1) balance the overall load distribution, and (2) minimize tasks inter-processors dependency; by mapping tasks with a high degree of mutual dependency onto the same processor. As an example of dependency, many mapping strategies exploits tasks' locality to reduce inter-processors communications [78] but it should be emphasized that dependency can also refer to other issues such as locality of access to memory (effective usage of caching).

Load balancing strategies can be classified in several ways, according to the method used to balance workload.

Static vs Dynamic Static load balancing makes the tasks distribution before execution according to the information of the system workload. Such a decision may not apply to a dynamical environment. Dynamic load balancing, on the other hand, makes more informative load balancing decisions *during execution* by the runtime state information. In general, dynamic approaches result in

better performance. However, the drawbacks of the dynamic approaches include the runtime overhead for collecting the resource status and the need of precise information about performance prediction.

Centralized vs Distributed Load balancing mechanisms can also be classified as centralized and decentralized. The centralized approach adopts one computing node as the scheduler which gathers the system informations and performs load balancing decisions. Instead, the decentralized approach allows the nodes in the system involving in the load balancing decisions. However, it is very costly to obtain and maintain the dynamic system information. Despite that centralized algorithm have a wider vision of the computation, hence may exploit smarter balancing techniques, they have two problems: the presence of a global synchronization, and the communication bottleneck that involves the master node.

Prediction Prediction-based approach offers a further change to improve load balancing: If we know in advance an estimation of the computational time need by a task, we may use this information to distribute workload between processors in a better way. Such prediction could be exploited in several ways. For instance, we can perform an *adaptive partitioning* during decomposition, or instead improve load balancing decision at runtime (i.e. dynamic load balancing).

1.2.3 Intuition: Distributed Load Balancing is More Efficient

The key observation for improving load balancing of multi-processor (rather multi-core or many-core) architectures is that, in this context, distributed algorithms work better than centralized ones. This is due to several emerging factors.

First, having a huge amount of processors, synchronizations should be handled carefully. Global centralized synchronizations are a bottleneck and kill performance, especially when the number of processor is high. We need to synchronize the fewer possible number of processors, hence to move from centralized synchronization schemas to distributed ones.

Second, in modern architecture using several level of memory hierarchies, only coherent memory accesses are fast. Data locality is even good when performing data transfers: processor sharing a level of memory (e.g. L2 or L3) have faster transfer rate. This is another point that advantage distributed load balancing schemas.

1.3 Previous work

Several effective distributed load balancing strategies have been developed in the last couple of years, for a number of various applications [68].

According to [3], we roughly identify seven categories of parallel problem, discussing related approaches to load balancing.

Dense Linear Algebra, where data sets are dense matrices or vectors, is implemented by library such as BLAS [9] and SCALAPACK [21]. Generally, such applications use unit-stride memory accesses to read data from rows, and strided accesses to read data from columns. A classical approach to balance workload in SCALAPACK is the cyclic block data distribution, in order to assure scalability [42].

In *Sparse Linear Algebra*, data sets include many zero values. Data is usually stored in compressed matrices to reduce the storage and bandwidth requirements to access all of the nonzero values. An example is block compressed sparse row (BCSR). Because of the compressed formats, data is generally accessed with indexed loads and stores. DLA methods often present a higher load unbalance than SLA ones. For instance, the LU factorization approached in [39] requires accurate implementations and strategies when ported to a distributed memory parallel architecture [67].

Fast Fourier Transform (FFT) is an example of *Spectral methods*. They use multiple butterfly stages, which combine multiply-add operations and a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others. This class of problems usually requires all-to-all communication to implement a 3D transpose, which requires communication between every link. If we consider the FFT, there have been many implementations for different architectures, ranging from hypercube [62] to CRAY-2 [14]. Recently, Chen et al. optimized FFT on a multi-core architecture introducing strategies to balance workload among threads [20].

N-Body methods compute the interactions between many discrete points. Variations include particle-particle methods, where every point depends on all others, leading to an $O(N^2)$ calculation, and hierarchical particle methods, which combine forces or potentials from multiple points to reduce the computational complexity to $O(N \log N)$ or $O(N)$. Load balancing is critical for this class of problems, and several solution have been proposed [5, 4], even for more recent multi-core GPU architectures [61].

In *Structured Grids*, points inside a regular grid are conceptually updated together. Similar techniques have a high spatial locality. Updates may be in place or between two versions of the grid. In areas of interest, grid may be subdivided into finer grids (e.g. Adaptive Mesh Refinement) and the transition between granularities may happen dynamically. An interesting problem belonging to this class is Lattice Boltzmann simulations. Recent work in this topic introduced an auto-tuning approach in order improve performance on multi-core architectures [107].

In *Unstructured (or irregular) Grids*, location and connectivity of neighboring points must be explicit. Unlike structured grids, unstructured grids require a connectivity list, which specifies the way a given set of vertexes make up individual elements. Grids of this type may be used in finite element analysis when the input to be analyzed has an irregular shape. The points on the grid are updated together. Updates typically involve multiple levels of memory ref-

erence indirection, as an update to any point requires first determining a list of neighboring points, and then loading values from those neighboring points. Finite Element Method (FEM) is a known problem where adaptive refinement solutions introduce troubles in load balancing [40, 64, 109].

On *Monte Carlo methods*, calculations depend on statistical results of repeated random trials. Usually, they are considered an embarrassingly parallel problem, whereas communication is typically not dominant. An example of such methods is the Quasi-Monte Carlo Ray tracing (we will consider Parallel Ray Tracing as a case study). This class of computational problem often presents high irregularity, therefore load unbalance. For example, in the context of Parallel Ray Tracing, several previous works afford the problem of load balancing [35, 103].

There are many other problems not included in this classification, such as Graph Traversal applications, Finite State Machines, Combinational Logic, and many others Computer Graphics problems.

1.4 Contributions of This Dissertation

In this dissertation, we first show a centralized approach to load balancing (Chapter 2), then we propose some distributed approaches for two specific problems having different parallelization and communication pattern (Chapter 3 and 4). We try to efficiently combine different approaches to improve performance, in particular using predictive metrics to obtain a per task compute-time estimation, using adaptive subdivision, improving dynamic load balancing and addressing distributed balancing schemas. The main challenge tackled on this thesis has been to combine all these approaches together in new and efficient load balancing schemas.

1.4.1 Assessing Load Balancing Algorithms in Real Applications

Despite several theoretical works address load balancing with theoretical models and elegant solutions, we believe that nowadays architectures, which expose complex memories arrangement and different kind of parallelism together, are too complex and need real world case studies. The conventional way to guide and evaluate architecture innovation is to study a benchmark suite based on existing programs. Similarly, [3] introduces the Seven Dwarfs, which constitute class of parallel problems where membership in a class defines similarity in computation and data movement. The Dwarfs specifies a high-level abstraction, in order to allow reasoning about their behavior across a broad range of applications. Problems that are members of a particular class can be implemented differently and the underlying numerical methods may change over time, but the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future.

In this dissertation, we contribute to each proposed technique with implementations in real world scenario and well-known problems, discussing results and issues emerging from implementations on parallel hardware.

1.4.2 Parallel Ray Tracing

Many applications exhibit irregularity between units of parallel computation. Such as irregularities can be due to several factors.

Algorithm presenting recursion, where each branch of recursion reaches different deep, are a typical class of problem presenting load imbalance. An example of this kind of problem comes from the Computer Graphics: *Parallel Ray Tracing* . Depending by the particular rendering technique we use, we may have a difference of computation between pixels that is high. The paradox of ray tracing algorithms is that they present both an embarrassingly parallel pattern and a high work unbalance. Hence, a little or no effort is required to separate the problem into a number of parallel tasks, and there is not dependency (or communication) between those parallel tasks; however, a naïve subdivision and assignment policy do not guarantee best performance.

We study in deep this problem: First, we introduce Parallel Ray Tracing such as a case of mesh-based computations, proposing an interesting centralized balancing technique called Prediction Binary Tree (Chapter 2). Second, we develop a state of the art implementation that exploit parallelism in several ways (using SIMD, multi-threading, and MPI between node clusters), and we apply several techniques to improve load balancing in all parallelism levels (Chapter 3). In particular, we use a various and powerful set of tools like:

- randomized work stealing, a distributed dynamic load balancing scheme, popularized by the runtime system for the Cilk parallel programming language [10]
- GPU-based rendering techniques to compute a per-task prediction
- adaptive subdivision techniques based on cost prediction
- pooling strategy to best combine multi-threading parallelism with distributed multi-processors balancing techniques

The combined use of these tools is a winning strategy, in particular when we address complex parallel architecture having a high number of processors.

1.4.3 Agent-based Simulation

Irregularity often arises due to the sparsity present in the data. For example, in scientific simulations spatial sparsity of the system often translates into sparsity in the numerical model. An example of this kind of irregularity is a class of simulations dubbed *Agent-based Simulations*. These simulations represent a challenging problem for parallel load balancing for several reasons. First, data locality strongly leverages good parallel performance. Thus, we should carefully

handle expensive data movement between processors. In the context of agent-based simulation on distributed memory architecture, in particular, the use of algorithms like work stealing with very random steals is not suitable. By converse, such simulations seem to be a perfect candidate to experience different kind of distributed strategies. The second reason is that agent models often lend to clusterize agents, hence producing a high load unbalance. We afford in detail problems and issues of Agent-based Simulations in Chapter 4, proposing a new distributed dynamic load balancing schema.

1.4.4 Organization of the Thesis

In Chapter 2 we introduce the problem of load balancing in mesh-like computations to be mapped on a cluster of processors. We show a centralized and effective algorithm called Prediction Binary Tree in order to subdivide work in equally computationally-balanced tasks. Thus, we asset the problem on a significant problem, Parallel Ray Tracing.

In Chapter 3 we show a state of the art parallel implementation of the ray tracing algorithm, tuned for an hybrid cluster of multi-core workstations and a GPU visualization node. The highly optimized packet-based ray tracing implementation allows the computation of millions of ray-triangle intersections per second, and fully exploit modern multi-core CPUs or GPUs. Load balancing is attacked by presenting a method that uses a cheap GPU rendering technique to compute a cost map: an estimation of the per-pixel cost when rendering the image using ray tracing. Using this information, we improve load balancing, task scheduling, and work stealing strategies.

In Chapter 4, we focus on Agent-based Simulation where a large number of agents move in the space, obeying to some simple rules. We present a novel distributed load balancing schema for a parallel implementation of such simulations. The purpose of such schema is to achieve a high scalability. Our load balancing approach is designed to be lightweight and totally distributed: the calculations for the balancing take place at each computational step, and influences the successive step.

Finally, Chapter 5 outlines some important considerations emerged among all the work presented in such dissertation.

Appendix A enlists the hardware platforms used; Appendix B enlists the publications related to the dissertation.

2

Load Balancing on Mesh-like Computations

In this Chapter we consider mesh-like computations, where a set of t independent tasks are represented as items in a bidimensional mesh. We are interested in decomposition/mapping strategy for step-wise mesh-like computations, i.e. data is computed in successive phases.

We aim at exploiting the temporal coherence among successive phases of a computation, in order to implement a load balancing technique to be mapped on a cluster of processors. A key concept, on which the load balancing schema is built on, is the use of a *Predictor* component that is in charge of providing an estimation of the unbalancing between successive phases. By using this information, our method partitions the computation in balanced tasks through the *Prediction Binary Tree* (PBT). At each new phase, current PBT is updated by using previous phase computing time for each task as next-phases cost estimate. The PBT is designed so that it balances the load across the tasks as well as reduces *dependency* among processors for higher performances. Reducing dependency is obtained by using rectangular tiles of the mesh, of almost-square shape (i.e. one dimension is at most twice the other). By reducing dependency, one can reduce inter-processors communication or exploit local dependencies among tasks (such as data locality). Furthermore, we also provide two heuristics which take advantage of data-locality.

Our strategy has been assessed on a significant problem, Parallel Ray Tracing. Our implementation shows a good scalability, and improves performance in both cheaper commodity cluster and high performance clusters with low latency networks. We report different measurements showing that tasks granularity is a key point for the performances of our decomposition/mapping strategy.

2.1 Introduction

The number, and as a result, the size of tasks into which a given computation is decomposed determines the *granularity* of the decomposition. Increasing the granularity of decomposition, may help to have a better load balancing. How-

ever, several factors force our choice to coarse-grained granularity. For instance, dependencies among tasks incur inevitable communication overhead when tasks are assigned to different processors. Moreover, the finer is the adopted granularity by the system, the more is the generated inter-tasks communication. The interaction between tasks is a direct consequence of the fact that exchanging information (e.g. input, output, or intermediate data) is usually needed.

A good mapping strategy should strive to achieve two conflicting goals: (1) balance the overall load distribution, and (2) minimize tasks inter-processors *dependency*; by mapping tasks with a high degree of mutual dependency onto the same processor. As an example of dependency, many mapping strategies exploits tasks' locality to reduce *inter-processors communications* [78] but it should be emphasized that dependency can also refer to other issues such as locality of access to memory (effective usage of caching).

The mapping problem becomes quite intricate if one has to consider that:

1. task sizes are not uniform, that is, the amount of time required by each task may vary significantly;
2. task sizes are not known a priori;
3. different mapping strategies may provide different overheads (such as scheduling and data-movement overhead).

Indeed, even when task sizes are known, in general, the problem of obtaining an optimal mapping is an NP-complete problem for non-uniform tasks (to wit, it can be reduced to the 0-1 Knapsack problem [24]).

2.1.1 Mesh-like computations

Ore study focus on mesh-like computations, where a set of t independent tasks are represented as items in a bidimensional mesh. Edges among items in this mesh represent tasks dependencies. In particular, we are interested in *tiled* mapping strategies where the whole mesh is partitioned into m *tiles* (i.e., contiguous 2-dimensional blocks of items). Tiles have almost-square shape, that is, one dimension is at most twice the other: in this way, assuming the load in processors is balanced (in terms of nodes), the dependencies inter-processors are minimized because of isoperimetric inequality in the Manhattan grid.

Tiled mappings are particularly suitable to exploit the local dependencies among tasks, be it the *locality of interaction*, i.e., when computation of an item requires other nearby items in the mesh or when there is a *spatial coherence*, i.e., when computation of neighbors item access to some common data. Hence, tiled mapping, in the former case, reduces the interaction overhead, and, in the latter case, improves the reuse of recently data access (cache).

We are interested in decomposition/mapping strategy for step-wise mesh-like computations, i.e. data is computed in successive phases. We assume that each task size is roughly similar among consecutive phases, that is, the amount of time required by item p in phase f is comparable to the amount of time required by p in phase $f + 1$ (*temporal coherence*).

Figure 2.1: Interaction between components of a parallel scheduler using a Predictor. Arrows indicate how components influence each others: (left) Traditional approach; (right) Our system with the Predictor.

2.1.2 Our result

In this Chapter we present a decomposition/mapping strategy for parallel mesh-like computations that exploits the temporal coherence, among computation phases, to perform load balancing on tasks. Our goal is to use temporal coherence to estimate the computing time of a new computation phase using previous phase computing time.

We introduce an iterative novel approach to implement decomposition/mapping scheduling. Our solution (see Figure 2.1) introduces a new component in the system design, dubbed *Predictor* that is in charge of providing an estimation of the computation time needed by a given tile, at each "phase".

The key idea is that, by using the Predictor, it is possible to obtain a balanced decomposition without using a fine-grained granularity that may increase the inter-tasks communication of the systems, due to the interaction between clients, and, therefore, may harm the performances of the whole computation.

Our strategy performs a semi-static load balancing (decisions are made before each computing phase). Temporal coherence is exploited using a *Prediction Binary Tree* where each leaf represents a tile which will be assigned to a worker as a task. At the beginning of every new phase, the mapping strategy, taking into account the previous phase times as estimates, evaluates the chance of updating the binary tree. Due to the temporal coherence property it provides a roughly balanced mapping. We also provide two heuristics which exploit the PBT in order to leverage on data locality.

We validate our strategy by using interactive rendering with Parallel Ray Tracing [106] algorithm, as a significant example of such a kind of computations. In this example our technique is applied rather naturally. Indeed, interactive Ray Tracing can be seen as a step-wise computation, where each frame to be rendered represents a phase. Moreover, each frame can be described as a mesh of items (pixels) and successive computations are typically characterized by temporal coherence.

For Parallel Ray Tracing, our technique experimentally exhibits good performances improvements, with different granularity (size of tiles), with respect to the static assignment of tiles (tasks) to processors. Furthermore we also provided an extensive set of experiments in order to evaluate:

1. the optimal granularity with different number of processors;
2. the scalability of our proposed system;
3. the correctness of the predictions exploiting temporal coherence;
4. the effectiveness of the locality coherence heuristics exploiting spatial coherence;
5. the impact of resolution;
6. the overhead induced by the PBT.

It should be said that, besides other graphical applications (e.g. image dithering), there are further examples of mesh-like computations where our techniques can be fruitfully used, covering simple cases, such as matrix multiplication, but also more complex computations, such as *Distributed Adaptive Grid Hierarchies* [79].

2.1.3 Previous Works

Decomposition/Mapping scheduling algorithms can be divided into two main approaches: list scheduling and cluster-based scheduling. In list scheduling [80], each task is first assigned a priority by considering the position of the task within the computation DAG $\mathcal{G}$. Then tasks are sorted on priority and scheduled following this order on a set of available processors. Although this algorithm has a low complexity, the quality of scheduling is generally worse than that of algorithms in other classes. In cluster-based scheduling, processors are treated as clusters and the completion time is minimized by moving tasks among clusters [13, 65, 111]. At the end of clustering, heavily communicating tasks are assigned to the same processor. While this reduces inter-processor communication, it may lead to load imbalances or idle time slots [65].

In [76], a greedy strategy is proposed for the dynamic remapping of step-wise data parallel applications, such as fluid dynamics problems, on a homogeneous architecture. In these types of problems, multiple processors work independently on different regions of the data domain during each step. Between iterations, remapping is used for balancing the workload across the processors and thus, reducing the execution time. Unfortunately, this approach does not take care of locality of interaction and/or spatial coherence.

Several online approaches have also been proposed. An example is the work stealing model [12]. In this model when a processor completes its task it attempts to steal tasks assigned to other processors. We notice that, although online strategies are shown to be effective [12] and stable [7], they introduce communication overhead anyway. Furthermore, it is worth noting that online strategies, like work stealing, can be integrated with our assignment policy. In that case, being our load balancing efficient, online strategies introduce smaller overheads.

Many researchers have explored the use of time-balancing models to predict execution time in heterogeneous and dynamic environments. In these environments, performance processors are both irregular and time-varying because of uneven underlying load on the various resources. In [110] authors use a conservative load prediction in order to predict the resource capacity over the future time interval. They use expected mean and variance of future resource capabilities in order to define an appropriate data mappings for dynamic resources.

2.2 Our Strategy

Our strategy is based on a traditional *data parallel model*. In this model, tasks are mapped onto processors and each task performs similar operations on different pieces of data (*Principal Data Items* (PDIs)). Auxiliary global information (*Additional Data Items* (ADIs)) is replicated on all the workers. This parallelization approach is particularly suited to the *Master-workers paradigm*.

In this paradigm, the master divides the whole job (the whole mesh) into a set of tasks, usually represented by *tiles*. Then, each task is sent to a worker which elaborates the tile and sends back the output. If other tiles are not yet computed, the master sends another task to the worker. Finally, the master obtains the results of the whole computation reassembling partial outputs.

Crucial point in this paradigm is the granularity of the mesh decomposition: in fact, the relationship between m, number of tiles, and n, number of workers, strongly influences the performances.

There are two opposite driving forces that act upon this design choice. The first one is concerned about the *load balancing* and requires m to be larger than n. In fact, if a tile corresponds to a zone of the mesh which requires a large amount of computation, then, it requires much more time with respect to a simpler tile. Then, a simple strategy to obtain a fair load balancing is to increase the number of tiles, so that the complexity of a zone of the mesh is shared among different items.

On the opposite side, two following considerations suggest a smaller m. An algorithm that has large m requires larger *communication costs* than an algorithm with smaller m, considering both the latency (more messages are required; therefore, messages may be queued up) and the bandwidth (communication overhead for each message). Other considerations that would suggest to use small m are (a) the *locality of interaction* and (b) the *spatial coherence* that are motivated because (i) computation of a task relies usually on nearby tasks and (ii) two close tasks usually access some common data. Then, in order to make an effective usage of the local cache for each node, it is important that the tiles are large enough, so that each worker can exploit spatial coherence of tiles, having a good degree of (local) cache hits.

Our strategy takes into account all the considerations above, by addressing the uneven spread of the load by using a Predictor component (the PBT), with a negligible overhead. The goal we aim to is to keep the load balanced without resorting to increase the number of tiles. Thereby, our solution does not

increase significantly the data-movement overhead and it reduces tasks interactions. Therefore, we are able to address simultaneously and positively all the issues above, by providing a technique that uses a moderate amount of tiles.

2.2.1 The Prediction Binary Tree

In this section we present how we use the Prediction Binary Tree (PBT) to help balancing the load among the computing items. The PBT is in charge of directing the tiling-based load balancing strategy as follows: each computing phase is split into a set of m tiles (we assume, here, for sake of simplicity that $m = n$ but the arguments apply to general cases) and tiles size is adjusted accordingly to (an estimated) tile computing time that is set as the computational time as measured during the preceding phase. The hypothesis is that the computing time required by a tile on two consecutive phases are quite similar because of temporal coherence.

Now we define the Prediction Binary Trees and then describe an on-line algorithm which, before each computing phase, resizes unbalanced tiles in such a way to minimize the mesh computing time. A PBT T stores the current tiling being defined as a rooted binary tree with exactly m leaves, in which each (internal) node has 2 children. The root of T, called r, represents the complete mesh. The two children of an internal node v store the two halves (more details follow on how the mesh is split) of the mesh represented by v. Consequently, each level of T represents a partition of the mesh. Moreover, each internal node v represents a tile which is the sum of the tile assigned to the leaves of the tree rooted in v and consequently, the leaves of T (henceforth $L(T)$) represents a partition of the mesh. In order to maintain a good spatial coherence and minimize tasks interaction, the children of an internal node v which belongs to an odd (resp. even) level of T are obtained halving the tile in t along two horizontal (resp. vertical) axes. This assures that tiles have an almost-square shape (i.e. one dimension is at most twice the other). Each leaf $ell \in L(T)$ also stores two variables: $e(\ell)$ that is the estimate of the time for computing tile in ℓ and $t(\ell)$ that is time used by a worker to compute (in the last phase) the tile in ℓ. Figure 2.2 gives an example of a PBT, with the corresponding mesh partition on the left.

The PBT stores the subdivision of tiles and each leaf of T is a task to be assigned to a worker. At the end of each phase, the PBT receives (with the tile output) also the information about the time that each worker has spent on the tile. This time is received as $t(\ell)$ for each leaf, and is used as estimate by copying it into $e(\ell)$. By using the previous phase times as estimates, the PBT is efficiently updated for the next phase. Here we describe an effective and efficient way of changing the PBT structure so that the next phase can be executed (given the temporal coherence) more efficiently, i.e., equally balancing the load among the processors.

