

# Parallel Raytracing: A Case Study on Partitioning and Scheduling on Workstation Clusters

Bernd Freisleben, Dieter Hartmann, Thilo Kielmann

*Department of Electrical Engineering and Computer Science, University of Siegen*

*Hölderlinstr. 3, D-57068 Siegen, Germany*

*{freisleb,dieter,kielmann}@informatik.uni-siegen.de*

## Abstract

*In this paper, a case study is presented which is aimed at investigating the performance of several parallel versions of the POV-Ray raytracing package implemented on a workstation cluster using the MPI message passing library. Based on a manager/worker scheme, variants of workload partitioning and message scheduling strategies, in conjunction with different task granularities, are evaluated with respect to their runtime behaviour. The results indicate that dynamic, adaptive strategies are required to cope with both the unbalanced workload characteristics of the parallel raytracing application and the different computational capabilities of the machines in a workstation cluster environment.*

## 1 Introduction

Raytracing [9, 13, 24] is a widely used method for generating realistically looking images on a computer, and it is employed by many 3D modelling and animation systems for the final image rendering. The input to a raytracing algorithm is the *scene* – the description of the geometry of 3D objects and the definition of the objects' materials, the lights, and the imaginary camera. The output is the image of the scene, i.e. the colour of each pixel of the image (also called a *frame*), as seen by the defined camera.

In order to compute the colour of the pixel  $(x, y)$ , a ray is casted from the camera through the frame at the position  $(x, y)$  into the scene and the intersection with the first object is determined (if it exists). Based on the position of the intersection point, the surface normal at this point, the position and colour of lights and the material of the intersected surface, the light intensity and colour at the intersection point are computed. Then, the ray is reflected and/or refracted according to the reflectivity and transparency of the surface, and the process is repeated recursively with the reflected/refracted rays, adding the light intensities at all intersection points to get the final colour of

the traversed pixel in the frame. The recursive process stops when the light contribution gets below a threshold, or at a user-defined depth of recursion.

It is quite obvious that raytracing algorithms require a considerable amount of computing power, since they involve a large number of floating-point calculations for mathematically modelling the light rays as they bounce around the scene until they end up in the lense of the imaginary camera. The required computing power increases strongly when very complex scenes need to be visualized; computation times of several weeks on high performance workstations are not unusual for particular applications [17].

A natural way to reduce the computation times is to perform raytracing in parallel on a multi-processor machine [10, 18, 19]. Two types of parallelism are quite evident when generating raytraced images. The first is a domain decomposition of the input data, i.e. subsets of the objects of a scene are assigned to and processed by different processors. The second is a domain decomposition of the output to be produced, i.e. the pixels of the image are distributed to different processors and coloured independently. The first (data parallel) approach requires a possibly large amount of communication during raytracing, because information about other subsets of objects may be needed to perform the computations. In the second (task parallel) approach, each processor must have the complete scene description to be able to have access to all the information about the rays involved in determining the colour of a pixel. The possibly large memory requirements for replicating the scene data in each processor is one of the reasons why most of the current proposals for parallel raytracing algorithms [15, 16, 23, 25] follow the first approach. Nevertheless, the second approach is an ideal candidate for achieving high speedups, because raytracing parallelized in this way constitutes an *embarrassingly parallel* problem [7], i.e. the amount of inter-process communication is reduced to a minimum,

and the data replication problem may be somewhat alleviated by using a distributed shared memory architecture [3]. The second approach is investigated in this paper, since it allows to use raytracing as a suitable testbed application for studying the behaviour of different partitioning and scheduling techniques for embarrassingly parallel problems.

Such problems are typically approached by a *manager/worker* [6, 7] scheme in which a central manager process is given responsibility for work distribution and results processing. There are several worker processes each of which gets assigned or repeatedly requests a subproblem from the manager, solves it and sends the results back to the manager. The main communication occurs during the distribution of the work before the computation and in the receipt of the results afterwards.

Although the implementation of a manager/worker scheme might seem straightforward at first glance, it is nevertheless particularly challenging in the context of a raytracing application. The main reason is that each ray will take a different amount of processor time to compute, depending on the complexity of the intersections that occur. If a big region of the frame is distributed to each processor, this will lead to load balancing problems if, for example, several tasks are responsible for processing the rays that do not intersect with the scene, while a single one must unfortunately do all the work. Since it is impossible to predict the complexity of the intersections for a given part of the frame, the only reasonable solution is to distribute the work dynamically, and adaptively for every scene, among the workers.

In addition to the workload aspect of raytracing, the implementation of an efficient manager/worker scheme becomes particularly more complicated when the implementation platform is a workstation cluster [2, 14, 22]. Unlike dedicated parallel computers, the individual machines in a network of workstations are heterogeneous with respect to their hardware architectures, their processing power, and they usually also have significantly different main memory sizes. Furthermore, the type of interconnection network and network protocol (e.g. Ethernet, FDDI or ATM [21]) has a large impact on the communication speed, and the workload caused by other workstation users influence the runtime behaviour of parallel applications. Therefore, due to the heterogeneity of both the computational demands of a parallel raytracing approach and the computational capabilities of the workstations in a network, the aim of any attempt of realizing a manager/worker scheme must be to use the available

resources, the workstations and the network, as efficiently as possible, i.e. the workload must be distributed dynamically and adaptively.

In this paper, several dynamic and adaptive manager/worker schemes for parallel raytracing are investigated. The sequential source code of the well known *Persistence of Vision* (POV-Ray) raytracing package [5] has been extended to implement the various schemes on a cluster of DEC Alpha workstations, using the features provided by the *Message Passing Interface* (MPI) communications library [11]. A total of 16 different workload partitioning/message scheduling methods and their impact on the runtimes are analyzed. The results indicate that particular methods are clearly more favourable than others when the size of the frames and the complexity of the scenes increase. The impact of the underlying networking technology, studied by comparing Ethernet and FDDI, is quite low in this application.

The paper is organized as follows. Section 2 describes the parallel MPI implementation of POV-Ray in the Alpha cluster. Section 3 presents the different workload partitioning and message scheduling strategies examined. In section 4, the test setup and the results obtained in an Ethernet environment are presented. Section 5 discusses the impact of an FDDI interconnection on the results. Section 6 concludes the paper and outlines areas for future research.

## 2 MPI POV-Ray

As already mentioned, the work reported in this paper is based on the widely known POV-Ray raytracing package [5], of which we used version 2.2. In our parallelized implementation, the MPI [11] message passing interface is used as the communications library which has by now been established as the de-facto standard for message passing programs. In particular, the MPICH [1] implementation in version 1.0.11 is used, and the implementation language is *C*. The hardware platform available at our site is a cluster of Digital Equipment Alpha workstations of various kinds, running Digital UNIX (formerly OSF/1) in release 3.2. The machines are connected via an Ethernet segment and (some of them) additionally by an FDDI crossbar switch (DEC GigaSwitch/FDDI).

The feasibility of parallelizing POV-Ray for clusters of workstations has already been demonstrated by the PVMPOV package [4] which uses PVM [8] as the communication platform. This package is based on the task-parallel approach using a single partitioning strategy by distributing workload in form of small squares whose size can be selected by the

user. Unfortunately, performance results for PVM-POV have been reported for multi-processor shared-memory machines only, so we cannot compare them directly to our measurements presented in Section 4.

For our MPI version, the original sequential POV-Ray code has been rewritten by dividing it into a manager and a worker program, also following the task-parallel approach. The worker program contains most of the original code whereas the manager implements the control over the worker processes. The manager program interprets a set of command line directives controlling the various kinds of partitioning and message scheduling we have implemented. These are outlined in detail in Section 3. Because all workstations in our cluster have only single processors, we further assume that every workstation will be assigned exactly one process of a parallel computation in order to maximize application speed. We ensure this by using MPICH's startup mechanism.

In a very coarse-grained view, the sequential POV-Ray program performs three steps: it first parses the scene data file, then it produces the corresponding trace image, and finally it generates statistics data about the raytracing process. The parallel version has been designed to exploit as much of the available concurrency as possible. Therefore, the manager program immediately assigns the initial tasks to the available workers. Due to the MPI design philosophy, the manager process knows at program startup how many workers will be available and how to contact them. Hence, a worker registration protocol is not necessary. After distributing the initial tasks, the manager waits for messages from its workers. As soon as a message arrives, it checks whether the message is a request for a new task. In this case, the corresponding worker will be assigned a new task as long as there are tasks to be done. In the case of trace data being sent from a worker, the manager stores it. If there are no more parts of the trace data missing, the manager collects statistics data from its workers and terminates.