First we define the variance as a metric to measure the (estimated) computational unbalance that is expected given the tiling provided by the PBT

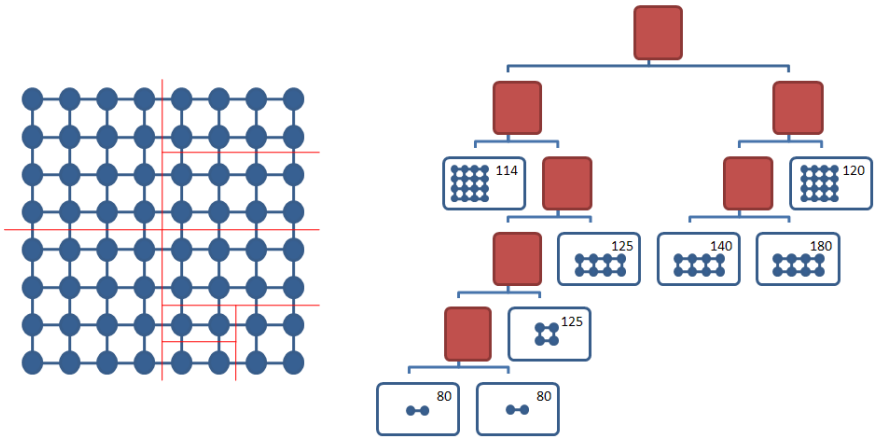


Figure 2.2: An example of a PBT tree: the mesh on the left has been computed with the computation times (in ms) for each tile shown on the leaves.

T:

$$\sigma_T^2 = \frac{1}{m} \sum_{\ell \in L(T)} (e(\ell) - \mu_T)^2,$$

where $e(\ell)$ represents the estimated time to compute the corresponding tile to the leaf ℓ of T and μ_T is the estimated average computational time, that is, $\mu_T = \frac{1}{m} \sum_{\ell \in L(T)} e(\ell)$. Clearly, the smaller the variance σ_T^2 is, the better is T's balancing of the load to the processors.

Given a PBT T at the end of a phase, the estimated computation time associated to each leaf, $e(\ell)$, is taken by the computation time $t(\ell)$ at the phase just executed; then, we use a greedy algorithm that finds the new PBT T^*. The idea of the algorithm PBT-Update (shown as Algorithm 1) is to perform a sequence of simultaneous *split-merge* operations, that consists in splitting a tile whose estimated load is "high", and merge two tiles (stored at sibling nodes) whose (combined) estimated load is "small".

We now prove, by means of the following theorem, that the PBT-Update algorithm terminates.

THEOREM 1. *Algorithm PBT-Update terminates after a finite number of iterations.*

Proof. We will show that PBT-Update goes from a PBT $T = T^{(0)}$ to a PBT $T^{(s)} = T^*$ through a set of PBTs $T^{(1)}, T^{(2)}, \ldots, T^{(s-1)}$ in such a way that $\sigma^2_{T^{(i)}} > \sigma^2_{T^{(i+1)}}$ for each $i = 0, \ldots, s-1$.

Let T be a PBT tree and T' be obtained from T by halving a leaf ℓ_a into two leaves ℓ_{a_1} and ℓ_{a_2} and merging two sibling leaves ℓ_{b_1} and ℓ_{b_2} into ℓ_b.

We prove first that, if $e(\ell_a)^2 > 4e(\ell_{b_1})e(\ell_{b_2})$ then $\sigma_T^2 > \sigma_{T'}^2$. So, it is not possible to improve the variance of the (estimation of the) computational time

by means of a simultaneous split-merge operation if $e(\ell_a)^2 \leq 4e(\ell_{b_1})e(\ell_{b_2})$ which is the test in line 8 of the algorithm.

Algorithm 1 PBT-Update

```
1: T ← CurrentPBT
2: for all ℓ ∈ L(T) do
3:     copy computational time t(ℓ) in estimated time e(ℓ)
4: end for
5: while true do
6:     let ℓ_a be the leaf in T with max e(ℓ), ∀ℓ ∈ L(T)
7:     let ℓ_{b1}, ℓ_{b2} be the two siblings such that e(ℓ_{b1}) · e(ℓ_{b2}) is minimized over all the pairs
       of siblings in L(T)
8:     if e(ℓ_a)^2 ≤ 4 e(ℓ_{b1}) · e(ℓ_{b2}) then
9:         return T
10:    else
11:        Split ℓ_a in ℓ_{a1} and ℓ_{a2}          // Now ℓ_a is internal
12:        e(ℓ_{a1}) ← e(ℓ_a)/2
13:        e(ℓ_{a2}) ← e(ℓ_a)/2
14:        Merge ℓ_{b1} and ℓ_{b2} into ℓ_b     // Now ℓ_b is a leaf
15:        e(ℓ_b) ← e(ℓ_{b1}) + e(ℓ_{b2})
16:    end if
17: end while
```

Let us evaluate the difference between the variance on T and the variance on T'.

$$\sigma_T^2 = \frac{1}{m}\sum_{\ell\in L(T)}(e(\ell)-\mu_T)^2 = \frac{1}{m}\left(\sum_{\ell\in L(T)} e(\ell)^2 - \frac{1}{m}\left(\sum_{\ell\in L(T)} e(\ell)\right)^2\right).$$

Hence, we have

$$\begin{aligned}\sigma_T^2 - \sigma_{T'}^2 &= \frac{1}{m}\left(\sum_{\ell\in L(T)} e(\ell)^2 - \frac{1}{m}\left(\sum_{\ell\in L(T)} e(\ell)\right)^2\right) - \\ &\quad \frac{1}{m}\left(\sum_{\ell\in L(T')} e(\ell)^2 - \frac{1}{m}\left(\sum_{\ell\in L(T')} e(\ell)\right)^2\right).\end{aligned}$$

Since, by the operations executed in lines 12-13 and 15 of the algorithm, it holds that $\sum_{\ell\in L(T)} e(\ell) = \sum_{\ell\in L(T')} e(\ell)$, then, we have that:

$$\sigma_T^2 - \sigma_{T'}^2 = \frac{1}{m}\left(\frac{e(\ell_a)^2}{2} - 2e(\ell_{b_1})e(\ell_{b_2})\right)$$

Then, if $e(\ell_a)^2 > 4e(\ell_{b_1})e(\ell_{b_2})$, a split-merge operation can improve the variance of the times on the tree. The result follows by the observation that the variance is positive, by definition. □

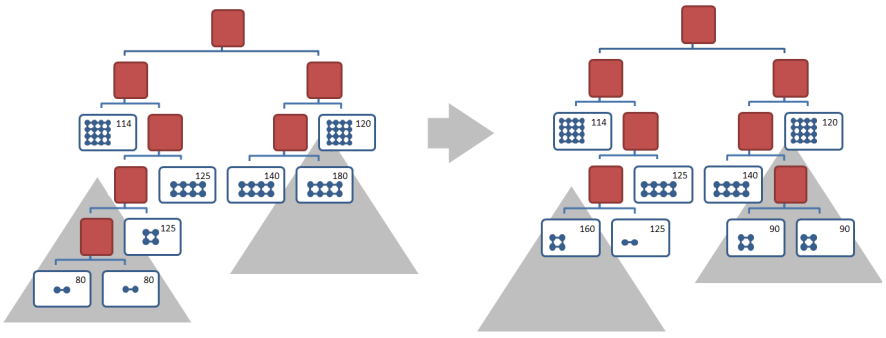


Figure 2.3: A merge and split operation on the PBT tree of Figure 2.2 where the estimation times $e(\ell)$ drive the updates.

Finally, it should be noticed that the improvement on the variance is proportional to $e(\ell_a)^2 - 4e(\ell_{b_1})e(\ell_{b_2})$. Then at each step, the greedy algorithm PBT-Update chooses ℓ_a and the siblings pair ℓ_{b_1} and ℓ_{b_2} (in lines 6-7) in order to have the higher (local) improvement in variance.

An example of a PBT is shown in Figure 2.2, and one of the updates of the PBT-Update algorithm is shown in Figure 2.3.

Exploiting Local Coherence In this paragraph we investigate how to leverage the PBT in order to better exploit data locality: the rationale behind this investigation is that jobs carried out by two siblings workers in the PBT, or by the same worker in consecutive computing phases, will probably follow similar memory access patterns.

In order to exploit locality we define the concept of *affine tiles*; in particular we will consider two kinds of affinity: processor-tile affinity and tile-tile affinity. A tile is affine to a processor if it has been assigned to that processor in the previous phase; two tiles are affine if they are neighbors in the mesh. When a worker asks for a tile the master node tries to assign it an affine tile. Then, the intent is to use affine tiles in order to exploit data-locality.

We implemented two heuristics in order to determinate affine tiles. The first heuristic is based on a greedy strategy, dubbed *PBT-Greedy*. For each computing phase, if a tile is not involved in a merge/split operation then it maintains his affinity with the processor it has been assigned to during previous phase. In case of tiles involved in a merge/split operation, let's consider a tile a split in tiles a_1 and a_2 and tiles b_1 and b_2 merged into tile b: a_1 is assigned to processor that handled a before; b is assigned to processor that handled b_2 before; a_2 is assigned to processor that handled b_1. Just this last assignment, on a total of 3 assignments, will, probably, not exploit cache and for this reason we say that this heuristic is 2/3 effective in leveraging locality.

In the second heuristic, named *PBT-Visit* the affinity is defined by visiting

the PBT: tiles are assigned following the in order visit. One can easily check that a subset of affine tiles, tiles "near" in the mesh, is assigned to the same processor phase-by-phase.

2.3 Case study: Parallel Ray Tracing

Ray Tracing algorithms [94, 96] are a widely used class of techniques for rendering images with the intent of achieving a high grade of realism. Ray Tracing is at the core of many global illumination algorithms. The input for Ray Tracing is a scene description that specifies the geometry of objects together with the definition of every object materials, position/orientation of the lights. The output is an image of the scene as seen through a virtual camera.

For sake of clarity we will shortly summarize the simplest form of Ray Tracing algorithm. For each pixel (x, y) in the final image a ray is casted from the virtual camera through the scene: this is called the *primary ray*. If an intersection between the primary ray and a surface is found then different parameters are considered in order to compute the light intensity at the point: the material of the surface, the position and the color of lights, whether or not the point is shadowed by other surfaces. In the Whitted-style Ray Tracing [106] a ray can be reflected and/or refracted according to surface properties and the process is repeated recursively with these new rays. At the end, the process adds the light intensities at all intersection points in order to get the final color of the pixel. Ray Tracing is considered a computationally intensive algorithm because it depends on the amount of rays shot throughout the scene and this amount can be easily increased by modifying lights properties, objects positions and materials.

Since its introduction several techniques have been explored to accelerate Ray Tracing. In animated scenes we report an interesting observation about the fact that a new frame can be very similar to the previous frame if the viewpoint did not change drastically. This similarity is an instance of the concept of *temporal coherence* (cft. Section 2.1) and can be exploited to reduce the amount of calculations needed for every new frame [18].

Let p be the pixel of generic coordinates (x, y) in frame f_i and let p' be the pixel with the same coordinates (x, y) (i.e. the same pixel) in frame f_{i+1}. Let r be the ray through p and r' the ray through pixel p'. The idea of the temporal coherence is based upon a simple consideration: the ray r and the ray r' will follow similar paths across the scene.

Introducing Ray Tracing algorithms

Ray Tracing is not just an algorithm, but instead a wide class of techniques used to generate photo-realistic images.

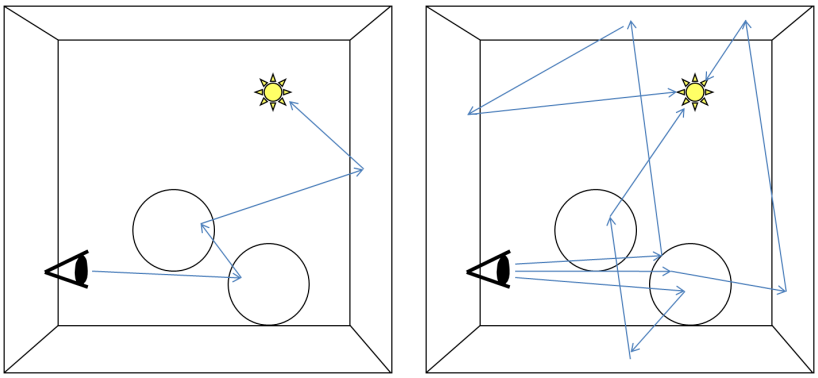
We briefly describe some known ray tracing flavor by using the *path notation*. A path represent a ray starting from the eye (E), traced through the path to the light source (L). Hence, each path is terminated by the eye and a light. Each bounce involves an interaction with a surface. We

characterize the interaction as either reflection or transmission. There are different types of reflection and transmission functions. At a high-level, we characterize them as

- D, diffuse reflection or transmission
- G, glossy reflection or transmission
- S, specular reflection or refraction

Diffuse implies that light is equally likely to be scattered in any direction. Specular implies that there is a single direction; that is, given an incoming direction there is a unique outgoing direction. Finally, glossy is somewhere in between. Particular ray tracing techniques may be characterized by the paths that they consider (using regular expression):

- Appel ray casting: $E(D|G)L$. It is a simple local illumination model that just calculate the first ray bounce from the eye, and traces a single light ray to the light source. Appel's algorithm traces only path of length 3, ignoring longer paths. Thus only direct lighting is considered.
- Whitted ray tracing: $E[S^*](D|G)L$. Specular reflection/refraction is recursively calculated for each single primary ray. It traces paths of any length, but all paths begin with a sequence of 0 or more mirror reflection and refraction steps. Whitted's algorithm ignores paths as $EDSDSL$ or $E(D|G)S^*L$, hence cannot simulate effect where there are multiple light bounces from a light source (e.g. caustic).
- Kajiya path tracing: $E[(D|G|S)^+(D|G)]L$. In respect of Whitted ray tracing, it traces glossy and diffuse reflection rays, hence is more compute expensive. Like others Monte Carlo methods, Kajiya path tracing uses a random walk to generate paths: we move from sample to sample, or from point to point, where the samples are drawn from a continuous probability distribution. Hence, the huge amount of rays generated by the technique is pruned by using a continuous probability distribution (*Russian roulette*).
- Radiosity: ED^*L. It recursively handles (Lambertian) diffuse reflections.
- An accurate method that correctly evaluate all path must handle the expression: $E(D|S|G)^*L$.



Whitted ray tracing Kajila path tracing

In this thesis we just focus on two ray tracing techniques: Whitted ray tracing, and Monte Carlo Kajila path tracing (used in Chapter 3). Of course, Kajila path tracing is computational more expensive than Whitted, but produces images of higher quality.

We do not discuss here the rendering equation used to calculate the incoming radiance for each ray. Philip Dutré wrote a good introduction to formulas and equations for global illumination algorithms [43, 44].

Parallel Ray Tracing The Ray Tracing algorithm has been defined *embarrassingly parallel* [48] because no particular effort is needed to segment an instance of the problem in tasks considering that there is no strict dependency between parallel tasks. Each task can be computed independently from every other task in order to achieve a speed up by executing them in parallel. There are two different approaches in designing a parallel ray tracer: object-based and screen-based [17]. In objects-based approach the scene is distributed among clients. For each ray casted the clients forward rays between clients. In the screen-based approach the scene is replicated on each client and the rendering of pixels is assigned to different clients. The second approach is the one investigated in this Chapter by a frame-to-frame load partitioning schema.

Speeding up Parallel Ray Tracing for interactive use on multi-processor machine has received a big impulse during last years, thanks to an efficient implementation designed to fit the capabilities of modern CPUs [8] and the use of commodity PC clusters [101]. In particular, several techniques are employed to amortize communication costs and manage load balancing. In [101] is suggested a task prefetching and work stealing, whereas [36] is presented a distributed load balancer.

2.3.1 Exploiting PBT for Parallel Ray Tracing

In order to exploit the PBT to accelerate Parallel Ray Tracing (PRT) algorithm we provide here a mapping between the concepts of the general case, introduced in previous sections, and the concepts strictly bounded to the Ray Tracing. The computation carried out in the PRT is the rendering of a sequence of frames. Every frame is rendered pixel by pixel; in terms of PBT acceleration each of these pixels is an item of the mesh. The memory buffer where each frame of the sequence is rendered can be considered a bidimensional mesh that represents an image: the PDIs managed by nodes are portions of this frame. The information that is available on every worker (ADI) and used to perform the assigned task is the scene description. The number of primitives, usually triangles, the dimension of the textures to be mapped on the geometry and the number of light sources are elements that increase the computational complexity of a scene to be rendered.

The master divides the frame buffer in tiles, which are rectangular areas of pixels. These tiles are assigned to workers to be rendered. Since two rays will follow similar path if they are close, in order to make an effective usage of the local cache for each node, it is important that the tiles are contiguous and large enough, so that each worker can exploit spatial coherence of tiles, having a good degree of (local) cache hits. Another task performed by the master is to handle the frame buffer for both visualization or to save it into a file.

The granularity of our decomposition strategy is chosen defining $m = k \cdot n$ where m is the number of tiles, n is the number of workers and k is multiplicative constant. The greater is k, the smaller is tiles sizes. We experimentally tested several values of k and found that no large k is needed since, after small values of k, performances degrades due to the higher communication cost (cf. Section 2.4.2).

2.4 Experiments and Results

Our serial implementation of Ray Tracing algorithm exploits some, but not all, optimizations techniques used by last cutting-edge ray tracers. Actually, the kind of serial implementation that is used is not relevant for our purposes. Special attention is paid to the acceleration structure. We use a Kd-tree built to minimize the number of traversal and intersection steps, done using the well-know Surface Area Heuristic [104]. Our Kd-tree implementation also provides a fast Kd-tree traversal by using a cache friendly data-layout [99].

We implemented a synchronous render system, with a synchronization barrier at the end of each frame for visualization and camera update purpose. Furthermore, we adopt a demand driven task management strategy where a task manager maintains a pool of already constituted tasks. On receipt of a request from a worker, the task manager dispatches the next available task from the pool (for $k > 1$). We also added a threshold to the number of single merge/split updates into the PBT-Update algorithm in order to avoid to perform many

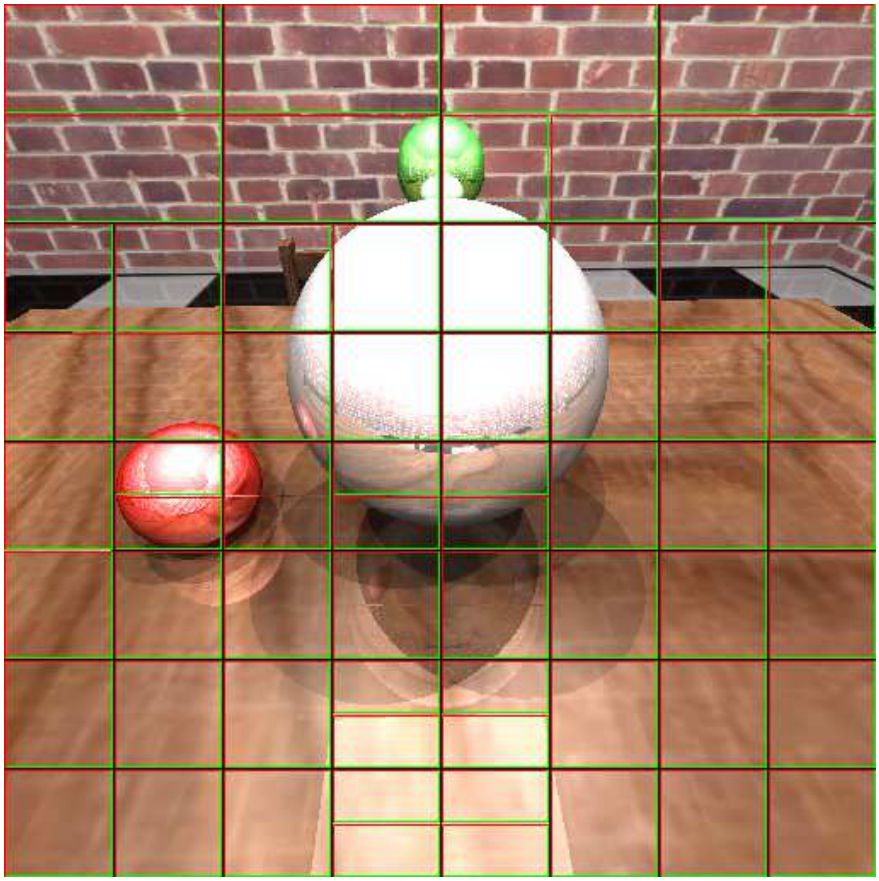


Figure 2.4: A frame in the walk-trough for scene ERW6-4, with the tiling shown.

small changes to the tree that would not affect much the overall performances.

We coded our system in C++, compiling it with Intel C++ Compiler 10.1 for Linux. We used MPI [69] for node communications, having care to disable Nagle algorithm [74] in order to decrease latency.

2.4.1 Setting of the experiments

Because the aim of this work is to exploit temporal coherence in load balancing, we decided to use a distributed memory system, as a cluster of workstations, and test scenes with remarkable unbalance between tiles. We ran several tests on three hardware platforms:

Test platform

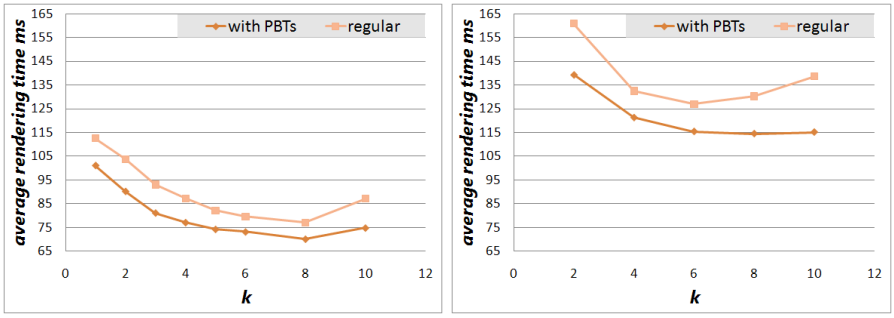


Figure 2.5: Average per frame rendering time on increasing k, comparing regular and PBT-based job assignments. (Left: ERW6. Right: ERW6-4)

List of test plaforms used in this Chapter. Appendix A contains additional details for each single platform.

Hydra IBM BladeCenter Cluster of 33 nodes (1 master node, 32 worker nodes) ⟼ details on Appendix A.4

Cacau a NEC Xeon EM64T Cluster of 64 nodes ⟼ details on Appendix A.1

ENEA Cresco Sezione 2 an IBM HS21 Cluster with 256 nodes ⟼ details on Appendix A.2

We tested our scheme on two scenes, each of them with different shading aspects. Since the focus is on manage unbalancing, we used a modified standard ERW6 test scene (see Figure 2.4) (about one thousand primitives in total). Unbalancing is due to the surface shading properties used in the scene. We developed two versions of this test scene: ERW6, has one point light source; ERW6-4 has four light sources. The more light sources are present in the scene the bigger is unbalancing because of the increased number of rays to be shot. In both test scenes, we have a predefined walk-through of the camera around the scene, with movements in all directions and rotations too. The image resolution is 512×512 pixels, unless differently stated.