The worker processes concurrently parse the scene data and then wait for tasks to compute. Once a worker receives a task description, it produces the corresponding trace data, requests a new task from the manager and finally transmits its image data. In case of line-wise transmission of trace data (see Sect. 3.2), a worker sends image lines already as soon as they are completed. If a message from the manager indicates that the computation is finished, a worker produces the statistics data corresponding to its tasks, sends this data to the manager, and terminates.

Since requesting a new task before sending trace

data increases the overlap of computations between manager and worker and hence improves application speed, this option was always employed in our implementation.

Another design decision is concerned with the communication mode to be used for transmitting trace data. The corresponding design decision has been made as a result of preliminary runtime tests. Since we intend to exploit as much of the available concurrency as possible, our first decision was to choose MPI's asynchronous send mode for worker processes. According to the MPI definition, this mode maximizes the overlap between sender (the worker) and receiver (the manager). Because one design goal of MPI is to guarantee *progress* [12] (no deadlocks due to blocking send operations because of missing buffer storage), the MPICH implementation uses internal message buffers in order to store messages on the receiver side as long as the receiver program has not yet performed corresponding receive operations. Although this kind of message passing introduces a maximal amount of operation overlap between sender and receiver, it unfortunately allows no user control of the allocated buffer size on the receiver side. Consequently, in case of an overloaded receiver, its buffer size will be increased without limitation. Typically, the increase only stops when the virtual memory is exhausted. But as soon as the buffer size increases the receiver process' address space beyond the limit of real memory, page thrashing starts, contributing significantly to receiver overload. Unfortunately, this effect could be observed in our preliminary tests. Considering that the real memory of our machines has a size of 64 MBytes, the amount of image data has to be stored at least two times at the manager side, once in the application buffers and once in MPI's message buffers. This possibly large amount of data, together with the set of (operating system-related) processes running on the workstations, repeatedly overloaded the machine acting as the manager, causing an application slowdown or even aborts due to missing memory.

Consequently, we had to use a different send mode for trace data. We finally have decided to use MPI's synchronous send mode for the workers which causes them to wait until the manager process is ready to receive their messages. This decision clearly reduces the available concurrency between manager and workers, but it made the application feasible at all. As an additional effect, it reduces collisions on the network which otherwise would have been caused by several simultaneously sending workers, because now the manager indirectly controls when a particular worker is allowed

to send its data.

### 3 Partitioning and Message Scheduling Strategies

In workstation clusters, the performance of a parallel application is dominated by two major factors, the processing power of the workstations and the transmission capacity of the interconnection network. In manager/worker settings, a dedicated manager process typically schedules single tasks to idle workers. Since we follow the task-parallel approach, data locality issues do not play a role here; all worker processes are roughly equivalent to each other. Hence, tasks can in principle be distributed in a roundabout fashion. However, in a workstation cluster, processors are inherently heterogeneous, and even similarly equipped machines show varying speed. Furthermore, single machines vary in processing speed during application runtime due to congestion problems concerning real memory and processor time between the application and other (e.g. system) processes. Due to this variability in processing speed and the typical imbalance of load requirements between parts of scenes to be visualized by raytracing, it is important to employ a load distribution strategy in order to achieve good performance. In the absence of data locality considerations, an effective load distribution strategy can only be based on the number of available workers and on the shape and size of the tasks produced by task partitionings.

In order to investigate the impact of different partitioning strategies, we have implemented and tested eight variants. In combination with two message scheduling strategies (which will be discussed in Sect. 3.2), this makes a total of 16 different versions of task partitioning and message scheduling strategies the results of which we report in this work.

#### 3.1 Task Partitioning

In the following, the eight different variants of task partitioning investigated will be outlined. The first variant is a static distribution, shown in Fig. 1. Although static partitioning ignores heterogeneities of workstation power and data-dependent load requirements, we use this version as a reference in order to assess the behaviour of the dynamic partitionings. Because POV-Ray internally uses lines of pixels as basic data structures, we designed our partitioning variants around lines or parts of lines of the image.

Hence, our static partitioning variant simply divides the number of lines of the image by the number of available workers and equally distributes the generated parts between them. In the case when the

number of lines of the image is not a multiple of the number of workers, a few lines remain undistributed. In this case, those lines will be added to the task of the last processor.