2.4.2 Results

We performed several test in order to evaluate and estimate: the effectiveness of the PBT, its scalability, the optimal tiles granularity for different number of processors, the impact of temporal and spatial coherence, how the resolution affects the scheduling technique, the overhead incurred by the calculations for the PBT on the total computing time.

In some tests, we compared the technique that uses PBT with a *regular subdivision* technique of equal-sized tiles.

Effectiveness of Prediction Binary Tree In these tests, ran on Hydra, we evaluate the improvement provided by using the PBT instead of a regular subdivision strategy.

The results are shown in Figure 2.5 for both scenes. Results are obtained using different granularity and $n = 32$ workers. Our technique offers a speedup, ranging from 5 to 15 percent, for all the values of k tested. When k is large, the performances degrade due to two factors: the number of updates on the PBT increases. Our test shows that there are few updates for smaller values of k, but they grow quickly as k increases. The second is related to the heuristic that we have chosen. Indeed, measuring the rendering time for tiny tile has some approximation problems due to discretization. Our algorithm gives good performances for small values of m. For big values of m, the decomposition algorithm may be a bottleneck.

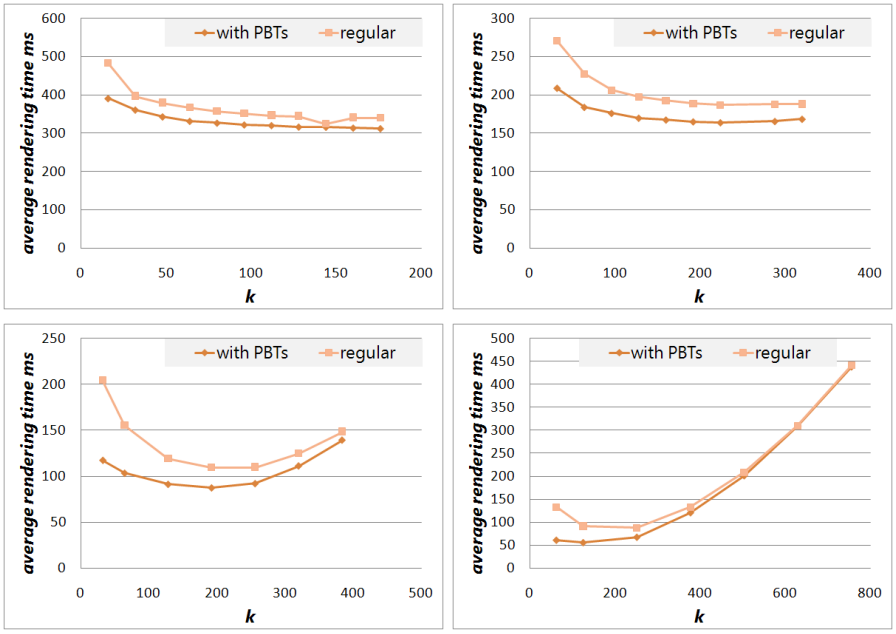


Figure 2.6: Optimal subdivision granularity with both regular and PBT subdivision. Test done with 8 (top-left), 16 (top-right), 32 (bottom-left) and 64 (bottom-right) processors. The horizontal axis represents the number of tiles, whereas vertical axis represents rendering time. ERW6-4 test scene.

Optimal Granularity We tried to estimate the optimal granularity of the schema, i.e., the optimal choice of the number of tiles m, with 8, 16, 32 and 64

processors (see Figure 2.6). The rationale behind this test is to determinate how our schema affects the tuning of the granularity in the parallel implementation, compared with the regular subdivision schema.

Test results, performed on Cacau, raise several interesting considerations. The optimal granularity for the PBT comes with a lower number of tiles, in respect of the normal regular approach. In particular the PBT works better with coarser tiles, introducing beneficial effects because of the lower overhead of communication.

Scalability We compared our schema based on the PBT against the regular subdivision schema with 2, 4, 8, 16, 32 and 64 processors, in order to evaluate the efficiency of our strategy (see Figure 2.7). The tests, ran on ENEA, shows that our schema works always better than the regular one, and presents almost linear scalability. In all the tests the granularity coefficient k is fixed to 4. However, the test with 64 processors also shows that when the number of tiles increases the use of an adaptive subdivision is still not enough in order to assure a good scalability. We conjecture that, in order to improve scalability, the value of k should be tuned in such a way that tile's sizes do not become extremely small.

Temporal coherence tests In order to explore the performances of the PBT in exploiting temporal coherence, we checked it under different conditions. We ran a set of tests on Hydra in order to evaluate the correctness of the prediction made using the PBT. First we tested two different task granularity (we recall that we consider $m = k \cdot n$, where m is the number of tiles and n is the number of worker): we have chosen $k = 1$ (one tiles for each worker) and $k = 4$ (four tiles, on average, for each workers). Moreover, we considered two different camera speeds (1x and 2x). In all the tests the number of workers n is 32. To make a comparison, we measured the total amount of tiles which has been estimated correctly using 85^{th}, 90^{th} and 95^{th} percentile (see Table 2.1). As an example row 2 (Perc. 90^{th}) represents the percentile of estimations having an error up to 10%. In other words, when $k = 1$ and the camera speed is 1x the 93.2% of estimations have an error smaller than 10%, while when $k = 4$ and the camera speed is 2x the 79.8% of estimations have an error smaller than 10%.

Corr. Perc.	$k = 1$ **1x**	$k = 1$ **2x**	$k = 4$ **1x**	$k = 4$ **2x**
85	96.2%	95.3%	92.1%	89.7%
90	93.2%	92%	86.2%	79.8%
95	92.6%	84%	68%	55%

Table 2.1: Results of the predictions in 85^{th}, 90^{th} and 95^{th} percentile.

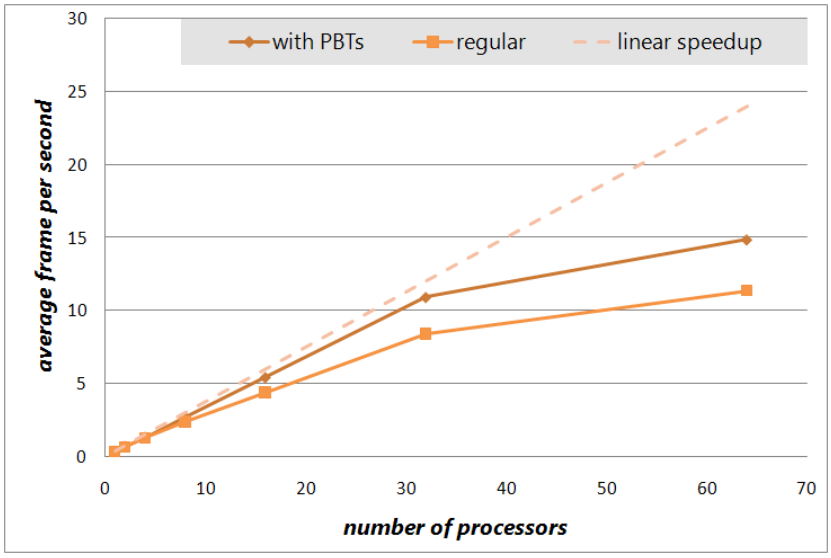


Figure 2.7: Scalability. Frame rates on increasing number of processors comparing the regular subdivision, our PBT-based subdivision and the optimal linear speedup. ERW6-4 test scene.

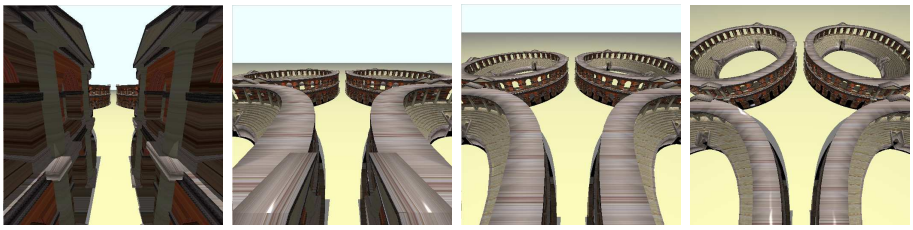


Figure 2.8: Amphitheater test scene, used in spatial coherence test set.

Spatial coherence tests In this paragraph we investigate how the PBT exploits data locality using the two locality-aware heuristic described in Section 2.2.1. The idea is to let workers to better use their CPU cache.

The test considers 4 different scenes with an increasing number of triangles, ranging from less than 30000 to about 950000. In Table 2.2 are reported the number of triangles and the number of nodes in the Kd-tree. The difference between scenes is an increasing number of amphitheaters, all of them with a simple diffusive shader (see Figure 2.8). Scenes are animated by a predefined walk-through of the camera.

The rationale behind the test is to verify that, whenever the size of the Kd-tree is bigger than the cache size, a drop of the performance can be measured, due to the number of cache misses.

Test scene	Triangles	Nodes in Kd-tree
1 Amphitheater	29759	251017
4 Amphitheaters	119026	999237
16 Amphitheaters	476098	3970097
32 Amphitheaters	952130	7970097

Table 2.2: Description of the test scenes used for spatial coherence tests. For each scene, we report the corresponding number of primitives and the Kd-tree's size.

All tests shows that improvements obtained by using the PBT (with both the *locality-aware* heuristics and a random assignment named *PBT-Random*) with respect to the regular assignment (cf. *Exploiting Local Coherence*, Subsection 2.2.1). With more details, on small scenes, the whole scene fits into the cache and then the assignment strategy does not matter. When the scene becomes larger, so that it does not fit into the cache, the data locality also provides a modest improvement of the performances of the system (around $1-5$ ms) using both the *locality-aware* heuristics. This test has been performed on Hydra.

Impact of Resolution In Parallel Ray Tracing the resolution represents the total amount of work. We ran a set of tests on ENEA, with a fixed task granularity ($k = 4$), using different resolutions in order to show that our implementation of PRT scales linearly with the number of the primary rays. In order to have a measure of the effectiveness of our technique with higher workloads, we tested the PBT with three well-known resolutions. In particular we compared both techniques (PBT and regular subdivision) with PAL (720×576), HD720 (1280×720) and HD1080 (1920×1080) resolutions on the same test scene (ERW6-4).

The tests show that the PBT is effective with all the workloads and its ability in balancing the work between nodes is beneficial with heavy loads (see Figure 2.9).

Total time analysis The last test is focused on how the whole computing time (i.e., the parallel rendering time) is spent.

The time spent by the master node in computing out PBT-based subdivision schema from a given prediction is serial code and pure overhead introduced by our approach. This overhead corresponds to the time spent in subdividing the image in tiles and updating the PBT.

Our purpose is to determinate: the ratio between time spent by the workers in local rendering and communications; how unbalancing affects performance; how much time is spent by the master node in serial code.

Figure 2.10 shows the test results for 16 and 32 processors at different granularity. The time spent in updating the PBT is proportional to m, hence to the granularity and the number of processors, but in our test is always lower than

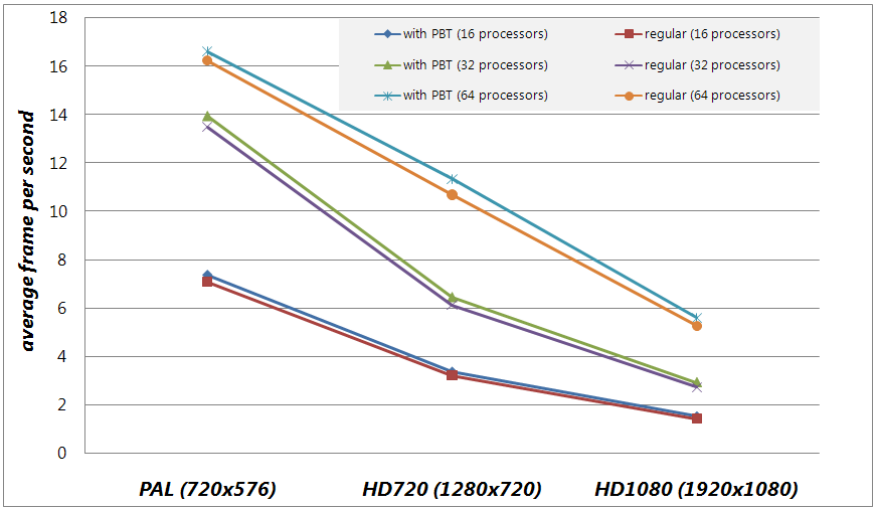


Figure 2.9: Average frame rate using different resolutions: PAL(720×576), HD720(1280×720), HD1080(1920×1080), comparing regular and PBT-based job assignments. ERW6-4 test scene.

5 ms. Thus the overhead by the adaptive subdivision is small compared with the gain in balancing. This test has been performed on Cacau.

2.5 Conclusion

In this Chapter we presented a scheduling strategy based on a data structure called PBT. To assess the effectiveness of the proposed scheduling strategy we carried out some experiments, that provided a large amount of results. Our scheduling strategy: (i) improves load balancing; (ii) allows to exploit temporal coherence among successive computation phases; (iii) minimizes the inter-processors dependency. By some assumptions on temporal coherence, we showed that an estimate of next phase workload can be used to quickly divide the mesh in almost-squared tiles assigned to each worker. PBT is effectively used to evaluate the load balance of each phase and, eventually, to update tasks assignment in order to reduce their completion time.

We tested our strategy on a significant problem: Parallel Ray Tracing. We carried out an extensive set of experiments where our PBT-based strategy is compared against the regular subdivision schema. We showed that by using our technique, the optimal granularity comes with a lower number of tiles. Moreover, the PBT approach assures a better scalability.

The predictions used in our approach are based on temporal coherence. We proved that for simple scenes (e.g. a predefined walk-through of the camera

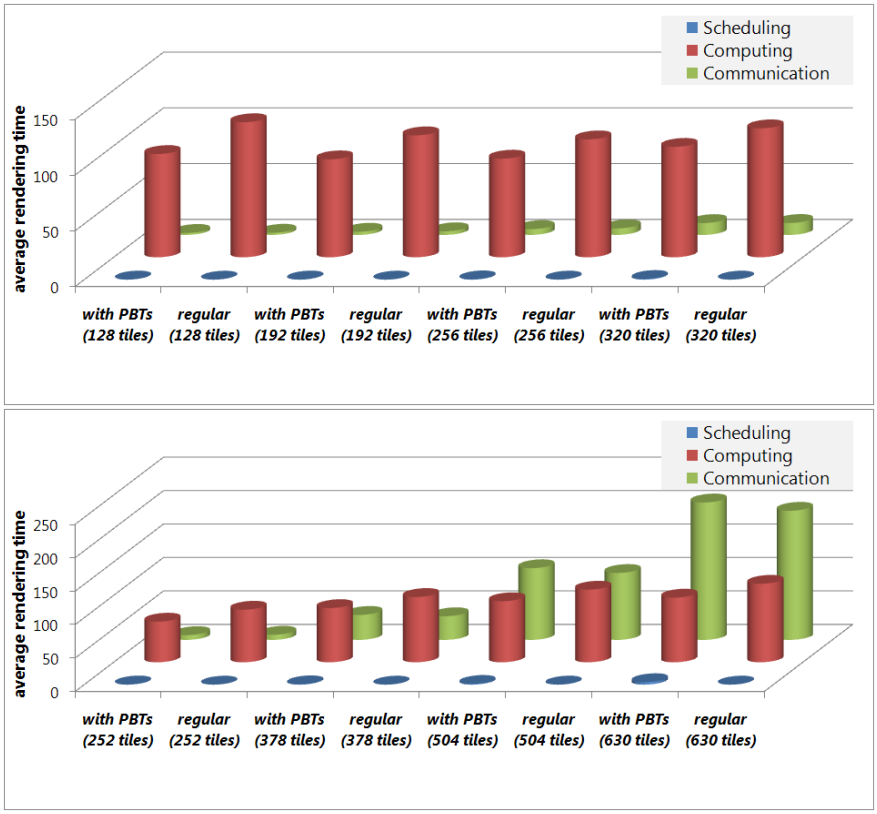


Figure 2.10: Contributions in rendering time with 32 (up) and 64 (down) processors, at different granularities, with both adaptive and regular subdivision techniques. The total rendering time is split in: the time spent in subdivision (i.e. update the PBT for adaptive subdivision); the maximum per-node compute time, such as an estimate of the load balancing; the remaining time, mostly due by communications. Rendering times shown are the average in a test scene (ERW6-4) of 600 frames.

around the scene), the proposed estimation heuristic is affordable. Spatial coherence for data locality has been exploited by using two locality-aware heuristic during assignment. However we showed that such heuristic provides an improvement on performance only with larger scene (at least 1 million of triangles).

Tests with higher resolutions show the ability of the PBT in balancing the work with heavy loads.

Finally we provided a total time analysis of the computing time and the overhead introduced by the PBT. The time spent in updating the PBT is proportional to the number of tiles, but in our test is always lower than 5 ms. Indeed, the overhead of the adaptive subdivision is small compared with the

gain in balancing.

The variety of architectures used on our tests suggests that the technique improves performances in both cheaper commodity cluster and high performance clusters with low latency networks.

Acknowledgments

A portion of this work was carried out under the HPC-EUROPA++ project (project number: 211437), with the support of the European Community - Research Infrastructure Action of the FP7.

We gratefully thank for their collaboration in providing some of the computational resources the ENEA (Ente per le Nuove Tecnologie, l'Energia e l'Ambiente) – Research Center in Portici (Napoli, Italy) and the HLRS Supercomputing Center at Universität Stuttgart (Germany).

3

Load Balancing based on Cost Evaluation

If we known in advance a cost estimate of a task, we could take smarter subdivision decisions in order to improve load balancing. The Prediction Binary Tree showed in Chapter 2 is an example of such use of a cost evaluation by using a centralized load balancing schema. However, the test case proposed (Parallel Ray Tracing) can be improved in two ways: (1) by taking advantage of a faster *serial* ray tracing implementation; (2) by enhancing the proposed parallelization (e.g. shifting from a centralized balancing technique to a distributed one).

Ray tracing algorithms have seen enormous progress in recent years. Highly optimized packet-based ray tracing implementations allow the computation of millions of ray-triangle intersections per second, and fully exploit modern multi-core CPUs or GPUs. Anyway, complex scenes and lighting, and high quality renderings with anti-aliasing are still not feasible at interactive speed, and only possible when using compute clusters. As we showed in Chapter 2, good load balancing is crucial in order to exploit the computational power, and not to suffer from communication overhead and synchronization barriers.

In this Chapter, we resume the Parallel Ray Tracing problem: Instead of use it as a test scenario for our proposed balancing strategies, we begin from a state of the art implementation of ray tracing[1], thus we parallelize it being aware of the underlying hardware architecture. Our implementation successfully exploit: SIMD vectorial parallelism, by using ray-packets (i.e. the algorithm traces more rays at once, instead of a single one); multi-core parallelism, by scheduling ray-packets among multiple threads; multi-computer parallelism, by assigning tile (i.e. set of pixel, further subdivided in ray-packets) to remote compute-node in a cluster of workstation. In addition, GPU availability at the visualization node is exploited using *ad hoc* techniques. For these reasons, implementation used here outperforms to one used in Chapter 2[2].

[1] Implementation used in Chapter 2 is different from the one developed here

[2] Important differences: SIMD vectorial instructions have been used as a wrapper for color and vector, resulting in a less effective implementation compared to ray-packets; MPI parallelization was on-demand and centralized, did not use pooling neither asynchronous prediction techniques; the only rendering technique supported was Whitted ray tracing

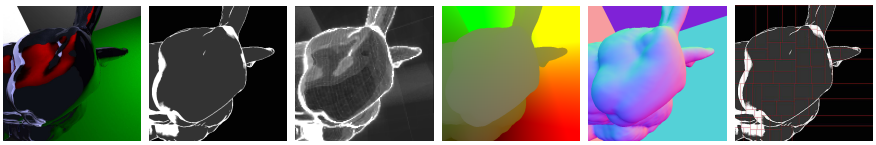


Figure 3.1: Left to right: the ray-traced image, the GPU-based estimation of the rendering cost, the actual packet-based rendering cost, position and normals in eye-space (used for the cost estimation), and using our method for an adaptive tiling of the image for parallel rendering.

We present a method that uses a cheap GPU rendering technique to compute a cost map: an estimation of the per-pixel cost when rendering the image using ray tracing. Using this information, we improve load balancing, task scheduling, and work stealing strategies.

3.1 Introduction

Ray tracing algorithms [51] model physical light transport by shooting rays into the scene with the ultimate goal of producing photorealistic images. Considerable efforts have been made in order to investigate new ways to reduce the high computational demands of ray tracing. In particular, recent advances in ray tracing include exploiting coherence between neighboring pixels with packet traversals [100], frustum traversal [88], and fast updating of the acceleration data structure for animated scenes [114]. Thanks to the recent advances in both software and hardware, Whitted-style ray tracing reaches interactive frame rates on single CPUs [77] and GPUs [66]. However, if we want to provide more realism in the produced images, e.g. by computing global illumination, we need to drastically increase the number of secondary rays.

Even though several optimizations strategies [88] allow a certain amount of interactivity on static scenes, the use of complex shading, shadows, and reflections requiring a large number of secondary rays, increases the computation cost. Indeed, the number of rays grows from less than one million of typically coherent rays, up to several millions of mostly incoherent rays. In this context, distributing ray tracing among several workers is the only solution to reach interactive frame rates.

In this Chapter, we describe a parallel ray tracing system exploiting balancing and distributing rendering tasks across workers in a network. We use efficient GPU-techniques for an efficient estimate of the per-pixel ray tracing cost, and then use this information for work stealing, in-frame steals, and dynamic load balancing algorithms.

3.2 Previous Work

Ray tracing parallelization approaches can be classified into two main categories: *image space parallel decomposition*, and *scene geometry parallel decomposition* (SGPD), sometime referred as *data parallel* [1]. With *image parallel* methods, each worker is responsible for a region of the image (e.g. a pixel or a tile), while the scene data is replicated in the memory of each node. In contrary, in *SGPD* methods, each worker is responsible for a part of the scene, while rays propagate among the nodes. Image space parallelism is typically the better solution for scenes that can be stored in a single node.

For complex models, where the scene exceeds the capability of a single worker node, the scene data has to be distributed among all nodes. However, additional efforts are necessary to manage the scene data and to minimize the impact of remote data requests with caching and prefetching strategies. Several intermediate solutions can be obtained by merging the two models; see for example [86].

Shared memory systems Early works in parallel ray tracing reaching interactive performance used massively parallel shared memory supercomputers, e.g. [72]. Current hardware trends in processor designs are turning towards multi-core architectures, and parallel ray tracing is well suited for off-the-shelf hardware. Manta [8] is an interactive ray tracing system combining a high level of parallelism with modern packet-based acceleration structures. It uses a multi-threaded scalable parallel pipeline in order to exploit parallelism on multi-core processors. Manta has been successfully used in the context of massive model interaction, providing the capability of easily visualizing huge aircraft models in an inspection and maintenance scenario [95]. Recently, more systems have been developed focusing on exploiting the massive parallelism of multi-core hardware; for example [49].