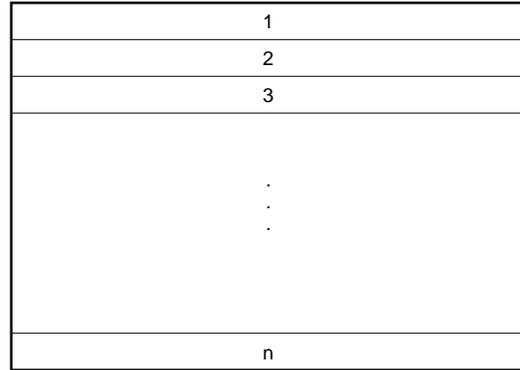


Figure 1: Static task partitioning (strategy 0).

Besides static partitioning, dynamic schemes are of course important in order to evenly distribute workload across available worker processes in order to overcome imbalance effects. Traditional dynamic schemes partition the work into many small, evenly sized tasks.

But before describing the ones we have implemented, we introduce partitioning schemes that can be seen to be in between the dynamic and the static approach. We call this class of partitioning schemes *adaptive* partitionings, because they adapt themselves to the size and shape of the whole task. They do so by recursive subdivisions into smaller parts.

Our first adaptive variant (see Fig. 2) comes close to static partitioning. Here, the whole image is divided into the upper and the lower half of image lines. The upper half is evenly distributed among all worker processes as in strategy 0. The lower half will be dynamically distributed to workers. Each time a worker becomes idle, it gets a new task out of the remaining tasks to be done (in a first-come-first-serve fashion). Therefore, the lower half will be treated as the new image and subdivided into upper and lower half. Its upper half will be distributed to idle workers, whereas the lower half will again be subject to recursive subdivision. This division will take place until a certain minimal size will be reached which will be handed out as a single task.

The main idea behind this strategy is to adapt task sizes to processor speeds. This is achieved by starting with large task sizes and decreasing them in the recursion from step to step. As a result, fast processors will get their next task sooner than slower processors. Hence, fast processors get large tasks, slow processors

get smaller tasks. With this strategy we intend to check whether it is possible to dynamically distribute workload without the drawback of classical schemes which typically create much more tasks than workers available, usually leading to significant communication overheads.

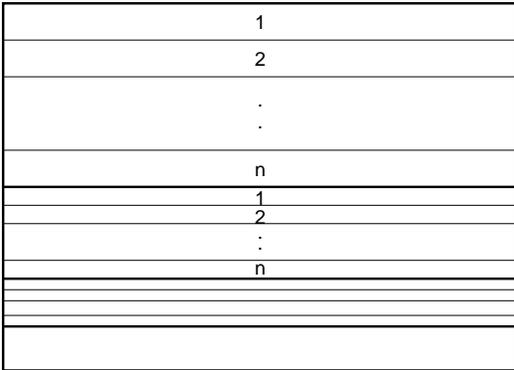


Figure 2: Adaptive task partitioning (strategy 1).

As it can be seen in Fig. 2, the line-oriented recursive division results in several tasks with only a few, but entire lines. A possibly better partitioning strategy might be to recursively divide the image in the currently longer dimension. This is shown in Fig. 3. In this example, the first division halves the size of lines. The left half of the image is divided into evenly sized chunks of lines and distributed among the workers. The right half of the image is recursively subdivided as with strategy 1, but it is alternatingly divided in both dimensions.

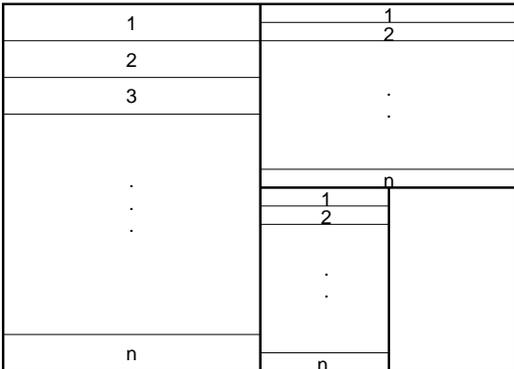


Figure 3: Adaptive task partitioning (strategy 2).

As a variation of strategy 2, the image is recursively divided in alternating directions, but tasks from chunks of columns are formed instead of lines. This is shown in Fig. 4.

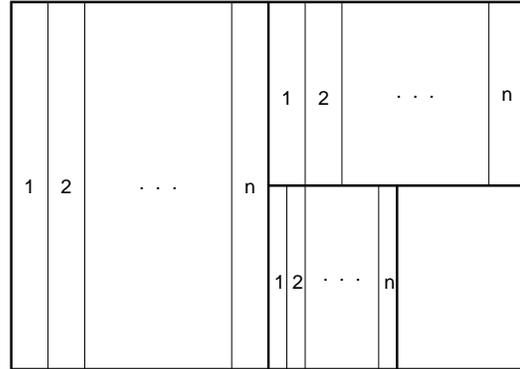


Figure 4: Adaptive task partitioning (strategy 3).

In addition to these adaptive partitioning schemes, we also investigated classic dynamical strategies which are based on producing many small, evenly-sized tasks. One approach, also used with the PVMPOV package [4] is to have tasks consisting of small squares. Two square sizes were investigated, namely  $64 \times 64$  and  $128 \times 128$  pixels. We call these variants partitioning strategies 4 and 5, respectively; they are shown in Fig. 5.

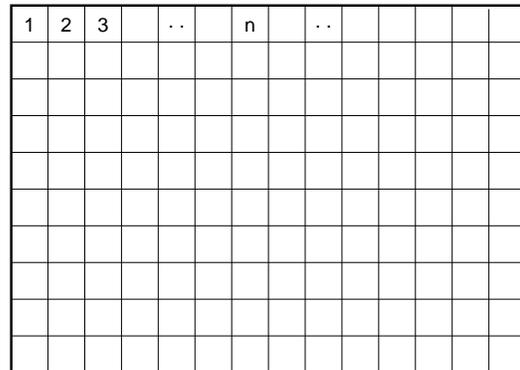


Figure 5: Dynamic task partitioning (strategies 4 and 5).

Besides tasks in the shape of small squares, we also investigated small tasks in the form of chunks of lines (strategy 6) (Fig. 6) and chunks of columns (strategy 7), as shown in Fig. 7. The chunk size investigated was 20 lines or columns, respectively. In case the image could not be divided evenly, the last worker gets assigned more than 20 lines/columns.

### 3.2 Message Scheduling Strategies

Interconnection networks in workstation clusters can be characterized by higher latency and lower transmission rates compared to dedicated supercomputers. Hence, it is important to carefully design com-

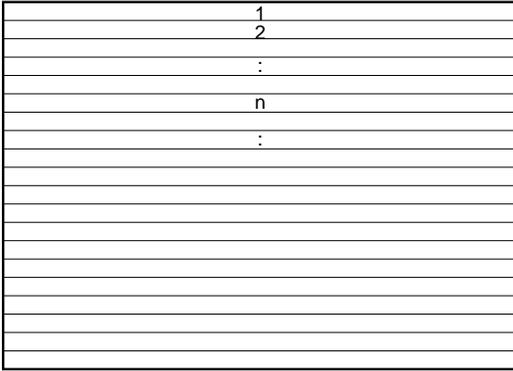


Figure 6: Dynamic task partitioning (strategy 6).

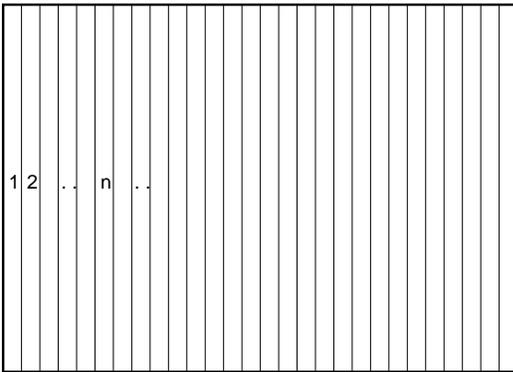


Figure 7: Dynamic task partitioning (strategy 7).

munication between the manager and the workers. In general, high latency for message sending results in the need for reducing the overall number of messages sent during an application run. Thus, fewer but longer messages should be preferred over more but shorter messages. Typically, parallel programs for workstation clusters are designed in order to pack several results into single messages. This design will clearly improve application speedup as long as longer messages will not cause congestion problems in the presence of several concurrent senders.

However, in a manager/worker scenario, workers communicate only with their manager. So, the manager can soon become a bottleneck when it has to process lots of data produced by its workers. In such a case, it might be a better design to increase the overlap between worker tasks and the postprocessing of the results in the manager. In order to do so, results should not be packed into single messages but should be transferred to the manager as soon as they become available.