Distributed memory systems Developing interactive ray tracing systems for distributed memory systems is an intricate process. Extending a renderer's architecture to a cluster of workstations requires implementing several components, such as a high-performance communication layer and an efficient dynamic load balancer. Without these techniques, the overhead of the communication causes poor scalability and performance penalties. Commodity-based clusters offer a cost-effective solution to speed up ray tracing and are becoming more widely available.

In [105] the authors used coherent ray tracing techniques in distributed memory architectures. In particular, they exploit spatial coherence in the acceleration data structure, and temporal coherence between subsequent frames. Their system design was centralized, but they were able to render large and complex models at interactive rates by using a two level BSP for per-node caching of geometry. Later, several works improved these techniques in order to render massively complex models, e.g. [103] and [41].

DeMarle et al. [35] presented a distributed interactive ray tracing system

using page-based distributed shared memory. Each node of the cluster manages different parts of the scene data and reserves space to cache remote parts. The hybrid parallel design of their system allows the rendering of large data sets quickly, even if a single node's memory can store a fraction of the total data only.

Load balancing is one of the challenges of parallel ray tracing. In particular, if a rendering system is subject to a barrier synchronization point (e.g., in synchronous rendering), then the slowest task will determine the overall performance. Achieving a good load balance is not trivial and can be achieved using two main strategies: trying to equally partition tasks, or using dynamic work assignment. In this context, choosing the *granularity* of the subdivision (i.e. the number of tiles of the image distributed among all nodes) is non-trivial. For example, a fine granularity facilitates load balancing, but it also results in high communication overhead, which is critical in slow networks.

Researchers have proposed many strategies for addressing load balancing in this context. Heirich and Arvo [54] discussed the importance of dynamic load balancing for ray tracing in interactive settings. They also showed that strategies based on image tiling and predictions are ineffective when used with static load balancing.

Further related work examines the importance of the subdivision granularity [83], and suggests adaptive subdivision to balance the workload [29].

In demand driven approaches, the image is usually subdivided in tiles of fixed size. Typically, the rendering times per tile vary, thus a simple static load balancing scheme is not suitable and a dynamic assignment is required instead. Although a demand driven centralized balancing scheme achieves a well balanced workload, it involves significant master-to-worker communication which becomes a bottleneck network network transmission delay and the number of workers increases. A decentralized load balancing scheme, such as work stealing [11] or work redistribution, eliminates the communication bottleneck thus improving performance and scalability. For instance, DeMarle et al. [38] improved performance by moving from a centralized demand driven load balancing scheme to a decentralized scheme based on work stealing. Later in [37], work stealing has also been used to balance work between two different frames: once a worker finishes its assignments for a given frame, it picks a node at random and requests more work form it. That node only responds, if it has work available to share. In their implementation, task migration is done at the beginning of the next frame (frame-to-frame steals), and the synchronization bottleneck at the master node is hidden by an asynchronous task assignment.

3.3 Overview

Our method described in this chapter follows the paradigm of image-based parallelization, tailored for a cluster of workstations, and scenes that can be stored on a single node. It uses ray-packets, is synchronous, and introduces several heuristics in order to better distribute work between nodes.

The target hardware is a cluster of workstations, where a master node, equipped with a GPU, is also responsible for displaying the solution, and several workers, equipped with multi-core CPU, perform ray tracing computation.

Our rendering system is based on a packet-based ray tracing implementation, and considers a *task* as the rendering of a *tile of an image*. Our parallel rendering architecture uses GPU-techniques to estimate the ray tracing cost and by this improve dynamic load balancing strategies. This yields good load balancing while maintaining a coarse granularity, and thus allows for high rendering performance even with slow networks.

Our method performs the following steps to balance the rendering load, which we discuss in detail in the following sections:

1. Compute a per-pixel, image-based estimate of the rendering cost, called *cost map*.

2. Use the cost map for subdivision and/or scheduling in order to balance the load between workers.

3. A dynamic load balancing scheme improves balancing after the initial tile assignment.

3.4 The Cost Map

In this section we show how to obtain the cost map, i.e. an image-based, per-pixel cost estimate of the rendering process (see Fig. 3.1 and 3.2). First, we define the problem statement and related approaches. Then, we introduce a GPU technique capable quickly compute an approximate cost map. Finally, we analyze the cost estimate error.

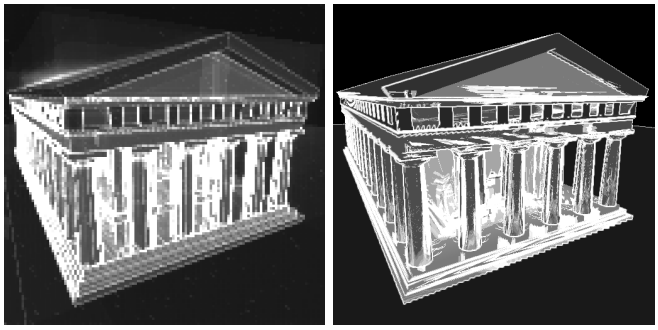


Figure 3.2: A comparison of the real cost (left) and our GPU-based cost estimate (right).

3.4.1 Rendering Cost Evaluation

Sources of load unbalance Several factors affect the rendering cost in ray tracing, such as scene size, resolution, rendering technique, coherence between rays, and material properties. In particular, shading algorithms affect the number of secondary rays traced into the scene, and cause load unbalance among different pixels in the image (Figure 3.3).

Our work focus on calculating an image-based cost estimation by using the information that is available in a geometry buffer rendered on a GPU.

Cost evaluation approaches Before introducing our technique, we briefly discuss some techniques known in literature. In [50] the authors suggest that an approximated cost map can be generated from a rasterized scene preview. This technique has been used combined with a profiling strategy in order to achieve an accurate cost prediction. They made a comparison of different pixel profiling strategies which may be used to predict the overall rendering cost of a high fidelity global illumination solution.

Albeit samples measuring and profiling techniques have been successfully exploited in a global illumination scenario, they fit badly our rendering system. In a distributed memory architecture, profiling may be performed on the master rather than the workers. However, in the latter case, workers need to send back sample measures. This drastically increase the time spent to compute load balancing calculation (at least, if we suppose to be performed centrally by the master node). Furthermore, recent advances in ray tracing (i.e. packet traversal) assume that rays are coherent. This means that usually, ray tracing of few incoherent rays is slow. For these reasons, despite this approach performs well in a global illumination scenario, it is not suitable for distributed architecture aiming at interactive visualization.

Beyond image-space approaches, rendering cost estimation has been explored even for SGPD. For example, Reinhard et al. [87] used a voxel-based cost estimate to efficiently distribute the scene across processors.

3.4.2 A GPU-based Cost Map

We propose a fast method to compute an approximate cost map by only using the GPU. We combine well-known GPU techniques like geometry buffers from deferred shading [53] and image-space sampling to accomplish our goal. The underlying idea is that it is often possible to detect potentially expensive areas, e.g. with multiple inter-reflections, performing a fast image-space search.

Algorithm description Our algorithm (Algorithm 2) works in image-space, and assume that for each pixel, information on material properties are available. Every pixel P_i has a certain basic cost depending on its material (e.g. the cost for evaluatin the BRDF model).

In addition to this basic shading cost (for every surface point P_i), we perform an image-space search if the surface at P_i is reflective. The sampling step

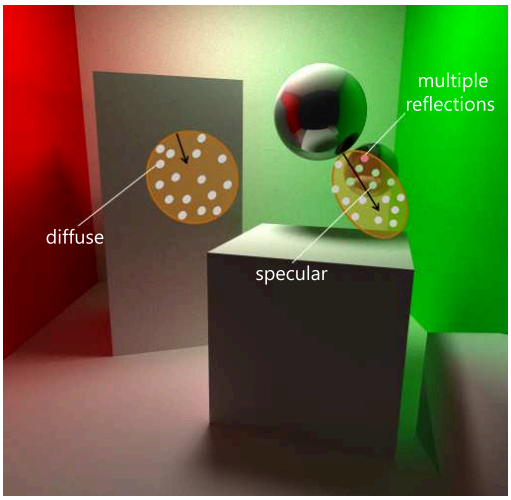


Figure 3.3: Estimating the rendering cost: diffuse surfaces typically have lower cost (in particular for Whitted-style ray tracing), while specular surfaces generate more secondary rays thus causing higher rendering cost. Regions where multiple reflections occur are typically more expensive and can be found by a search in image-space. When using Monte Carlo-based techniques, the image-space search can be adapted according to the specular coefficient of the surface (see the sphere in the images).

is performed in order to detect expensive areas, e. g. with multiple intereflections. For this, we use a uniformly distributed set of sampling points which is transformed before sampling. The sampling pattern is computed according to shading properties, rendering technique and the reflection vector R. The pattern is scaled to become more narrow for surfaces with higher Phong exponents, thus positioned at the surface point P_i in question and oriented along the projection of its reflection vector (Figure 3.3).

For every sample S_j, we retrieve its surface location and orientation from the geometry buffer. Next, for each sample we perform a test in its surroundings to detect the sample cost. We test if P_i and S_j are mutually front-facing (line 11). Later, a test detects if the sampled surface is reflective (line 13). If the surface at S is reflective as well, and the specular reflection (e.g. the Phong lobe) of S_j points towards P_i then we detected a region with a potentially high number of inter-reflections, and further increase the cost estimate. Afterward all the samples contributions are gathered and the resulting cost is calculated.

Sampling pattern The sampling pattern is randomly generated (a) and scaled (b) according to the surface properties. Thus, the pattern is translated (c) and rotated (d) towards the R vector. Step (b) depends by shading material and rendering technique. In particular, we distinguish a wide sampling pattern for path tracing and Lambertian materials. Instead, the pattern collapse to a line for Whitted-style ray tracing and for path tracing using a specular material. In fact, the wideness of the sampling lobe depend by the spreading of the secondary rays, i.e. the Phong specular coefficient. The sampling is performed once per pixel, not recursively.

Sample gathering Once sampling has been performed, we gather the contribution from all samples. This step is somewhat correlated to the rendering

Algorithm 2 Approximated cost map computation algorithm. Code for pixel P_i

```
1:  //All the data of the hit surface on the pixel P_i are available
2:  cost_i ← basic material cost of the hit surface
3:  if P_i is reflective then
4:     // Determinate the sampling pattern toward the reflection vector R
5:     S = compute_sampling_pattern(P_i, R)
6:     // Calculate cost contribution, for each samples
7:     for each sample S_j in S do
8:        sample_j ← 0
9:        if visibility_check(S_j, P_i) then
10:          increase sample_j
11:          if secondary_reflection_check(S_j, P_i) then
12:             increase sample_j
13:          end if
14:       end if
15:    end for
16:    // Samples gathering
17:    cost_i = cost_i + gather(S)
18: end if
19: return cost_i
```

technique. In particular, the cost may be computed in two ways: by summing up the sample contribution, or by taking the maximum. The first approach is used in path tracing, where we suppose that secondary rays spread along a wide area. In the contrary, using Whitted ray tracing, all samples belong to one secondary ray and we consecutively estimate the cost by taking the maximum.

Edge detection Whenever packet-based ray tracing is used, packet splitting raises the cost because of the lose of coherence between rays. This occurs at depth discontinuities that we detect using a simple edge-detection filter on the geometry buffer. We experienced that this extra-cost is relatively remarkable only when using Whitted ray tracing. By further increasing the cost estimate at edges, we account for the impact of packet splitting.

Implementation details The algorithm has been implemented in a two pass shader. In a first pass, we render the scene to a geometry buffer, using multiple render targets to store data. For every pixel we store the position, the normal, and a value indicating if the surface is reflective, for the first visible surface seen from the camera (Figure 3.3). We also store three additional values: The basic shader cost, the specular coefficient and the Phong exponent. In the second pass, we generate the cost map using the image-space information stored in the geometry buffer. The basic value has been used in several points of the algorithm (i.e. lines 2 and 12). The other values both contribute in the sampling phase

(line 5 and Figure 3.4). Details about cost map computation parameters are shown in Table 3.9.

The cost map generation is fast: Whereas both CPU and GPU computations take less than 1 ms, the most expensive task is the data transfer between GPU and CPU (Table 3.1). However, this time is spent by the master node in barrier, i.e. when preparing the cost map before distributing the work load. For that reason, this time is further hidden using an asynchronous prediction optimization (see Section 3.6.5).

Task	Time in ms
cost map generation	< 1
SAT computation	< 1
SAT transfer	5-6
CPU tasks (i.e. sorting or adaptive tiling)	< 1
Total	6

Table 3.1: Cost map computation timings. They are similar for all test scenes.

3.4.3 Cost Estimation Error Analysis

Obviously, the resulting cost map is approximate. We analyze the error in order to understand where and why the estimate is inaccurate. First, we measure a per-packet difference between real and estimated cost. Second, we analyze a difference maps between real and estimated cost map, showing where the estimate is less accurate. Later in Section 3.7, we measure performance of the whole system, evaluating how much the balancing techniques described in this Chapter are sensitive to the cost map accuracy.

Limitation of the approach Our technique produces a good estimate in all our test scenes, for almost all view points (Figure 3.13). However, the above

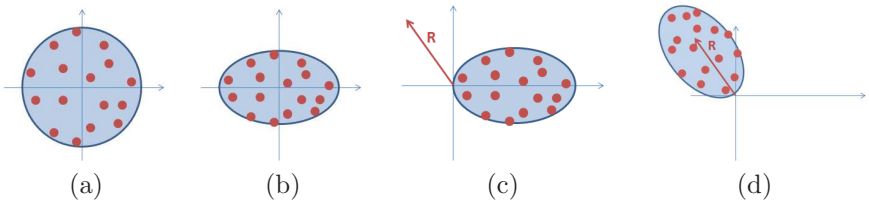
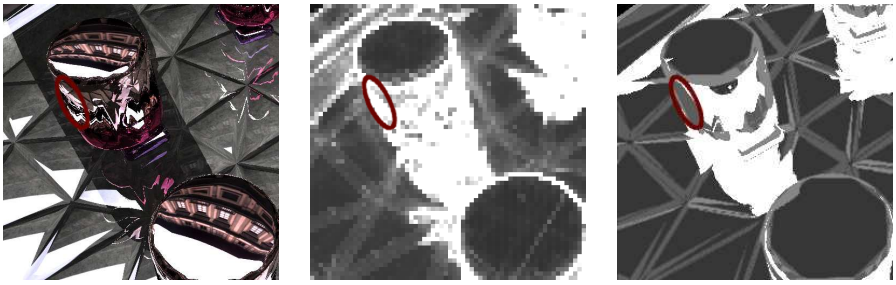


Figure 3.4: Sampling pattern. The sampling patterns used in the cost map generation algorithm. At first, an uniformly distributed set of points is generated (a). According to the shading properties, the pattern is scaled (b) and translated to the origin (c). In the last step, it is transformed according to the projection of the reflection vector in the image plane(d).

algorithm leaves one issue unaddressed: the *off-screen geometry* problem. Because our strategy works on a rasterized fast scene preview, our algorithm only works on visible geometry. Starting from this, our algorithm try its best working on image space, estimating the cost of each pixel. Nevertheless, reflected rays may fall in geometry not present in the frame buffer. When this happens, the algorithm is not able to detect a reflected surface and the resulting calculated cost is under-estimated in respect to the real one. Figure 3.5 shows an explanatory example.



(a) (b) (c)

Figure 3.5: Off-screen geometry problem. We show a particular where the problem is emphasized: The image indicates an area where secondary rays fall outside geometry in the rendering buffer (a), hence raising the cost of these pixels (b). Our GPU technique under-estimates the cost of this area (c).

Some advanced rendering techniques may partially settle this problem. For instance, a trivial solution consist in rendering the cost map in an extended image plane in order to compute surfaces outside the current view. The use of the A-buffer guarantees that the whole geometry is available on the render buffer [15]. Further errors may arise when sampling phase fails to detect high cost regions, for instance because the number of samples is not sufficient. Nevertheless, in this Chapter we do not investigate other intricate techniques, focusing on the effectiveness of the proposed one.

Error distribution Figure 3.6 shows the error distribution from 4 test scene, each of which shows the (packet-based) difference between the real computing time and the GPU computed one. The analysis shows that the estimate is accurate, and errors usually lead to a positive difference (hence we have an *under-estimation*, caused by reasons discussed above). The Cornell box scene is the more accurate one. Instead, the Ekklesiasterion is the one with less accurate in estimation.

Figure 3.13 shows real cost map, GPU calculated cost map, and *difference map* between real and approximate cost map. Figure 3.12 shows final renderings.

Error impact in our framework As we will show later in this Chapter, we use the cost estimation in two different ways. We anticipate that, while

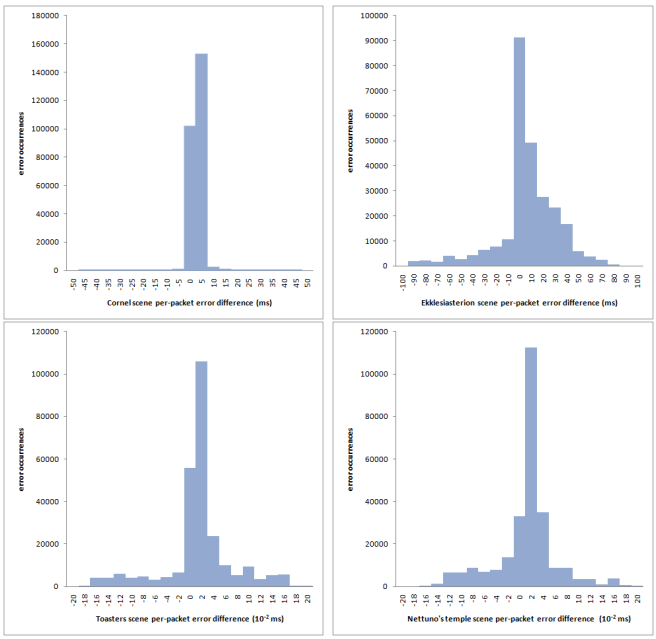


Figure 3.6: Error distribution of the estimation. Each packet-based rendering time is subtract to the cost map estimate, for the same corresponding packet of pixel. The x-axis shows the difference in error intervals, from negative values (left, over-estimation) to positive ones (right, under-estimation). The y-axis plots error occurrences for each error interval. These tests have been performed on one Intel Pentium IV CPU 3.40GHz with 2048KB cache size.

adaptive techniques may suffer of inaccuracy, using estimation for sorting-based approaches relax the requirement of an accurate cost estimation.

3.5 Load Balancing

In this section, we describe how we use the cost map, and a summed area table [34] created from it, for two different load balancing strategies. Summed area tables (SATs) allow us to compute the sum of values in a rectangular region of an image in constant time. Given a cost map of dimension $n \times n$ we can compute a SAT directly on the GPU in $O(\log n)$ time [56].

SAT Sorting A direct use of the SAT is to compute the cost of a tile after subdividing the image in equally sized tiles. Next, we can sort the tiles for decreasing cost, assuring that computationally more expensive tiles are scheduled before cheaper ones. The reason for this approach is that dynamic load balancing typically works better if nodes work on more expensive tasks at first, and task transfers or steals are performed for smaller tasks afterwards.

SAT Adaptive Tiling We can alternatively use the SAT to achieve an adaptive subdivision of the image into tiles of roughly equal cost. Our adaptive subdivision algorithm can be figured as a weighted kd-tree split using the SAT to locate the optimal split in each step.

We introduce two temporary deques (double-ended queues), P and Q, and the cost map C and the number of iterations l as input. During subdivision, we use Q to store the current tiles to be split, and P to store the tiles that have already been split. The resulting subdivision of our algorithm, shown in pseudo-code below, is well balanced and all tiles exhibit almost equal cost:

Algorithm 3 The SAT-tiling algorithm. Starting with a single tile covering the entire image (i.e. of the same size as the cost map), each iteration chooses a split-axis and subdivides the tile using a binary search in order to obtain two tiles T_1 and T_2 with approximately the same cost. These new tiles are enqueued in P and might be further split in subsequent iterations. The running time $T(n)$ of SAT-tiling, for a SAT of $n \times n$ pixels, is $O(n \log n)$.

```
1:  // Set the first split axis (0=x-axis, 1=y-axis)
2:  Axis ← 0
3:  // Create and enqueue the initial tile (covering the entire image) to P
4:  Enqueue(P, CreateTile())
5:  // Loop l times to obtain 2^l tiles
6:  for i ← 0 to l do
7:     // Q contains all tiles to be split
8:     Q ← P
9:     // P stores the newly split tiles
10:    P ← ∅
11:    while Q ≠ ∅ do
12:       // Remove a tile T from the queue
13:       T ← Dequeue(Q)
14:       if axis = 0 then
15:          (T1, T2) ← SplitX(T)
16:       else
17:          (T1, T2) ← SplitY(T)
18:       end if
19:       // Enqueue two tiles T1 and T2 and select the next split axis
20:       Enqueue(P, T1)
21:       Enqueue(P, T2)
22:       Axis ← 1 - Axis
23:    end while
24: end for
```

3.6 Implementation

Our implementation of the work assignment, scheduling and dynamic load balancing is decoupled from the ray tracing implementation that is running on the individual worker. We base our ray tracing implementation on Manta [8], which provides the components of a modern ray tracing architecture: the synchronous parallel pipeline and the set of modular components, such as shaders and image traversal. We extended this system with new image traversal algorithms, load balancers, shaders, and by adding new components. In particular, our parallel code is hidden behind the traversal logic, with a master-side and a worker-side component. The first is responsible for the prediction and assignment, and implements a GPU-based rendering system with programmable shaders. The latter hides the CPU-based ray tracing system and the dynamic distributed load balancer. A material table is in charge of linking the ray tracer shading information with the GPU-cost map generation. As we use static scenes, we use precomputed kd-trees built with a SAH metric as acceleration data structure.

In the following, we discuss the main challenges of our ray tracing system: the tile-to-packet mapping, the dynamic load balancing, the multi-threading parallelization, the network optimizations, and the asynchronous prediction.

3.6.1 Tile-to-Packet Mapping

The use of two different levels of parallelism, one being the packets of the ray tracer, and one being the splitting of the image into tiles, raises the problem of how to map a tile to packets. Each packet-based ray tracer usually has an optimal packet-size, mainly depending on the scene, the acceleration data structure, and the hardware architecture. Recent work encourages the usage of large ray packets for Whitted-style ray tracing [77]. Similarly, a parallel distributed memory system has an optimal task size that depends on the ratio of computation to the amount of communication, being critical in systems like a cluster of workstations. Fixing the same granularity for both with an *one-to-one* approach does not reach optimal performance of the whole system. A one-to-one approach also complicates, and limits, the exploitation of the cost map for adaptive subdivision. Our system uses a *one-to-many* approach instead: each tile is subdivided into packets of fixed optimal size, e.g. each tile is subdivided in packets of 8×8 rays per packet.

3.6.2 Work Stealing

Our parallel system performs *in-frame steals* to improve the load balancing computed from the cost estimation. Note that a perfect cost map would make work stealing superfluous; however, this cannot be expected from an image-space estimation. Our work stealing implementation follows the scheme suggested in [11] where each worker has a *queue*[3]: each worker first processes his own tasks

[3] According to [11] we use the term *queue*. However, as our work stealing algorithm performs operations in both the top and the bottom of the queue, the correct term would be *deque*.

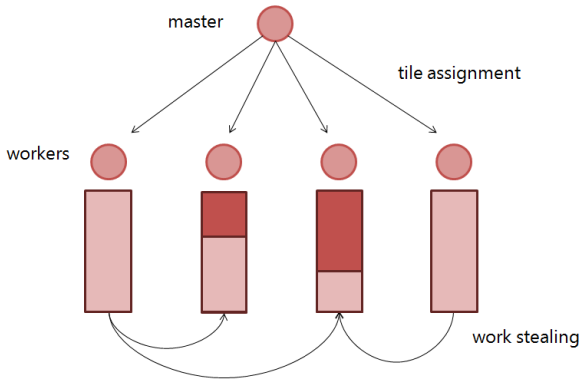


Figure 3.7: A master node is responsible for the first task assignment. Using work stealing idle workers search among the other workers in order to find unprocessed tiles.

starting from the top of his queue. When the queue is empty, workers start stealing tasks from another randomly chosen worker (Figure 3.7). Although dynamic load balancing is distributed, our task assignment is not distributed. All tiles are assigned at the beginning of the frame by the master node, using prefetching to hide latency and assuring fairness. Then the dynamic distributed load balancing algorithm takes care of an initially unbalanced work distribution. Further optimizations in the work stealing protocol can save communication when, for instance, two nodes send crossed steal requests. In this case, we can avoid sending the two negative ack messages. An important aspect when implementing in-frame steals is to take care of the frame synchronization barrier. A node entering into stealing mode performs steal requests until a new frame starts. The new frame message, however, is sent by the master node without a guarantee for order-preserving and delivery, i.e. it may happen that a node at frame $f + 1$ receives an old steal request by a node at the frame f. We solve this problem by adding the frame number to the steal request and steal ack messages, avoiding transfer to nodes that are still unaware of the new frame.

3.6.3 Multi-threading Parallelization for Multi-core CPUs

In the previous section, we discussed the problem of parallelization across distributed worker nodes. On each node the ray tracing itself can also be parallelized via ray packets if multi-core CPUs or SIMD instruction sets are available. The parallelization of multi-threaded ray tracing engines has been explored in several works. Manta uses a centralized on-demand scheduler, which assigns groups of packets to threads; in order to increase balancing, groups are bigger at the beginning and smaller at the end, assuring a fine grained balancing. The implementation of the scheduler takes advantage of a fast atomic counter for synchronization; the only barrier in this approach is the frame barrier (one for

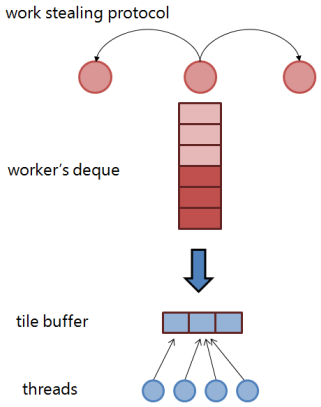


Figure 3.8: Multi-threading with tile buffering. A small number of tiles are moved from the queue to the buffer. This allows separating the work stealing algorithm, working on the queue, from the multi-threading parallelization, working on the tile buffer. The tile buffer can effectively improve multi-threading performance avoiding expensive synchronizations.

each frame).

On top of Manta, our distributed parallel architecture introduces the queue of tiles as an additional element, which raises new questions on exploiting multi-threading parallelism.

A first, trivial attempt to parallelize the queue access is to take a tile from the queue, and distribute packets to threads by using the same approach that is used in Manta. The main problem of this approach is that it introduces a thread synchronization for each tile, leaving threads idle and waiting for the last one that finishes rendering. In fact, this approach shows a bad scalability and the speed-up is 2-3× for scenes where the native Manta is close to 4× (on a quad-core CPU obviously).

We addressed this problem by introducing a tile buffer (Fig. 3.8). At the beginning, a thread moves T tiles from the queue into the tile buffer. When a thread finishes working on a tile, it continues with the next tile in the buffer.

When the last thread working on a tile finishes, it is in charge of sending the image to the master node and moving the next tile from the queue to the buffer. The last thread working on a tile can be easily determined by using an atomic counter for each tile. If the queue is empty, the work stealing protocol may perform one or more steals in order to detect the next tile to be inserted into the buffer.

As we will show in Section 3.7, this approach improves scalability and assures that threads have almost no idle time during rendering. However, the size of the buffer should be carefully chosen. If the size is too large, then the work stealing protocol may not function properly since only tiles in the queue are subject to steal. In contrary, a smaller buffer may introduce thread idle times.

In our test we measured higher performance for $T = 2 \dots 4$ with few workers, whereas $T = 2$ was the optimal value for 8 workers.

3.6.4 Network and Latency Hiding

We utilize MPI as a means to exchange data between nodes [69]. MPI is the dominant programming model in the high performance computation domain. It provides message passing utilities with a transparent interface to communicate between distributed processes without considering the underlying network configurations. However, for interactive purposes often an *ad hoc* low-level implementation is preferred (for example [102] use an optimized TCP version, whereas [37] use both TCP and MPI). The main problem in using libraries is the lack of control of low-level mechanisms, such as the Nagle optimization in TCP [73]. In [102] the authors illustrate that enabling the Nagle algorithm is desirable for scene updates and scene data streaming, but it is less favorable for operations that require a minimal latency (e.g. sending tile requests). Despite that, our image-based ray tracer does not perform any scene data streaming. Moreover, current MPI implementations often set up the best configuration to hide latency (for instance, OpenMPI disables Nagle algorithm by default when using TCP network). Another advantage in using MPI is the ease of adapting our system to different networks and their optimized vendor-provided MPI versions (e.g. Myrinet or Infiniband). In this work, we incorporate MPI as the basic means to communicate data between distributed computation nodes. Our system hides latency by implementing task prefetching and using asynchronous data transfer where possible. At the beginning of each frame, the camera position and a list of prefetched tiles is sent to each worker.

3.6.5 Asynchronous Prediction

The overhead introduced for generating the cost map and the SAT and to compute the tiling causes a longer barrier. Thus, our approach is only effective if we have a significant improvement in balancing. We hide this overhead by doing an asynchronous prediction: after the task assignment for the frame f, when all workers are busy, the master node starts computing the prediction for the frame $f + 1$. The overall architecture is still synchronous, whereas just the prediction phase is computed asynchronously.

3.7 Results

Test platform

We ran several benchmarks on a test platform consisting of a cluster of workstations equipped with Xeon Quad-Core Clovertown CPUs running at 2.33 GHz and having 16 GByte RAM. Nodes are interconnected with

an Infiniband network (⟼ additional details on Appendix A.2). Our code has been compiled using the Intel C++ Compiler. The cost map computation has been implemented using OpenGL and the OpenGL Shading Language. Running the master and the worker node on a single host may reduce performance. For this reason, we used a specific visualization host as the master node. The master and visualization node is equipped with an NVidia GeForce 8800 GTX.

Test scenes All images have been rendered at a resolution of 1024×1024 pixels, with 64 tiles and a maximum recursion depth of 4, resulting in highly varying rendering cost in regions with inter-reflections. Asynchronous prediction has been enabled in all tests. In order to understand the impact of our techniques, we used four different test scenes (Figure 3.12), having different rendering parameters, rendering technique and overall workload: First two scenes use Kajiya path tracing and are computationally expensive. The First is a Cornell box with a reflective Bunny and an area light. For each pixel, we apply a jitter pattern of 128 rays/pixel, shooting 134.2 million of primary rays. The Ekklesiasterion scene contains an ancient Greek building. For this scene, we used 32 rays/pixel. Other two test scenes, instead, make use of Whitted ray tracing with respectively 8 and 1 rays/pixel. Toaster and Poseidonia-Paestum temple both present a large number of reflective surfaces. Table 3.9 shows ray-packet size, the number of primary rays and the rendering parameters used for each test scene. We run first two test scene (the more computational expensive) using up to 16 workers, the last two using up to 8 workers. Each worker uses 4 threads.

Notice that our work differs from [6] in two aspects: First, we use Whitted ray tracing and Kajiya path tracing instead of Instant Global Illumination; second, our test scene present high variance in shaders and geometry. Both contribute to have a high workload unbalance per pixel (as shown by real cost map in Figure 3.13), whose directly impacts on system scalability. For these reasons we have a difference approach to scalability, much similar to frameworks like [16].

For each test scene, we show results using 4 different balancing approaches: A simple not-balanced static approach, using equally sized tiles (*Regular without WS*); a regular approach using work stealing for load balancing (*Regular*); an adaptive approach build over a SAT of the cost map, and enabling work stealing (*SAT Adaptive*); a sorting-based approach exploiting SAT and enabling work stealing.

Table 3.2 shows performance for 8 workers. In addition, we calculate parallel speedup and efficiency.

A steal transfer analysis has been shown for all the test scenes and 8 workers. This analysis is helpful to understand how different tiling algorithms work and, once an initial tile set is assigned, how balancing algorithms integrate with work stealing algorithm.

Scene	Cornell box	Ekklesiasterion	Toasters	Temple
# triangles	69495	3346	11141	13556
Rendering tech.	path tracing	path tracing	whitted	whitted
Rays/pixel	128 jitter	32 jitter	8 jitter	1
Packet size	8	16	16	32
Primary rays	134.2M	33.6M	8.4M	1.0M
Sampling pattern	wide pattern	wide pattern	collapsed to a line	collapsed to a line
Edge detection	no	no	yes	yes
Samples gathering	sum	sum	max	max

Figure 3.9: Summary of the parameters used for rendering and cost map computation. The packet size is the optimal value for the test scene.

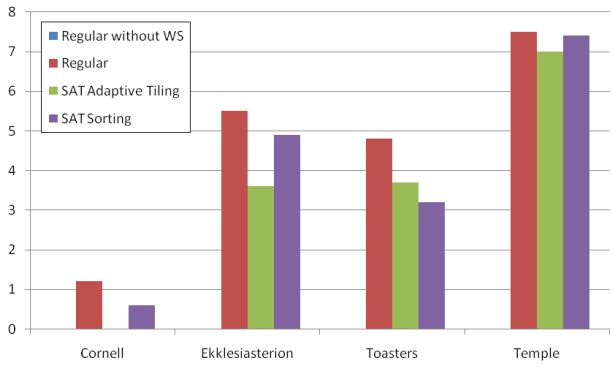


Figure 3.10: Steal transfers analysis. The graph shows the average number of steal transfers performed during 10 frames. The analysis refers to the 4 workers test asset. Note that nodes may steal even more than one tile per frame.

3.8 Discussion

SAT Adaptive Tiling Our results with adaptive tiling show that, compared to a *Regular* approach, provides some slight improvement in performance. However, it never overcomes the *SAT Sorting* approach. Steal transfers analysis (Figure 3.10) furnishes some significant additional informations. An important issue is that the number of steal transfers of adaptive tiling is often the lowest. This means that: (1) the initial tile assignment provided by the adaptive tiling is more balanced than the regular one; (2) however, work stealing do not work well with such kind of (almost balanced) workload. In fact dynamic load balancing, in our case the work stealing strategy, rivals adaptive tiling since both try to balance the workload: a fine grained balancing with dynamic techniques requires more expensive tiles at the beginning and the cheaper ones at the end; but in contrast to that, adaptive tiling aims to equally balance cost between tiles.

The adaptive tiling performance is the worst for the Ekklesiasterion scene. In fact, this scene shows the least accuracy (see Figure 3.6). This indicates that the accuracy required by an adaptive approach is critical, and in general a very accurate cost map is required in order to significantly increase rendering speed.

SAT Sorting The use of the SAT for sorting tiles always improves performance. Contrary to adaptive tiling, this technique does not require an exact estimation of the rendering cost: it just needs a correct tiles ordering. In particular, the use of the *SAT Sorting* strategy, combined with a distributed load balancing algorithm helps assuring a good load balance. The combined use of both techniques, sorting and dynamic load balancing, also achieves a good scalability with the number of workers (Table 3.2).

Multi-threading Scalability Multi-threading parallelization addresses the problem of accessing the tile queue and handling the send request communication (Section 3.6.3). In order to determine its efficiency we performed 3 tests (Table 3.3). First, we measured the rendering performance for a standard scene on a single node and the scalability for 1 to 4 threads. Afterwards, we performed a test by using a master and a worker node running on two different hosts measuring the multi-threading scalability up to 4 threads (for the workers) and the MPI overhead. Note that in this case, MPI calls use the OpenIB protocol (Infiniband network). Finally, we performed a similar test where both master and worker are running on the same host. In this test, MPI uses the fast shared memory protocol, but both processes compete for the same resources, i.e. primarily CPU cores. This test has been performed in a Intel Core 2 Quad Q6600 CPUs running at 2.40 GHz, using a Cornell box scene with different rendering parameters.

This test indicates that whenever we decide to use the same host for both visualization (hence image gathering) and computation, we should be aware that we potentially lose computational power in this host, and hence introduce a small load unbalance.

Distributed Memory System Scalability In order to determine how our system scales with the number of distributed workers, we did a scalability test with 1, 4, 8 and 16 workers. Table 3.4 and Figure 3.11 show the scalability for the Cornell box test scene. Results indicate that the *SAT Sorting* approach provides higher scalability and is particularly useful with high workload. The improvement of *SAT Sorting* strategy, compared to a naïve balancing-unaware approach is about 10-15% with 8 workers, 20-25% with 16 workers. Efficiency is always superior to 90% in Whitted test scenes. Instead, using path tracing we cannot assure similar performance, with an efficiency of 80-90% with 16 workers. We think that such correlation between parallel efficiency and rendering techniques is a new argument in the context of parallel scalability of high parallel rendering systems.

Despite work stealing is a popular approach to distributed dynamic load balancing, its performance had not well understood yet. The effectiveness of our sorting-based approach raises new interesting applications in the context of massively parallel processing. In our system, work stealing is particularly efficient when we have more than 8 workers. Hence, it represents a perfect candidate for today and future massively parallel systems.

Steal analysis also shows that the number of steal transfers is lower with high workloads. We suppose that this is related to the fixed number of tiles size and we plan to investigate how to beneficially change this number accordingly to the workload.

Increasing the quality, e.g. with more samples per pixel and using path tracing, just slightly affects the scalability of the system. However, we will show the raw MRays/sec provided by our system strongly change with different rendering quality settings.

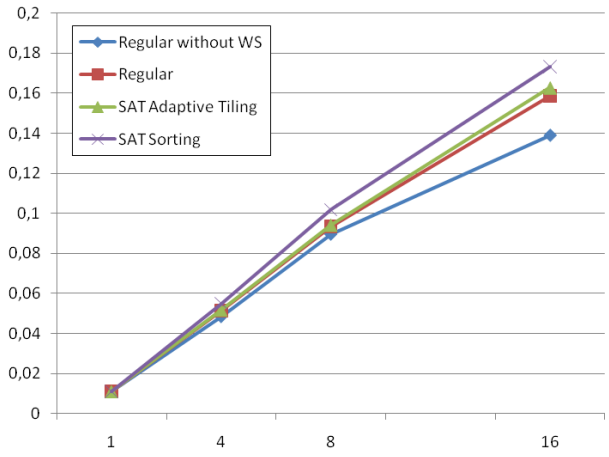


Figure 3.11: Scalability for up to 16 workers measured for the Cornell box. Timings are in frame per second.

Performance evaluation and coherence Despite our test platform do not represent the best CPU architecture available nowadays on the market, using our load balancing approach we provide a high efficiency up to 16 workers. Mrays/sec is a common unit to measure the computational power of a ray tracer. In this context, we try to have a Mrays/sec estimation for our architecture. Anyway, for several reasons (e.g. the difference between rays and shadow rays) this measure should be used carefully.

If we consider the Toasters test scene, using Whitted ray tracing and 8 rays/pixel, we have 33.6M of primary rays. Because we set a maximum depth of 4, the total amount of rays is roughly estimated as 536.871M. Moreover, we must sum to this value the amount of shadow rays generated by a point light source. Hence, a rough estimate of the performance with 8 workers is about 240 Mrays/sec. A similar value has been observed for the Temple test scene.

Instead, if we consider the Cornell box scene, using Kajiya path tracing and 128 rays/pixel, we have 134.218M of primary rays. We set a maximum depth of 4; hence the total amount of rays is roughly estimated as 536.871M. According to the test result, we have about 60Mrays/sec with 8 nodes and 100 with 16 nodes (note that Ekklesiasterion presents similar values). That's means that the system shows a good parallel efficiency, but the computational power exposed by a single node is rather lower than excepted, if compared to other test scenes rendered with Whitted ray tracing.

We argue that a chunk of performance is missing for some coherence-related issues. Global effects (e.g. path tracing) are incoherent, and packet-based techniques too do not help whenever rendering algorithm lack of coherence between rays. However, our path tracing implementation do not use any optimization to increase coherence, e.g. rays sorting. Also in Whitted ray tracing there is some coherence missing after first bounces.

3.9 Conclusion

In this Chapter, we described a parallel ray tracer for distributed memory architectures. Our method demonstrates a combination of different techniques to improve load balancing. We compute a per-pixel ray tracing cost estimate using fast GPU algorithms. We further exploit this information, and use work stealing, and in-frame steals, for dynamic load balancing algorithms. These techniques are combined with a fast packet-based ray tracer, and further network optimizations. Our results indicate that tile sorting based on a cost estimate fits well into dynamic load balancing, and provides good scalability for multiple workers.

In future work, we would like to see how effective these techniques might be for GPU-based ray tracing implementations (e.g. Nvidia OptiX [81] employs a three-tiered dynamic load balancing approach on multi-GPUs).

With increasingly computational power available for commodity hardware, we believe that similar balancing techniques will become of growing interest.

Acknowledgments

This work has been partially funded by a DAAD Scholarship and a HPC-EUROPA2 project (#228398). We wish to acknowledge Thomas Ertl and Vittorio Scarano for supporting this work. We also acknowledge the members of the SCI Institute at the University of Utah for their support with Manta code. Many thanks to the modelers Roberto Andreoli for the Paestum Temple and Ekklesiasterion, and Veronica Sundstedt (Computer Graphics Group, University of Bristol) for the Kalabsha temple. The Toasters scene is from the Utah 3D Repository, Cornell Box from the Cornell University, the bunny from the Stanford 3D Scanning Repository; Environment maps are taken from Debevec repository.

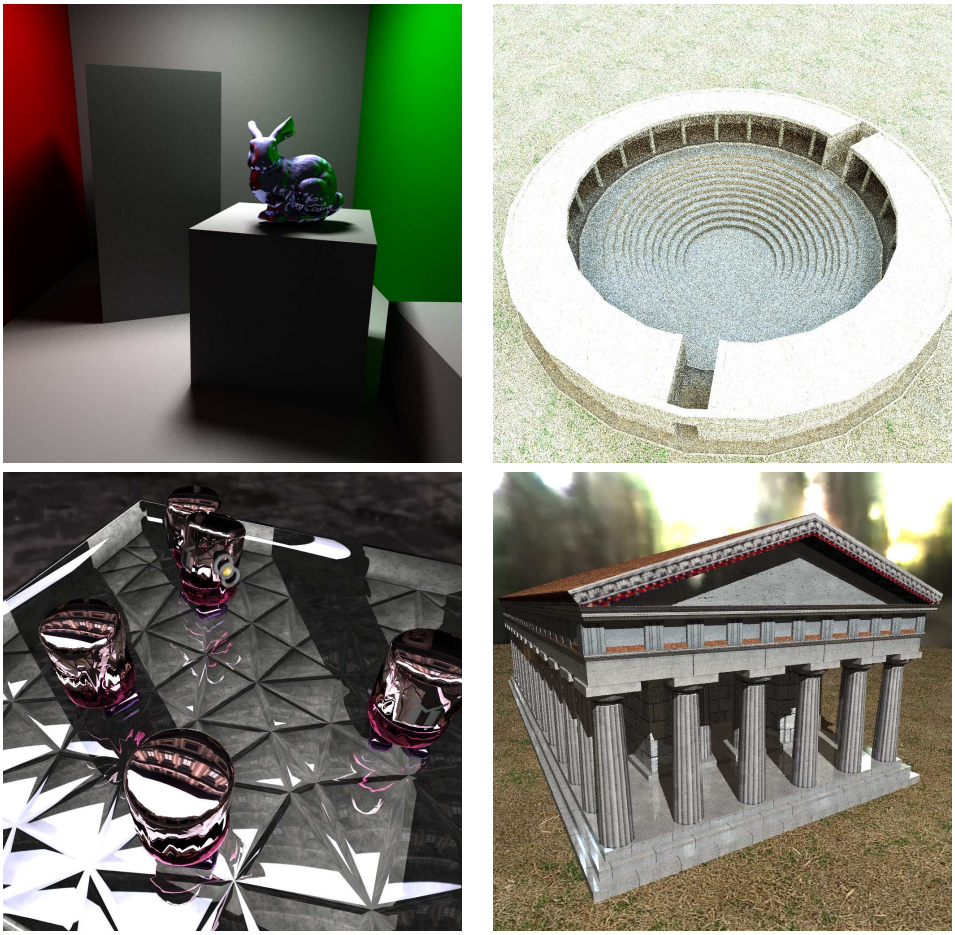


Figure 3.12: Images rendered with our parallel ray tracer.

Scene	Cornell box	Ekklesiasterion	Toasters	Temple
Regular without WS	11.1645	2.90615	0.262	0.0481
Regular	10.7078	2.88738	0.254	0.0433
SAT Adaptive Tiling	10.3100	2.86813	0.245	0.0417
SAT Sorting	9.8027	2.45724	0.225	0.0390
1 worker	74.656	16.494	1.778	0.2920
speedup*	6.7 - 7.6x	5.7 - 6.7x	6.8 - 7.9x	6.1 - 7.5x
efficiency*	83 - 95%	70 - 83%	84 - 98%	75 - 93%
	+12%	+15%	+14%	+19%

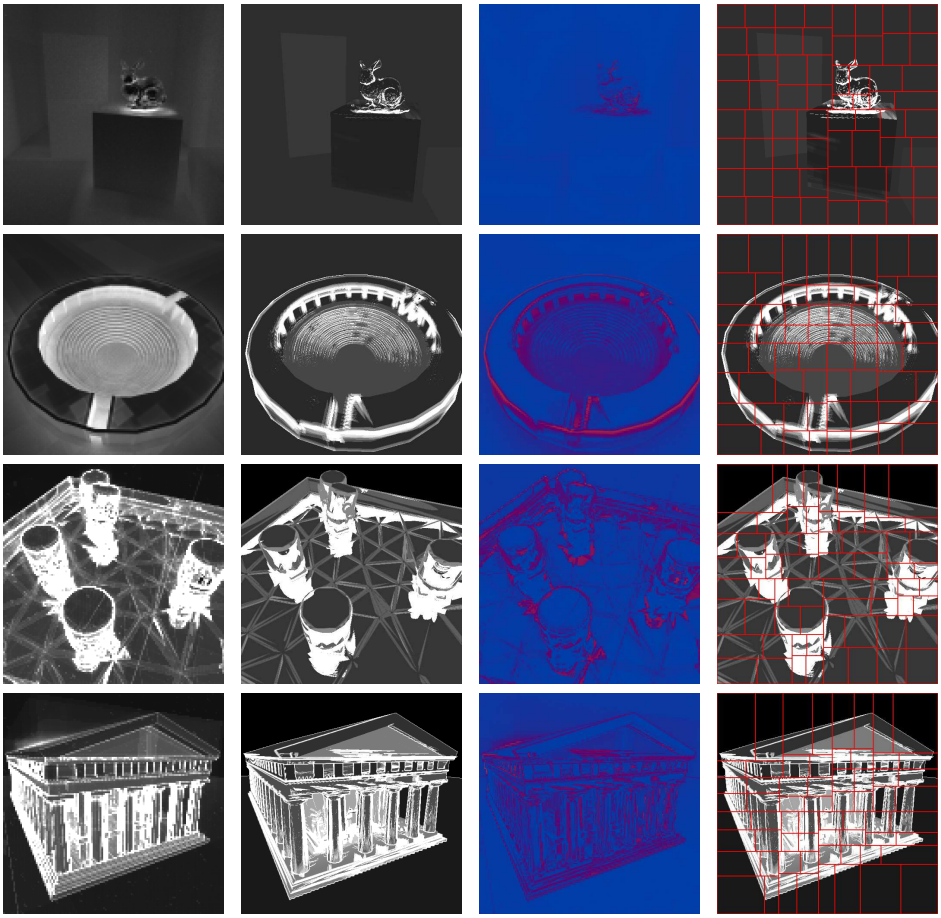
Table 3.2: Performance comparison for our four test scenes; timings are given in seconds per frame. Tests have been performed with 8 workers and multi-threading. (*) Speedup and efficiency are shown for *Regular without WS* and *SAT Sorting approach*

Test setup	1	2	3	4	Speed-up
Manta standalone	0.160	0.319	0.475	0.629	3.93×
Master + a remote node worker	0.159	0.315	0.468	0.617	3.88×
Master + a worker (same host)	0.159	0.314	0.464	0.468	2.94×

Table 3.3: Exploiting multi-threading scalability with a different number of threads and setups. Timings are given in frames per second. This test uses the Cornell box scene with maximum recursion depth of 4, 8 samples per pixel, and a resolution of 1024×1024. For the load balancing, we used the SAT sorting approaches.

Workers	1	4	8	16
Regular without WS	74.656	20.615	11.165	7.203
Regular	//	19.514	10.708	6.303
SAT Adaptive Tiling	//	19.300	10.610	6.142
SAT Sorting	//	18.705	9.823	5.770

Table 3.4: Scalability for the Cornell box test scene using different balancing techniques. Timings are given in second per frame.



(a) (b) (c) (d)

Figure 3.13: Effectiveness of the cost map generation in different test scenes. For each scene, we show real the packet-based cost map based on timings (a), the GPU-based cost map estimate (b), an explanatory difference map (c), and the resulting adaptive tiling (d). The cost maps are obtained mapping the cost of the pixel/packet into the range $[0, 1]$. The difference map is mapped on a gradient: In blue areas, estimation is more precise, whereas in red areas is less accurate.

4

Distributed Load Balancing on Agent-Based Simulations

We have shown that even an embarrassingly parallel problem may present high irregularities in the workload. Hence, we should care of load balancing by using advanced techniques and combining them together keeping in mind the underlying hardware architecture we are using. Tackling of load balancing is even trickier when tasks present dependencies to other tasks and we should carefully handle expensive data movement between processors. For instance, scientific simulations present both an intricate task dependency pattern and workload irregularity (i.e. load balancing problems). In this context, implementations on distributed memory architecture do not suit the use of algorithms like work stealing with very random steals. By converse, such simulations seem to be a perfect candidate to experience different kind of distributed strategies.

In this Chapter, we focus on a class of simulations dubbed *Agent-based Simulation* where a large number of agents move in the space, obeying to some simple rules. Since such kind of simulations are computational intensive, it is challenging, for this contest, to let the number of agents to grow and to increase the quality of the simulation. A fascinating way to answer to this need is by exploiting parallel architectures.

We present a novel *distributed load balancing* schema for a parallel implementation of such simulations. The purpose of such schema is to achieve an high scalability. Our approach to load balancing is designed to be lightweight and totally distributed: the calculations for the balancing take place at each computational step, and influences the successive step. To the best of our knowledge, our approach is the first distributed load balancing schema in this context. We present both the design and the implementation that allowed us to perform a number of experiments, with up-to $1,000,000$ agents. Tests show that, in spite of the fact that the load balancing algorithm is local, the workload distribution is balanced while the communication overhead is negligible.

4.1 Introduction

The simulation of groups of agents moving in a virtual world is a topic that has been investigated since the 1980s. A widespread approach to this kind of simulations has been introduced in [91] and it takes inspiration from *particles system* [85]. In a particle system there is an *emitter* that generates a number of particles that move accordingly to a set of physics-inspired parameters (e.g. initial velocity, gravity). The particle system approach is expanded with the purpose of simulating a group of more complex entities, dubbed *autonomous agents*, whose movements are related to social interactions among group members.

A classical example of use of this approach is the flocking model proposed by Reynolds [91], which allows to simulate a flock of birds in the most natural possible way. Elements of this simulated flock are usually named *boids* (from *birdoid*) and got instilled a range of *behaviors* that induces some kind of *personality*. The behaviors are, in the most of cases, simply geometric calculations performed on each boid. Commonly, every boid is subject to three different behaviors: *separation* from other boids, *alignment* to other boids flight direction, and *cohesion* to other boids. Each of these behaviors is rendered by a force that is applied on the boid, and whose intensity depends on a fixed number of near flockmates, within a given radius which determines the boid's Area Of Interest (AOI). Actual implementations of the boid model may vary [45, 75, 90, 97] but the idea is the following: at every step of the simulation, for every boid b and for each behavior in the personality the system calculates a request to accelerate in a certain direction as the result of a weighted sum of all the forces applied on b.

As a counterpart of the realism of the model the computation complexity of the model is $O(n^2)$, where n is the number of agents in the simulation. A way to achieve good performances, as the number of the agents increases, we can distribute the calculation on a number of workers. Several parallel implementation of the flocking model have been proposed (cf. [45, 46, 90, 113]). A good parallel implementation should strive to achieve two conflicting goals: (1) balance the overall load distribution, and (2) minimize the communication overhead due to tasks interdependencies.

A simple way to partition the whole work into different tasks is to assign a fixed number of agents to each available worker [93]. This approach named *agents partitioning* allows a balanced workload but introduce a significant communication overhead (an all–to–all communication is required).

By noticing that social forces decay exponentially with distance, most of agent-based simulations systems limit the interaction between agents to a fixed range that is agent's Area of Interest (AOI). Using this observation, several *space partitioning* approaches have been proposed [46, 112, 113] in order to reduce the communication overhead.

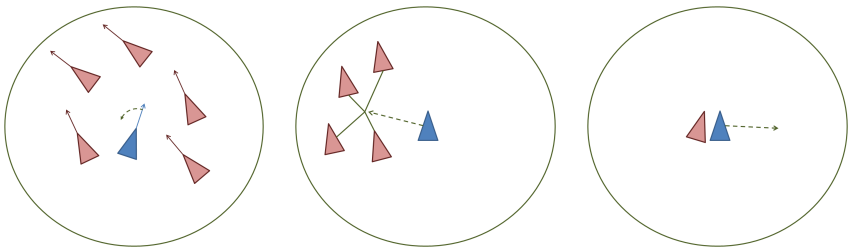
In this approach, the space to be simulated is partitioned into *regions*. Each region, together with the agents contained are assigned to a worker. Since the AOI radius of an agent is small compared with the size of a region, in this approach the communication is limited to local messages (messages between

workers, managing neighboring spaces, etc.). On the other hand, since agents can migrate between regions, load imbalance may occur among the workers. To maintain an even distribution a dynamic partition mechanism is needed: during the simulation, the space partition is updated according to the observed density (number of agents) of the regions. Again the dynamic partitioning should be conducted without introducing too much communication. For instance, it may be not reasonable to update the partitioning using a centralized approach, i.e. the master decides which agents are to be migrated to which worker, since it would require an all–to–all communications.

Briefly introducing the Reynolds's behavioral model

The behavioral model used in this Chapter has been introduced by Craig Reynolds [91] in order to simulate the animal motion of bird flocks and fish schools, also known as *boid*. The basic flocking model consists of three simple steering behaviors which describe how an individual boid maneuvers based on the positions and velocities of its nearby flock-mates:

- Alignment, steer towards the average heading of local flock-mates
- Cohesion, steer to move toward the average position of local flock-mates
- Separation, steer to avoid crowding local flock-mates



Alignment Cohesion Separation

The radius used to detect local flock-mates are usually different for each single behavior.

4.1.1 Related work

Parallel Agent-based Simulation While the area of agent-based simulations has been actively investigated for decades, commonly the results are concentrated on small scale simulations, i.e. with few thousands agents. Scientific interest raised in studying large scale simulation, with the interaction of more than $100,000$ agents [112].

Several shared memory agent-based simulations implementations have been proposed, on a large variety of hardware platforms.

In [45] the mapping of the flocking behavioral model with obstacles avoidance

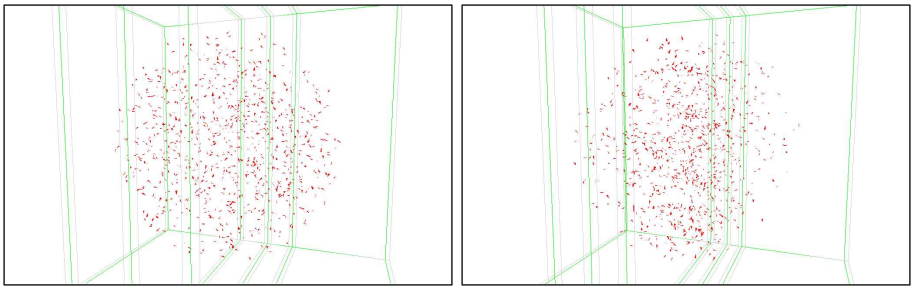


Figure 4.1: Two snapshots showing 1,000 agents simulated by 4 workers: (left) a simulation step corresponding to a sparse distribution of agents; (right) a dense distribution (middle regions are thinner). This screenshots have been obtained by gathering simulation data on a single worker that performed the rendering. A video is available [60].

on streaming-based GPUs is presented. An agent-based simulation optimized for large shared-memory platforms is described in [63]. Similarly, [90] implemented crowds and other flock-like group motion on a multi-core Cell Processor.

Different approaches have been tried on cluster of high performances PCs. Such architectures still require some efforts to tackle the communication/load balancing trade-off. In [84] a 2D parallel framework is proposed that is capable of simulating and rendering the motion of 10,000 pedestrians in real time. This framework is based on a master/worker paradigm: for each simulation step, the master assigns a portion of pedestrian to each worker. Each worker simulates the pedestrian assigned and sends back the result of its computation.

In [112] a result is presented that is particularly relevant to our discussion but from a different point of view: it present a non-conventional use of the flocking model for document clustering, furthermore they implement such algorithms as an hybrid solution, on a cluster of GPUs.

Dynamic load balancing Dynamic load balancing schemes represent a challenge for parallel implementation in several context [57]. For instance, in [23] a dynamic portioning schema have been proposed for Parallel Ray Tracing. In [113] the authors implemented Reynolds' model using a space partitioning scheme with centralized load balancing. Unfortunately, the high computational cost of the proposed load balancing schema precludes the use of such approach on each simulation step, as the authors report.

Other approaches have been explored to parallelize massive simulations on different architectures; for instance in [22] a system is presented that exploits a Peer-to-Peer infrastructure in order to distribute the computational load.

More complex partitioning approaches tackle the load balancing from a geometrical point of view: irregular shape regions (convex hulls) [98]; quad tree, k-d tree, and region growing [92]; Orthogonal Recursive Bisection [46].

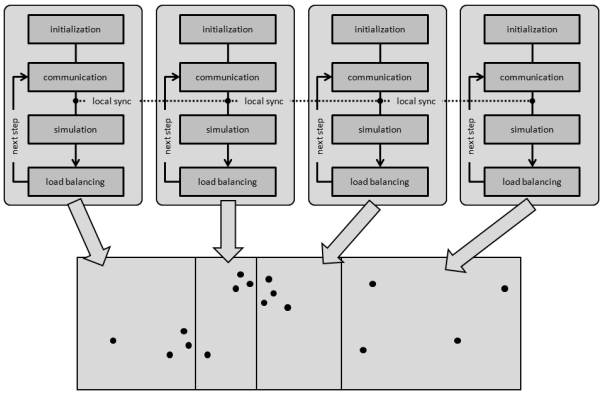


Figure 4.2: The simulation carried out on 4 workers: upper part the steps each worker executes; lower part the main region divided into 4 slices of different sizes, associated to workers.

4.1.2 Our Result

All the works previously cited in this sections exploits centralized load balancing schemes, i.e. involving a worker→master→worker communication pattern every time load balancing is needed. There are two reasons that let us suppose that centralized schema may not be the way to improve significantly the performances, especially in case of large simulations, where all-to-all communication is prohibitively expensive. First, the centralizing communications is bottleneck and may have a negative impact to the system scalability. Second, whereas a centralized balancing schema is used, complex calculations are usually involved, harming system performances.

In this Chapter we present a novel *distributed load balancing* schema whose purpose is to achieve efficiency *and* an high scalability. Our approach to load balancing is designed to be lightweight and totally distributed: the calculations for the balancing take place at each computational step, and influence the successive step. To the best of our knowledge, our approach is the first distributed load balancing schema in this context.

We present both the design and the implementation that allowed us to perform a number of experiments, with up-to $1,000,000$ agents, whose results are discussed in the following sections.

4.2 Background

4.2.1 Behavior Model

Our work is based on the flocking model developed by Reynolds [91]. Every agent has its own personality that is the result of a weighted sum of a number of behaviors. The simulation is performed in successive steps: at each step,

for each agent and for each behavior in the personality, the system calculates a request to accelerate in a certain direction in the space, and sums up all of these requests; then the agent is moved along this result.

The most trivial implementation of the neighborhood calculation consists in a $O(n^2)$ proximity screening, and for this reason the efficiency of the implementation is yet to be considered an issue. Actual implementations rely on the fact that the behavior model is designed to mimic natural-inspired models where the limited visibility of real birds allow to bound the range of interaction.

4.2.2 Parallel Agent Simulation

We developed a distributed agent-based simulation in order to evaluate and compare the performances of several fully distributed load balancing schemes (cf. Section 4.4).

We use a space partitioning model where each worker maintains a portion of the simulated space, henceforth region, and is responsible for the simulation of agents belonging to such region. In order to guarantee the consistency of parallel implementation with respect to the sequential one, each worker needs to collect information about the neighboring regions.

At the beginning of the simulation, the system randomly generates a quantity of agents within the main region. Each simulation step is formed by three phases (cf. Figure 4.2). We describe here the execution of a single step for a single worker. First of all the worker sends to its neighbors the information about the agents belonging its region but that also may fall into the AOI of the neighbor's agents. This information exchange is locally synchronized in order to let the simulation run consistently. During the simulation phase the contribution of each behavior for each agent is calculated as a weighted sum. At the end of simulation phase, each worker is able to yield some statistics on the distribution of the agents within the region. These statistics are shared with neighbors workers in such a way that all the workers are able to calculate the novel partitioning on their own. We emphasize that the load balancing algorithm, which moves the boundary between neighbor regions, is quite simple (cf. Section 4.4) and fully distributed, hence it will not represent a bottleneck for the system.

4.3 Agents partitioning

In order to better exploit the computing power provided by the worker of the system, it is necessary to design the system so that the simulation always evolves in parallel, avoiding bottlenecks. Since the simulation is synchronized after each step, the whole simulation advances with the same speed provided by the slower worker in the system. For this reason it is necessary to design the system in order to balance the load between the workers.

The whole simulation will be carried out in a tridimensional space that will be partitioned along one single dimension in regions (cf. Figure 4.1). Depending

on the kind of simulation, we define a number of parameters that will influence the load balancing schema. A *main region* will be set large enough to contain all the agents in each step of the simulation; this to assure that agents will not move outside this region. The radius for the agents' AOI is named ϵ. This radius, as well as the shape of the AOI, is correlated to the type of simulation. In the following we assume that ϵ is small compared to the size of a region and each agent, in a single step of simulation, is not allowed to move for more than ϵ. The value of ϵ is important because it will influence the amount of communication between two contiguous workers (i.e., workers with adjacent regions).

4.3.1 Handling the boundary

For sake of clarity we will discuss the rationale behind the design of our load balancing schema by describing it in the simpler case of two workers. In Section 4.4.5 we will generalize the schema to a any number of workers.

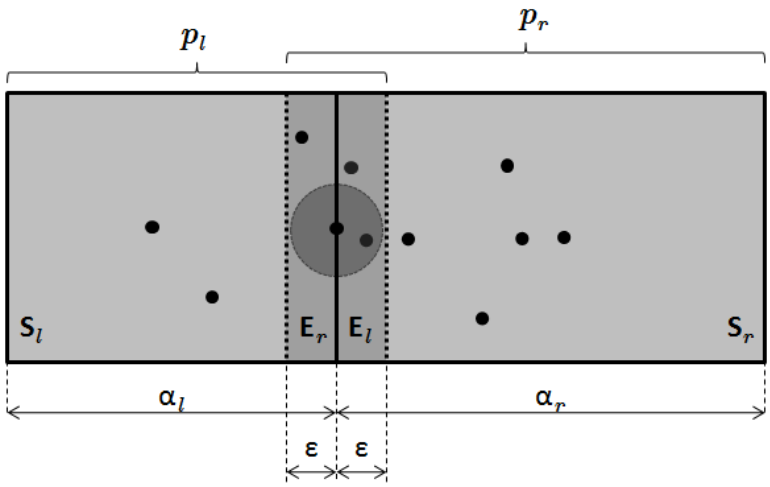


Figure 4.3: Load balancing with two workers: the main region is partitioned into S_l, simulated by p_l and S_r simulated by p_r. E_l (resp. E_r) represents the portion of S_r, (resp. S_l) that needs to exchange information before each simulation step.

The main region is partitioned into two regions, S_l and S_r. In Figure 4.3 we depict this situation, please note that even if our system is designed for tridimensional space, for sake of clarity, our figures will present a bidimensional space. For brevity, we will describe the idea for worker p_l, without loss of generality. The agents present in the region S_l are simulated by worker p_l. AOI of some of the agents in S_l will intersect S_r and, for this reason, throughout the simulation it will be necessary to share the information about the agents in such AOI. To handle this situation we define E_l as the portion of S_r that contains

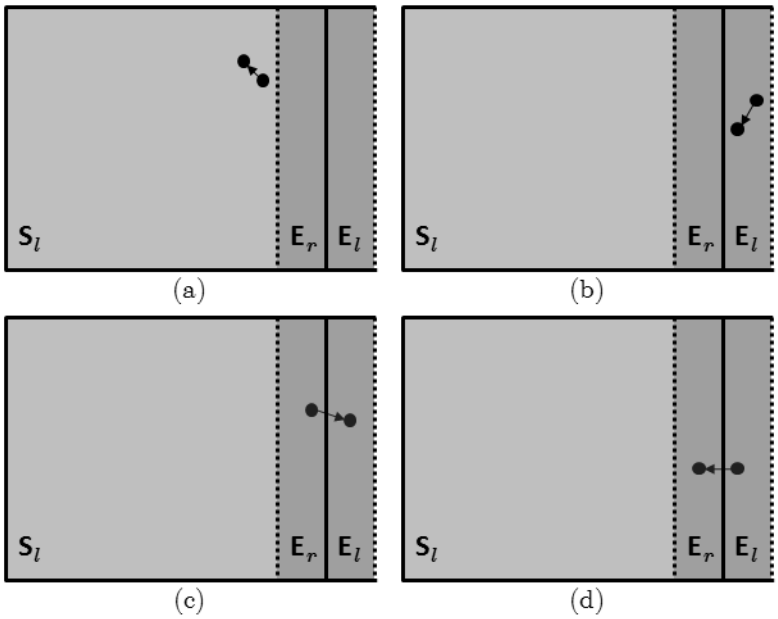


Figure 4.4: Four cases for agent position when moving to a new simulation step.

the agents laying in the AOI of some of the agents in S_l. In other words, E_l is the leftmost ϵ-wide slice of S_r.

The worker p_l, to carry out the simulation of agents in S_l, needs all the agents lying in both S_l and E_l. Before each simulation phase, the position of the agents lying in E_l needs to be updated with information coming from p_r, the same happens for E_r and p_l.

When more than two workers are involved, each worker, except for the first and the last one, has two neighbors. In this case before each simulation phase, each worker communicates with both its neighbor in order to be updated about the position of agents close to its boundaries.

4.3.2 Special Cases

The exchange of information between p_r and p_l needs to take into account the fact that agents may move across the regions and different cases may raise (cf. Figure 4.4): (a) an agent laying in S_l before and after the simulation step; (b) an agent laying in E_l before and after the simulation step; (c) an agent moves from S_l to E_l; (d) an agent moves from E_l to S_l. Case (a) and (b) are easy to be handled because the agent continues to be simulated by the same worker. Cases (c) and (d) is where communication between p_l and p_r is needed to hand over the agent.

We will shortly discuss case (c) in Algorithm 4 where an agent moves from S_l to E_l, in this case two things must be taken into account: the agent will be

sent to p_r (line 6) and it must be also kept by p_l because it may belong to the AOI of some of the agents in S_l (line 9).

Algorithm 4 Handling the boundary (code for worker p_l)

1: {Partition agent's set in 4 subset}
2: $M_a \longleftarrow$ { agents in case (a) }
3: $M_c \longleftarrow$ { agents in case (c) }
4: $O_c \longleftarrow$ { agents in M_a belonging to E_r }
5: $O_a \longleftarrow M_c$
6: *send* O_a and O_c to p_r
7: *receive* I_a and I_c from p_r
8: agents in $S_l \longleftarrow \{M_a \cup I_a\}$
9: agents in $E_l \longleftarrow \{M_c \cup I_c\}$

4.4 Distributed load balancing schemes

In this Section we present three load balancing schemes we have developed and tested on our system. The rationale of these schemes is to provide a distributed load balancing by using just local communication.

As described above, the main region is partitioned into regions by slicing it along one single dimension (cf. Figure 4.3). We denote with α_l the size of S_l along such dimension. Let s_l (resp. s_r, e_l, e_r) be the number of agents in S_l (resp. S_r, E_l, E_r). Based on the load observed by p_l and its neighbor, the load balancing algorithm will modify the value of α_l by moving the boundary; the purpose is to improve the load balancing for the successive step of computation.

More formally, the load balancing algorithm aims to select the best value of α_l in order to:

1. minimize the unbalancing, that is $|s_l - s_r|$;

2. minimize the communication required for the synchronization phase, that is $|e_l + e_r|$

To apply each one of the following load balancing schema, the workers, p_l and p_r, need some additional information that will be exchanged during the load balancing phase. We will shortly discuss this *overhead* for each of the following algorithms.

Assumptions Our load balancing schema relays on two assumptions: (i) the measure of the computational load of each worker is linear in the number of agents (ii) the agents are uniformly distributed along the dimension the splitting occurs. The effect of the first approximation will be mitigated by successive refinements of the method, the effects of the second approximation deserve

further investigations but we emphasize that, regardless of the effect of this assumption, we experienced good performances also with a large number of agents (cf. Section 4.5).

4.4.1 Static partitioning (*static*)

In order to provide a baseline scheme which will be compared with our proposals, we have implemented a static partitioning, where the value of α is fixed, for each worker, to d_x/w, where w is the number of workers/regions and d_x is the size of the main region along the splitting dimension. The scheme will be also used to evaluate the degree of unbalancing of a given testbed simulation.

4.4.2 Region wide load balancing (*dynamic*1)

Assuming that agents are uniformly distributed along the splitting dimension in the whole region assigned to a worker, we may define a simple algorithm that moves the boundary by evaluating the values of s_l and s_r.

Let $\alpha_l(t)$ be the value of α_l at simulation step t. The load balancing algorithm updates the value of α_l according to the following equation:

$$\alpha_l(t+1) = \alpha_l(t) + \frac{s_r - s_l}{2} \cdot \frac{\alpha_l(t) + \alpha_r(t)}{s_l + s_r}, \tag{4.1}$$

where the first fraction represents the number of agents to be moved to balance the load between p_l and p_r and the second one is the amount of linear space containing a single agent, under the "uniformly distributed agents" assumption stated above.

The overhead of communication, for p_l, is the transmission of s_r and α_r. Thus, after each simulation phase, p_l exchanges such information with p_r and then both, use Eq. 4.1 to compute the new partitioning. Notice that no additional communication is required to spread the updated boundary position. Clearly this communication does not represent a bottleneck because it is limited to only two values, for each step. Notice that when the number of worker is 2 the exchange of information is not strictly required because both s_r and α_r can be calculate by p_l by using s_l and α_l, respectively. Of course this is not true in the most general case with more workers.

4.4.3 Mitigated region wide load balancing (*dynamic*2)

In this algorithm we aim at mitigate the effects of approximation of considering agents uniformly distributed across the space. In general, such distribution depends on the behavioral model and is usually not uniform. For instance several flock simulations models converge to a state where agents aggregate into few dense groups [19]. On the other hand, if the distribution had been uniform, a static partitioning would be enough to achieve a good load balancing.

The load balancing schema above is prone to a phenomenon, when a large flock of agents rapidly moves between the two workers: the boundary is moved

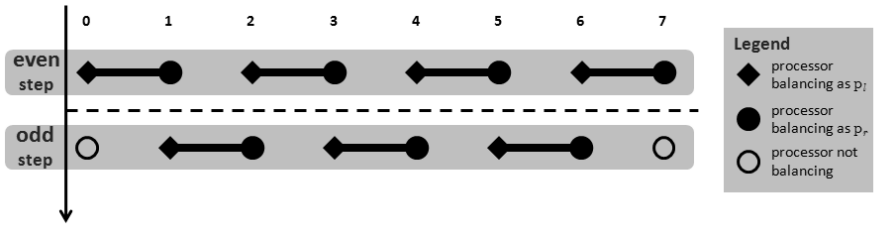


Figure 4.5: Load balancing with multiple workers. Each worker, after each simulation step, updates, by turns, one of is boundaries. Turns are chosen in such a way that neighbor workers always update the position of their shared boundary at the same simulation steps.

too aggressively in the attempt of balancing the flock movement (cf. Section 4.5 for test results) and it oscillates. Such oscillation is an effect of the assumption of uniformity of distribution, that underestimates the amount of agents approaching the boundary. Clearly the wider this oscillation is the larger is the number of agents to be handed over between p_l and p_r.

To mitigate this effect we have slightly changed the balancing equation, by adding a constant k that adds some inertia in moving the boundary:

$$\alpha_l(t+1) = \alpha_l(t) + k \cdot \frac{s_r - s_l}{2} \cdot \frac{\alpha_l(t) + \alpha_r(t)}{s_l + s_r} \tag{4.2}$$

We experimentally observed that this version of the equation with $k = 1/2$ provides good results in alleviating the oscillations.

4.4.4 Restricted assumption load balancing (*dynamic*3)

We refined the algorithm even more by relaxing the assumption of uniformity of the distribution of the agents: the assumption will be applied just to the spaces E_l and E_r, instead of S_l and S_r. As a counterpart we had to limit the per-step movement of the boundary to ϵ which represent the size of both E_l and E_r along the splitting dimension. However such restriction does not represent a real limitation since during the test we have performed, the requested movement was always smaller than ϵ.

$$\alpha_l(t+1) = \alpha_l(t) + \begin{cases} \min\left(\epsilon, \frac{s_r - s_l}{2} \cdot \frac{\epsilon}{e_l}\right) & \text{if } s_r > s_l, \\ \max\left(-\epsilon, \frac{s_r - s_l}{2} \cdot \frac{\epsilon}{e_r}\right) & \text{otherwise.} \end{cases}$$

In this new equation the overhead of communication consists in exchanging the values of s_l, e_l and α_l. Again such information can be rapidly shared between neighbor workers.

4.4.5 Generalization to multiple workers

The load balancing schemes we defined above for two workers can be easily generalized to any number of workers. We describe here a distributed load balancing schema for parallel agent based simulations. The system is composed by a set of n workers having a linear topology, i.e. worker p_i has two neighbors, p_{i-1} and p_{i+1}. Obviously, p_0 and p_{n-1} have a single neighbor.

The rationale behind the generalization is straightforward, in the 2-workers load balancing schema we used p_l and p_r to indicate the two workers. The workers will be distinguished, by their index, in *even* workers and *odd* workers. The idea is to apply the same 2-workers schema to couples of neighbors workers, on alternate steps: on even simulation steps, even workers play the role of p_l and the odd workers play the role of p_r, while in the odd simulation steps even workers will be p_r and odd workers will be p_l. In Figure 4.5 is depicted the generalization in the case of 8 workers: two successive step of simulation are shown. In the visualization it appears clear how on alternate steps worker 0 and worker n will not perform any load balancing.

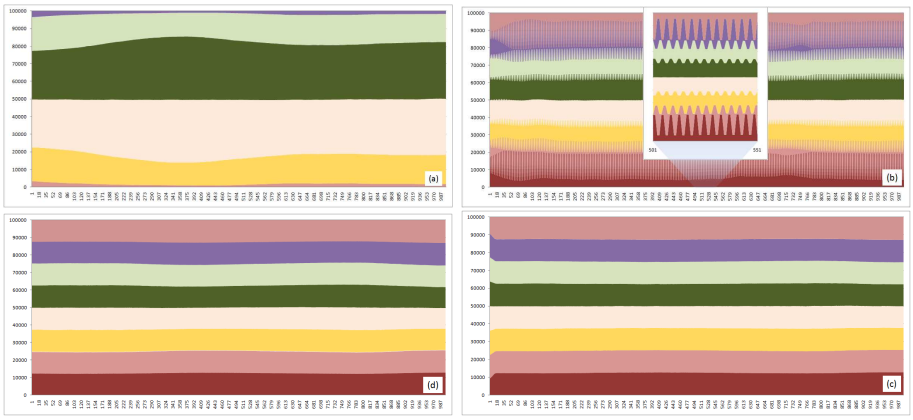


Figure 4.6: Distribution of agents per worker. Each color represents a different worker. x-axis indicates the simulation step (1,000 simulation steps are depicted) while th y-axis represents the number of agents. (a) (100,000; 8; *static*) (b) (100,000; 8; *dynamic*1) (c) (100,000; 8; *dynamic*2) and (d) (100,000; 8; *dynamic*3).

4.5 Tests and performances

4.5.1 Test setting

We have implemented the 4 methods by developing a parallel version of OpenSteer[89]. OpenSteer is an open-source C++ library that implements a plurality of steering behaviors to be used as a standard library for videogames.

Test platform
Test machine is an IBM HS21 Cluster with 256 nodes available at CRESCO Project computing platform, Portici ENEA Center. Each node is equipped with 2 Xeon Quad-Core Clovertown E5345 at 2.33 GHz and 16 GByte RAM. The nodes are interconnected with an Infiniband network ($\longmapsto$ additional details on Appendix A.2).

Software details Our parallelization is based on MPI [69]. In particular the system mapped MPI processes onto cores. Underlying MPI implementation implicitly switches between most suitable protocol to let workers to communicate (e.g. Infiniband rather than Inter-Processes Communication).

We performed tests using a different number of agents, but fixed agent density (hence setting the main region volume accordingly). We set a *bounding radius* in order to assure that agents does not go outside the main region. For each agent overtaking this radius, an additional backward steering force is added to the standard model.

The state of an agent comprises two vectors: position and speed. Other common properties (i.e. mass) are defined constant and are not shared/sent between workers. Of course, the bigger is the state of an agent, the more expensive will be the communication overhead of the parallelization.

4.5.2 Load balancing analysis

The batch of tests we performed simulates $1,000$, $10,000$, $100,000$ agents running on 8, 16, 32 and 64 cores. Each tests lasts $11,000$ simulation steps, the first $1,000$ steps have been discarded in order to let the simulation to stabilize. The set of tests we performed is the result of the Cartesian product $\{1,000,\ 10,000,\ 100,000\} \times \{8,\ 16,\ 32,\ 64\} \times \{static,\ dynamic1,\ dynamic2,\ dynamic3\}$. In the next paragraphs and in figures we will indicate the test setting by using a triple took from such set.

For each simulation step run, we collected the number of agents simulated by each worker/core, the number of agents exchanged between neighbor workers (communication).

The test results are encouraging and confirm that *dynamic*2 and *dynamic*3 handle the balancing of the agents between neighbor workers pretty well. Moreover, the amount of communication overhead injected by the hand over of the agents between neighbor workers is negligible. To avoid cluttering we illustrate the $(100,000;\ 8;\ *)$ cases in Figure 4.6, reporting the distribution of agents among workers, in each simulation step, from upper left and clock-wise we have: *static, dynamic*1*, dynamic*2*, dynamic*3. In Figure 4.6.(a) $(100,000;\ 8;\ static)$ is shown and it depicts the heavy unbalancing in the distribution of the load: two of the workers simulate $\approx 30,000$ agents instead of the ideal $12,500$. Figure 4.6.(b) shows the *dynamic*1 algorithm which provides a better balancing but suffers of the oscillation phenomenon mentioned in Section 4.4.3. To ameliorate

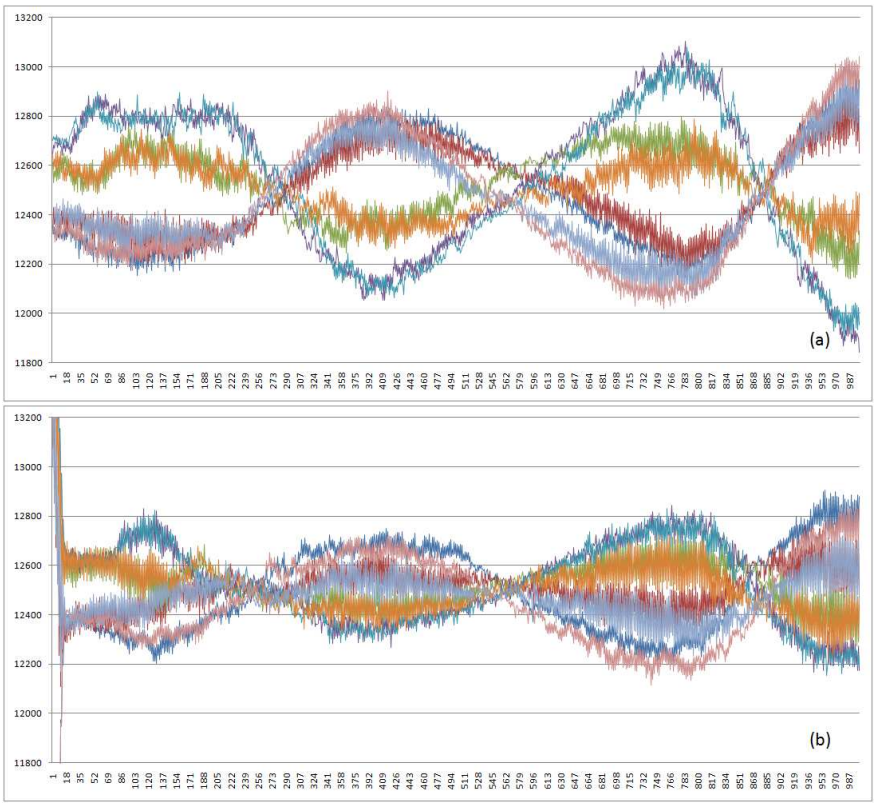


Figure 4.7: Number of agents per worker: each series indicates the number of agents per worker. x-axis indicates the simulation step (1,000 simulation step are depicted) while the y-axis represents the number of agents. (a) (100,000; 8; *dynamic*2). (b) (100,000; 8; *dynamic*3).

the visibility of such phenomenon we provides a zoomed section of the graph (50 simulation steps). The two successive algorithms, *dynamic*2 and *dynamic*3, are shown in Figure 4.6.(c) and (d), respectively. The figure represents an almost optimal balanced graph showing that each worker handles $\approx 12,500$ agents. Both the algorithms sensibly reduce the oscillations, even if they are still measurable, shown in Figure 4.7; please note how *dynamic*3 (b) behaves slightly better than *dynamic*2 (a), by providing oscillations that have smaller amplitude.

In Table 4.1 we summarize the results we measured in other test settings. For each test we report the standard deviation (σ) of the number of agents per worker and the total number of agents exchanged during the communication phase of the algorithm (communication). The results indicates that together with a good performance in reducing the unbalancing we measure an increase in the communication cost. The best refinement of the load balancing schema, *dynamic*3 provides substantially smaller standard deviation but

needed more than the double of agents exchange between workers, respect to the static partitioning. The oscillations we noted in *dynamic*1 deeply impact on the communication cost, this can be noticed by comparing the performances of $(100,000;\ 8/16;\ dynamic1)$ and $(100,000;\ 8/16;\ dynamic2/dynamic3)$. On the other hand, when the number of workers is higher, *dynamic*1 behaves as *dynamic*2 and *dynamic*3 in terms of communication but the balancing worsens. Overall the *dynamic*3 algorithm performs pretty well on all test cases. Moreover, the improvement provided by dynamic algorithms seems to be increasing as the number of either workers or agents grow (see scalability in Figure 4.8).

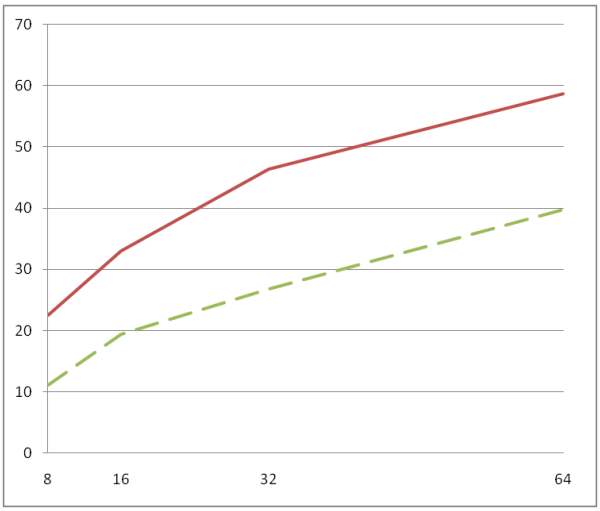


Figure 4.8: Scalability. Continuous line is $(100,000;\ *;\ dynamic3)$; dotted line is $(100,000;\ *;\ static)$. x-axis indicates the number of processors; y-axis plots the average number of simulation step/per second measured during a test of $12,000$ steps.

Discussion Finally, we performed a quite massive simulation: $(1,000,000;\ 64;\ dynamic3)$. The objective of this test is to measure the system performances in circumstances that can hardly be managed by a single machine. The execution of a run of $2,000$ steps required[1] $140,754$ ms, which correspond to 14.21 simulation steps computed per second. We have also observed that, after a small number of steps (around 100), where the load balancing algorithm stabilizes, the performances of the system are quite constant.

Our load balancing strategy even simplifies the tuning of the proximity data structure (e.g. selecting the 3d grid resolution). In fact, whenever workload balance is assured, we may assume that the number of agent per worker is roughly constant, hence to choose the right data structure tuning for this configuration.

[1] In order to measure the total parallel computation time, we used a specific worker that, on every step, collects information about the completion time for each.

4.6 Conclusion

Agent based simulations, due to their computational power requirements, appear to be a natural application for parallel architectures. In this context it is challenging to design the system so that the simulation evolves in parallel, avoiding bottlenecks, in order to better exploit such computing power. Since the simulation must be synchronized after each step, the system advances with the same speed as the slower worker in the system is capable of. For this reason it is necessary to take into account a good implementation of a load balancing mechanism. Several centralized load balancing schemes have been proposed. A common problem with these approaches is that the centralized management usually requires a large amount of communication – between workers and the master node, which act as a load balancing manager – that consumes bandwidth and introduces latency.

We presented a novel *distributed load balancing* schema whose purpose is to achieve an effective load balancing introducing a low communication overhead. Our schema is designed to be lightweight and totally distributed: the calculations for the balancing take place on every worker at each computational step, and influences the successive step; each communication is local (i.e., between neighbor workers). To the best of our knowledge, our approach is the first distributed load balancing schema in this context.

We presented both the design and the implementation that allowed us to perform a number of experiments, with up-to $1,000,000$ agents. The tests revealed that the architecture presents a quite good scalability: the communication overhead, due to the local workers interaction, is dominated by the speed-up achieved thanks to the better load balancing, provided by our schema.

Future works Our load balancing schema aims at balancing along a single dimension the uneven distribution of agents in a tridimensional space. A reasonable evolution of such schema is to take into account the fact that the space is 3d: current implementation does not properly balance work when agent clustering fully exploits the three dimensions. For instance, in the simulation of a flock of birds or a school of fishes we may find several flocks overlapping and spreading across the whole space and not just lying along one single dimension. We plan to extend our technique to a multi dimensional space.

We reported some early tests of a simulation with $1,000,000$ agents. One technical problem we solved was the creation of such amount of agents: this phase is still centralized, in the current version of the system, and for this reason the number of agents in the system was limited by the memory (and the capability of representation) of a single worker. We plan to distribute such phase in order to reach a number of agents that is proportional to the number of workers in the system and it is clear that in such scenario we would easily reach the goal of a multi-millions agents simulation.

Acknowledgments

We gratefully thank for its collaboration in providing some of the computational resources ENEA (Ente per le Nuove Tecnologie, l'Energia e l'Ambiente) Research Center in Portici (Napoli, Italy). This work has been supported by a ISCRA CINECA. We would like to thank Craig Reynolds for OpenSteer.

Table 4.1: Load balancing/communication results

$\times$	$(static)$	$(dynamic1)$	$(dynamic2)$	$(dynamic3)$
$(1,000;\ 8)\{\sigma$ - $com.\}$	131.16 - 272,976	36.03 - 624,562	12.44 - 573,275	6.39 - 584,961
$(10,000;\ 8)\{\sigma$ - $com.\}$	1292.54 - 581,611	550.83 - 2,336,951	13.44 - 1,355,381	6.49 - 1,236,951
$(100,000;\ 8)\{\sigma$ - $com.\}$	11,480.80 - 746,753	5,324.28 - 22,261,600	10.94 - 1,278,788	5.14 - 1,217,074
$(100,000;\ 16)\{\sigma$ - $com.\}$	5,815.35 - 1,407,684	2,146.36 - 16,067,865	12.13 - 3,504,703	7.43 - 3,484,038
$(100,000;\ 32)\{\sigma$ - $com.\}$	3,945.53 - 2,927,194	743.21 - 12,714,472	30.55 - 6,896,584	9.12 - 6,877,622
$(100,000;\ 64)\{\sigma$ - $com.\}$	1,975.22 - 5,622,336	260.21 - 15,888,243	106.06 - 13,679,714	19.44 - 13,204,903

5
Conclusion: Lesson Learned

In this dissertation, we discussed about the emerging problem of load balancing in parallel computing.

The approach used places emphasis on real world practical aspects: We proposed new models and algorithms for specific applications, hence we implemented such techniques in nowadays parallel architectures in order to assess their effectiveness.

In this Chapter we discuss about several interesting aspects emerged in this dissertation, introducing our considerations about some key points that, for our opinion, represent a remarkable lesson to attack the problem of load balancing today.

Towards Massive Parallelism

Nowadays, it is clear that multi- and many-core is the future of computing. We discussed in details about the reasons of that, in particular about three points (Section 1.1.2): the power wall, the memory wall and the ILP wall. That said, researcher in parallel computing will spend more and more time in effective parallelization and load balancing techniques oriented to such architectures.

Distributed is efficient

While centralized algorithms could be smarter, because of the wider knowledge of the computation they have (e.g. the Prediction Binary Tree in Chapter 2), they represent a serious limit to the scalability of a parallel system. For sake of clarity, we resume here the example of the PBT introduced in Chapter 2. For instance, we assume that the time to compute the PBT is $1ms$ (actually it costs fairly less). During this time, master cannot send new task, hence this time is an overhead. The time lost during this phase is about $n * 1ms$, thus it increases linearly with the number of processors. We cannot assure that, for higher number of processors, the gain in load balancing overtakes such overhead. This is a clear limit to the scalability.

In the opposite, if the load balancing technique is distributed, this overhead is essentially lower (it increases sub-linearly over the number of processors). For

example, in Chapter 4, such overhead is bounded to the data exchange of few bytes between close processors (i.e. two processors). Hence, instead of a global centralized synchronization, we have a local distributed one.

Load balancing algorithms should be fast

To introduce a new load balancing technique in a parallel system often means to introduce a new overhead. Such overhead is what we pay to run extra load balancing computation. Obliviously, if the overhead is higher than the advantage we have in balancing the workload, we fail to improve overall performance. Usually, more is the whole computing time, higher is the change we have to use more complex (i.e. time consuming) load balancing techniques. In the opposite, in some interactive contexts, every single computation spent in load balancing is valuable. In Chapter 3, we show that where the overhead is high (6 ms for such example), the only way to have an improvement with load balancing is by using a context-specific hiding technique (in the example, the asynchronous prediction).

When using cost evaluation approaches, we should pay attention to the fast vs accurate trade-off

In Chapter 4 we showed that an adaptive algorithm, i.e. an algorithm that adapts the work distribution according to a work estimate, increases scalability. Even if the adaptive approach is not new, our is the first one that effectively works in an interactive context, performing balancing decision at each computational step. This is possible because the proposed balancing technique is totally distributed, and do not perform complicated compute-intensive calculation. Hence, the resulting approach is fast to compute. Moreover, our adaptive approach is based on accurate predictive workload estimation (i.e. the number of agents in a given volume). We showed that our approach overtakes in performance several previous works based on expensive and elaborate centralized computations, which usually implies a global synchronization.

Adaptive load balancing algorithm is a suitable solution even for parallel computing and interactive context

For a long time, in context like Grid Computing researchers exploited adaptive load balancing schema in their works. However, in Parallel Computing often they are not considered an effective solution because of the high computation time usually involved. In this dissertation, we showed several approaches using adaptive technique. In works based on ray tracing and agent-based simulations (Chapter 2 and 4), nevertheless computing time is not high in respect of communication need, overhead easily overcomes the gain in balancing. Even in our work based on cost map (Chapter 3), we showed that a trivial adaptive schema is at least competitive with a dynamic one. We indicated two critical factors that may affect the effectiveness of an adaptive schema:

- First, the time spent in computing the estimation. This includes the time to obtain a prediction, the time spent in synchronization (usually present) and the time to compute the new task assignment. As short it is, as small will be the overhead of the approach.
- The second factor is the accuracy of the estimation. Depending by the problem, we may have accurate prediction based on trivial properties of the data set (e.g. the geometrical agent distribution used in Chapter 4), or we may assume some restriction to have a good enough estimate. The example of the Parallel Ray Tracing comes up in help. In Chapter 2, for instance, we assumed a strong temporal coherence in order to obtain estimation. Instead, in Chapter 3, we showed that a GPU technique does not require temporal coherence between successive frames.

We have to trade-off these two factors in order to exploit both the problem and hardware features.

Work stealing effectiveness strongly depends on task size and task dependencies

Work stealing algorithm is great in balancing workload, but we should point some considerations. In fact, depending by the context, steals could badly affect performance for two reasons:

- A steal is a task transfer, hence it involves in (perhaps expensive) data transfer.
- Random steals may involve tasks that are all but locals, hence badly affecting data locality.

For these reasons, it is not suitable for an application like Agent-based Simulation. Instead, it is perfect for Parallel Ray Tracing. Interestingly, like other dynamic balancing techniques, work-stealing performance strongly depends on task sizes. In Chapter 3, we showed that if load is almost balanced, it works poorly. In the opposite, we obtained best performance when combining work stealing on a regular subdivision with task-sorting based on estimation, i.e. with not balanced workload.

Ease of parallelization does not necessarily mean ease to balance workload.

The problem of Parallel Ray Tracing on a cluster of workstation represents the paradox that embarrassingly parallel problems are not necessarily embarrassingly evenly balanced. In the contrary, the irregularity that may arise pose challenges to balance the workload, requiring not ease balancing techniques. We also showed that Parallel Ray Tracing exhibits different workload characteristics with different techniques (in Chapter 3, Whitted ray tracing's workload is more balanced than Kajiya path tracing one).

Hybrid architectures may offer not-trivial design

The way that we can asset parallel computation on hybrid hardware may be very various. A first approach, for instance, is the one to run the same algorithm on all platforms available, i.e. using the same implementation. A second more efficient approach consists to distribute tasks between platforms as we do in the first approach, hence to use specialized implementations for each platform. This means that the underling algorithm used by the platform can be extremely varying depending by the platform characteristics. A third approach, instead, is to run completely different tasks for different platform architectures. In Chapter 3, we had very specific hybrid architecture, i.e. a cluster of multi core CPUs + a single visualization node equipped with a GPU. For such architecture, we showed an effective parallelization strategy were effective CPU computations are performed by the CPU, meanwhile a GPU spent its computational time to improve overall load balancing by producing a cost map.

A

Listing of test hardware platforms

A.1 HLRS, Universität Stuttgart

Cacau cluster	
Institute	HLRS, High Performance Computing Center Stuttgart (Höchstleistungsrechenzentrum Universität Stuttgart)
Location	Stuttgart (Germany)
Cluster type	NEC Xeon EM64T Cluster
Cluster nodes	64 used
Node characteristic	each equipped with 2 Intel Xeon EM64T processors (dual SMP)
Node main memory	2 GB
Network	Infiniband

A.2 ENEA Supercomputing, Portici

ENEA Portici Sezione 1, Alto Parallelismo	
Institute	Ente per le Nuove Tecnologie, l'Energia e l'Ambiente
Location	Portici (Napoli, Italy)
Cluster type	Cluster IBM x3850-M2
Cluster nodes	42 SMP nodes
Node characteristic	4 Xeon Quad-Core Tigerton E7330 clock 2.4GHz/1066MHz/6MB L2
Node main memory	32/64 GB
Network	Infiniband 4XDDR

ENEA Portici Sezione 2	
Institute	Ente per le Nuove Tecnologie, l'Energia e l'Ambiente
Location	Portici (Napoli, Italy)
Cluster type	Cluster IBM HS21
Cluster nodes	256
Node characteristic	2 Xeon Quad-Core Clovertown E5345 clock 2.33 GHz
Node main memory	16 GB
Network	Infiniband 4DDR

A.3 VISUS Stuttgart

VISUS cluster	
Institute	VISUS, Universität Stuttgart
Location	Stuttgart (Germany)
Cluster type	Cluster of commodity workstations
Cluster nodes	8+1 (used)
Node characteristic	Intel Quadcore clock 3 GH z
Node main memory	2GB
Network	Infiniband

A.4 ISISLab, Università degli Studi di Salerno

Hydra	
Institute	ISISLab, Università degli Studi di Salerno
Location	Salerno (Italy)
Cluster type	IBM HS20
Cluster nodes	32+1 (used)
Node characteristic	Intel Quadcore clock 3.20 GHz
Node main memory	1GB
Network	Gigabit Ethernet
Software	CentOS 5 linux, OpenMPI 1.1.1, Intel compiler

A.5 CINECA Supercomputing, Bologna

IBM SP6	
Institute	CINECA
Location	Bologna (Italy)
Cluster type	IBM pSeries P6-575
Cluster nodes	168 (available)
Node characteristic	IBM Power6, 32 cores clock 4.7 GHz
Node main memory	128 GB
Network	Infiniband x4 DDR
Software	OS AIX 6, MPI, OpenMP, IBM compiler

B

List of Related Publications

Parts of this dissertation have already been published in several publications. The following is a list of papers, journals and technical reports that have contributed to this thesis, and that might contain additional informations. Some of these publications by now are significantly older than the thesis, and thus may contain information (such as performance data) that is already outdated. Technical reports have been made as a summary articles for research projects (e.g. HPC-Europa projects). Their data and results are included in more recent papers or journals, where these results are discussed in deep.

[23] Gennaro Cordasco, Biagio Cosenza, Rosario De Chiara, Ugo Erra, and Vittorio Scarano.
Experiences with Mesh-like computations using Prediction Binary Trees.
Scalable Computing: Practice and Experience, Scientific International Journal for Parallel and Distributed Computing (SCPE), 10(2):173187, June 2009.

[30] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, Ugo Erra, and Vittorio Scarano.
Load balancing in mesh-like computations using prediction binary trees.
In ISPDC, pages 139146, 2008.

[25] Biagio Cosenza.
A Survey on Exploiting Grids for Ray Tracing.
In Eurographics Italian Chapter Conference, pages 8996, 2008.

[31] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, Ugo Erra, and Vittorio Scarano.
On Estimating the Effectiveness of Temporal and Spatial Coherence in Parallel Ray Tracing.
In Eurographics Italian Chapter Conference, pages 97104, 2008.

[27] Biagio Cosenza.
Load Balancing Techniques for Parallel Ray Tracing, 2008.
Poster at HPC-Europa++ TAM-Workshop 2008, presented on 15-17/12/08 at HLRS Supercomputing Center, Universität Stuttgart.

[28] Biagio Cosenza.
Synergy Effects of Hybrid CPU-GPU architectures for Interactive Parallel Ray Tracing.
Science and Supercomputing in Europe, Research Highlights 2009. HPC-Europa2 Technical Reports ISBN 978-88-86037-23-5, CINECA, 2009. ISBN 978-88-86037-23-5.

[26] Biagio Cosenza.
Evaluation of Adaptive Subdivision Schemas for Parallel Ray Tracing.
HPC-Europa: Science and Supercomputing in Europe, Technical Reports 2008 ISBN 978-88-86037-22-8, 2008. ISBN 978-88-86037-22-8.

[33] Biagio Cosenza, Carsten Dachsbacher, and Ugo Erra.
GPU Cost Estimation for Load Balancing in Parallel Ray Tracing.
(Submitted for journal publication)

[32] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, and Vittorio Scarano.
Distributed Load Balancing for Parallel Agent-based Simulations.
In PDP2011 - 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, 2011.

Bibliography

[1] Timothy Davis Alan Chalmers and Erik Reinhard. *Pratical Parallel Rendering*. AKPeters, 2002.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[3] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] F. Baiardi, P. Becuzzi, P. Mori, and M. Paoli. Load balancing and locality in hierarchical n-body algorithms on distributed memory architectures. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 284–293. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0037155.

[5] Ioana Banicescu and Susan Flynn Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *In Proceedings of Supercomputing95*, 1995.

[6] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. 22(3), 2003.

[7] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32(5):1260–1279, 2003.

[8] J. Bigler, A. Stephens, and S.G. Parker. Design for parallel interactive ray tracing systems. *IEEE Symposium on Interactive Ray Tracing*, 0:187–196, 2006.

[9] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[11] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, 1994.

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of ACM*, 46(5):720–748, 1999.

[13] C. Boeres and V. Rebello. Cluster-based static scheduling: Theory and practice. In *SBAC-PAD '02: Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing (SCAB-PAD'02)*, page 133, Washington, DC, USA, 2002. IEEE Computer Society.

[14] David A. Carlson. Using local memory to boost the performance of fft algorithms on the cray-2 supercomputer. *The Journal of Supercomputing*, 4:345–356, 1991. 10.1007/BF00129835.

[15] Loren Carpenter. The a -buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph.*, 18(3):103–108, 1984.

[16] Alan Chalmers, Kurt Debattista, Veronica Sundstedt, Peter Longhurst, and Richard Gillibrand. Rendering on Demand. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2006.

[17] Alan Chalmers and Erik Reinhard, editors. *Practical Parallel Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[18] J. Chapman, T. W. Calvert, and J. Dill. Exploiting temporal coherence in ray tracing. In *Proceedings on Graphics interface '90*, pages 196–204, Toronto, Canada, 1990.

[19] Bernard Chazelle. Natural algorithms. In *SODA '09: Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 422–431, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[20] Long Chen, Junmin Lin, and Guang R. Gao. Optimizing fast fourier transform on a multi-core architecture. In *in IEEE International Parallel and Distributed Processing Symposium*, 2007.

[21] Jaeyoung Choi and J.J. Dongarra. Scalable linear algebra software libraries for distributed memory concurrent computers. In *Distributed Computing Systems, 1995., Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of*, pages 170 –177, August 1995.

[22] Gennaro Cordasco, Rosario De Chiara, Ugo Erra, and Vittorio Scarano. Some considerations on the design of a p2p infrastructure for massive simulations. In *International Conference on Ultra Modern Telecommunications (ICUMT '09)*, October 2009.

[23] Gennaro Cordasco, Biagio Cosenza, Rosario De Chiara, Ugo Erra, and Vittorio Scarano. Experiences with Mesh-like computations using Prediction Binary Trees. *Scalable Computing: Practice and Experience, Scientific international journal for parallel and distributed computing (SCPE)*, 10(2):173–187, June 2009.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[25] Biagio Cosenza. A Survey on Exploiting Grids for Ray Tracing. In *Eurographics Italian Chapter Conference*, pages 89–96, 2008.

[26] Biagio Cosenza. Evaluation of adaptive subdivision schemas for parallel ray tracing. HPC-Europa: Science and Supercomputing in Europe, Technical Reports 2008 ISBN 978-88-86037-22-8, 2008. ISBN 978-88-86037-22-8.

[27] Biagio Cosenza. Load Balancing Techniques for Parallel Ray Tracing, 2008. Poster at HPC-Europa++ TAM-Workshop 2008, presented on 15-17/12/08 at HLRS Supercomputing Center, Universität Stuttgart.

[28] Biagio Cosenza. Synergy effects of hybrid cpu-gpu architectures for interactive parallel ray tracing. Science and Supercomputing in Europe, Research Highlights 2009. HPC-Europa2 Technical Reports ISBN 978-88-86037-23-5, CINECA, 2009. ISBN 978-88-86037-23-5.

[29] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, Ugo Erra, and Vittorio Scarano. Load balancing in mesh-like computations using prediction binary trees. In *7th International Symposium on Parallel and Distributed Computing*, pages 139–146, 2008.

[30] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, Ugo Erra, and Vittorio Scarano. Load balancing in mesh-like computations using prediction binary trees. In *ISPDC*, pages 139–146, 2008.

[31] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, Ugo Erra, and Vittorio Scarano. On Estimating the Effectiveness of Temporal and Spatial Coherence in Parallel Ray Tracing. In *Eurographics Italian Chapter Conference*, pages 97–104, 2008.

[32] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, and Vittorio Scarano. Distributed load balancing for parallel agent-based simulations. In *PDP2011 - 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2011.

[33] Biagio Cosenza, Carsten Dachsbacher, and Ugo Erra. [Submitted for journal publication] GPU Cost Estimation for Load Balancing in Parallel Ray Tracing.

[34] Franklin C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH '84 Conference Proceedings*, pages 207–212, New York, NY, USA, 1984. ACM.

[35] David E. DeMarle, Christiaan Gribble, and Steven G. Parker. Memory-savvy distributed interactive ray tracing. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 93–100, June 2004.

[36] David E. DeMarle, Christiaan P. Gribble, Solomon Boulos, and Steven G. Parker. Memory sharing for interactive ray tracing on clusters. *Parallel Computing*, 31(2):221–242, 2005.

[37] David E. DeMarle, Christiaan P. Gribble, Solomon Boulos, and Steven G. Parker. Memory sharing for interactive ray tracing on clusters. *Parallel Computing*, 31(2):221–242, 2005.

[38] David E DeMarle, Steven Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed interactive ray tracing for large volume visualization. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 87–94, October 2003.

[39] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[40] K. D. Devine, J. E. Flaherty, S. R. Wheat, and A. B. Maccabe. A massively parallel adaptive finite element method with dynamic load balancing. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 2–11, New York, NY, USA, 1993. ACM.

[41] A. Dietrich, A. Stephens, and I. Wald. Exploring a Boeing 777: Ray Tracing Large-Scale CAD Data. *IEEE Computer Graphics and Applications*, 27-6:36–46, 2007.

[42] Jack J. Dongarra, Robert A. van de Geijn, and David W. Walker. Scalability Issues Affecting the Design of a Dense Linear Algebra Library. *Journal of Parallel and Distributed Computing*, 22:523–537, 1994.

[43] Philip Dutré. Global Illumination Compendium.

[44] Philip Dutré, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[45] Ugo Erra, Rosario De Chiara, Vittorio Scarano, and Maurizio Tatafiore. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In *Proceedings of Vision, Modeling and Visualization 2004 (VMV)*, November 2004.

[46] Florian Fleissner and Peter Eberhard. Load Balanced Parallel Simulation of Particle-Fluid DEM-SPH Systems with Moving Boundaries. In *Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the International Conference ParCo 2007*. IOS Press, 2007.

[47] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sep. 1972.

[48] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kaufmann, May 1994.

[49] Iliyan Georgiev and Philipp Slusallek. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing*, 2008.

[50] R. Gillibrand, P. Longhurst, K. Debattista, and A. Chalmers. Cost Prediction for Global Illumination Using a Fast Rasterised Scene Preview. In *Proceedings of AFRIGRAPH '06*, 2006.

[51] Andrew S. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[52] John L. Gustafson. Reevaluating amdahl's law. *Communication of ACM*, 31(5):532–533, 1988.

[53] Shawn Hargraves. Deferred shading. In *Game Developers Conference, Talks*, 2004.

[54] Alan Heirich and James Arvo. A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing*, 12(1-2):57–68, 1998.

[55] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.

[56] Justin Hensley, Thorsten Scheuermann, Greg Coombe, and Montek Singh. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24:547–555(9), 2005.

[57] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing: Technology,Architecture,Programming*. McGraw-Hill, Inc., New York, NY, USA, 1998.

[58] Intel Software Network. Many-core processor definition from the Glossary of Technical Terms.

[59] Intel Software Network. Multi-core processor definition from the Glossary of Technical Terms.

[60] ISISLab. Distributded Steer Video.

[61] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling Hierarchical N-body Simulations on GPU Clusters. In *Proceedings of the ACM/IEEE Supercomputing Conference 2010 (to appear)*, 2010.

[62] S. L. Johnsson, R. L. Krawitz, R. Frye, and D. MacDonald. A radix-2 fft on connection machine. *SC Conference*, 0:809–819, 1989.

[63] Bjoern Knafla and Claudia Leopold. Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP. In *Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the International Conference ParCo 2007*. IOS Press, 2007.

[64] Sergey Kopyssov and Alexander Novikov. Parallel adaptive mesh refinement with load balancing for finite element method. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 2127 of *Lecture Notes in Computer Science*, pages 266–276. Springer Berlin / Heidelberg, 2001.

[65] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[66] Christian Lauterbach, Micheal Garland, Shubahbratav Sengupta, Davide Luebke, and Dinesh Manocha. Fast BVH Construction on GPUs. In *Proceedings of Eurographics*, 2009.

[67] Xiaoye S. Li and James W. Demmel. SuperLU DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[68] R. Lling, B. Monien, and F. Ramme. Load balancing in large networks: A comparative study (extended abstract). In *In Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 686–689, 1991.

[69] Message Passing Interface Forum. The Message Passing Interface (MPI) standard.

[70] Gordon E. Moore. Cramming more components onto integrated circuits, April 1965.

[71] Gordon E. Moore. No exponential is forever: but "Forever" can be delayed! In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, volume 1, pages 20–23, 2003.

[72] Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*, 1995.

[73] John Nagle, 1984. RFC 896. Congestion control in IP/TCP internetworks.

[74] John Nagle. Rfc 896: Congestion control in ip/tcp internerworks, 1984.

[75] Rahul Narain, Abhinav Golas, Sean Curtis, and Ming C. Lin. Aggregate dynamics for dense crowd simulation. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–8, New York, NY, USA, 2009. ACM.

[76] D. M. Nicol and Joel H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transaction on Computer*, 37(9):1073–1087, 1988.

[77] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. Large ray packets for real-time whitted ray tracing. *IEEE Symposium on Interactive Ray Tracing*, pages 41–48, 2008.

[78] Manish Parashar and James C. Browne. Distributed dynamic data-structures for parallel adaptive meshrefinement. In *Proceedings of the International Conference on High Performance Computing*, 1995.

[79] Manish Parashar and James C. Browne. On partitioning dynamic adaptive grid hierarchies. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*. IEEE Computer Society, 1996.

[80] Gyung-Leen Park, Behrooz Shirazi, and Jeff Marquis. Mapping of parallel tasks to multiprocessors with duplication. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7*, page 96, Washington, DC, USA, 1998. IEEE Computer Society.

[81] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[82] David Patterson. Latency lags bandwidth. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 3–6, Washington, DC, USA, 2005. IEEE Computer Society.

[83] Tomas Plachetka. Perfect load balancing for demand-driven parallel ray tracing. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 410–419, London, UK, 2002. Springer-Verlag.

[84] Michael J. Quinn, Ronald A. Metoyer, and Katharine Hunterzaworski. Parallel implementation of the social forces model. In *in Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics*, pages 63–74, 2003.

[85] William T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 359–375, New York, NY, USA, 1983. ACM.

[86] Erik Reinhard and Frederik W. Jansen. Rendering large scenes using parallel ray tracing. *Parallel Computing*, 23:67–80, 1997.

[87] Erik Reinhard, Arjan J. F. Kok, and Alan Chalmers. Cost distribution prediction for parallel ray tracing. In *Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 77–90. Eurographics, September 1998.

[88] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05 Conference Proceedings*, pages 1176–1185, New York, NY, USA, 2005. ACM.

[89] Craig Reynolds. OpenSteer, Steering Behaviors for Autonomous Characters.

[90] Craig Reynolds. Big fast crowds on ps3. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121, New York, NY, USA, 2006. ACM.

[91] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM.

[92] Anthony Steed Roula, Anthony Steed, and Roula Abou-haidar. Partitioning crowded virtual environments. In *In Proceedings of the ACM symposium on Virtual reality software and technology*, pages 7–14, 2003.

[93] Plimpton S. Fast parallel algorithms for short range molecular dynamics. *Journal of Computational Physics*, 117 n.1:1–19, 1995.

[94] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2003.

[95] Abe Stephens, Solomon Boulos, James Bigler, Ingo Wald, and Steven Parker. An Application of Scalable Massive Model Interaction using Shared-Memory Systems . In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 19–26, 2006.

[96] Kevin Suffern. *Ray Tracing from the Ground Up*. A. K. Peters, Ltd., Natick, MA, USA, 2007.

[97] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1160–1168, New York, NY, USA, 2006. ACM.

[98] G. Vigueras, M. Lozano, J. M. Ordu na, and F. Grimaldo. A comparative study of partitioning methods for crowd simulations. *Appl. Soft Comput.*, 10(1):225–235, 2010.

[99] Ingo Wald. Realtime Ray Tracing and Interactive Global Illumination. *PhD thesis, Saarland University*, 2004.

[100] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[101] Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek. Interactive Distributed Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. *In Proceedings of EuroPar '03, Lecture Notes on Computer Science*, 2790:499–508, 2003.

[102] Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek. Interactive Distributed Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. In *Proceedings of EuroPar*, volume 2790, pages 499–508, 2003.

[103] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An interactive out-of-core rendering framework for visualizing massive complex models. In *Proceedings of the Eurographics Symposium on Rendering*, pages 81–92, 2004.

[104] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006.

[105] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 277–288, London, UK, 2001. Springer-Verlag.

[106] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 1980.

[107] Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *in International Conference on Parallel and Distributed Computing Systems (IPDPS*, 2008.

[108] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[109] Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors. *Parallel Processing and Applied Mathematics, 5th International Conference, PPAM 2003, Czestochowa, Poland, September 7-10, 2003. Revised Papers*, volume 3019 of *Lecture Notes in Computer Science*. Springer, 2004.

[110] Lingyun Yang, Jennifer M. Schopf, and Ian Foster. Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 31. IEEE Computer Society, 2003.

[111] Tao Yang and Apostolos Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1994.

[112] Yongpeng Zhang, Frank Mueller, Xiaohui Cui, and Thomas Potok. Large-Scale Multi-Dimensional Document Clustering on GPU Clusters. In *IEEE International Parallel and Distributed Processing Symposium*, 2010.

[113] Bo Zhou and Suiping Zhou. Parallel simulation of group behaviors. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 364–370. Winter Simulation Conference, 2004.

[114] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transaction on Graphics*, 27(5):1–11, 2008.

Le persone viaggiano per stupirsi delle montagne,
dei mari, dei fiumi, delle stelle;
e passano accanto a se stesse senza provare meraviglia.
...
Sant'Agostino d'Ippona