These two design aspects contradict each other.

Since in our application the generated trace data has sizes between 2 MB and 18 MB per image, we expected the second effect to be relevant. Consequently, we investigated both approaches by combining the eight partitioning strategies with two ways of scheduling the messages containing trace data that are transferred from the workers to the manager. The first version is that a worker sends a line of its task to the manager as soon as the line has been computed completely. We refer to this as version *a*. The second version is to collect trace data of a task at the worker site until the task has been computed completely. Then, all lines of a task are transferred to the manager in one large message. We refer to this as version *b*. Thus, our 16 test settings are referred to as  $0a, 0b, \dots, 7a, 7b$ .

## 4 Test Setup and Results

In order to compare the different strategies, the 16 variants of partitioning and scheduling were applied to three scenes taken from the set of examples provided together with POV-Ray. We used the scenes known as *Fish13*, *Shadows*, and *Skyvase*. Figures 8, 9, and 10 give impressions of these images. All three scenes have been traced in three resolutions:  $320 \times 200$ ,  $640 \times 400$ , and  $960 \times 600$  pixels.

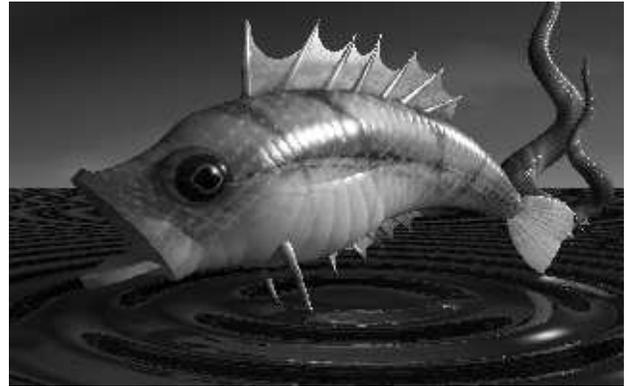


Figure 8: Reference scene *Fish13*

The experiments were conducted on a network of 12 workstations connected via an Ethernet. Since six of them are additionally connected by FDDI lines using Digital's GIGASwitch crossbar switch, we investigated settings with 3, 4, 5, 7, 9, and 11 workers using Ethernet, and 3, 4, and 5 workers using FDDI.

The runtimes of the parallelized POV-Ray were measured in all the combinations described (several times per combination) while our workstation clusters had no additional load by other users, neither in form of processes, nor in form of network load. In the fol-

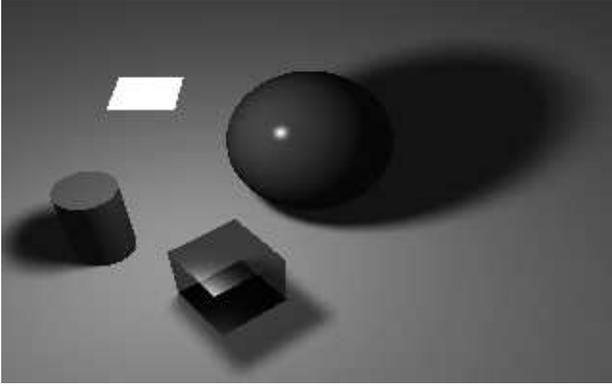


Figure 9: Reference scene *Shadows*



Figure 10: Reference scene *Skyvase*

lowing, we consider the average runtimes of the runs. Our measurements only concern the application kernel, i.e. the time is measured on the manager directly before it starts distributing tasks to the workers and directly after the last task’s results have been received. This kind of measurement was chosen in order to separate the influence of the partitioning and message scheduling from the other aspects of the application. In the remainder of this section, we will only consider our Ethernet results. The reason for this will be explained in the next section.

The large amount of measurements performed in the experiments yielded overwhelmingly many runtime results. This makes it really difficult to present them in an expressive manner. Since we are primarily interested in comparing the effects of our partitioning and message scheduling strategies, we intended to seek invariants across the other variables. We will therefore present figures which relate the strategies to each other. Since we are interested in the fastest strategies, we measure their quality by counting how often they

belong to a group of “best” strategies.

Therefore, we consider sets of measurements for a fixed number of workers and a certain scene in a given resolution. Then, we look up the runtime  $t_f$  of the fastest strategy and compute a threshold value  $t_t = t_f \times 1.1$ . We then form the group of the best strategies from those whose runtimes are within the interval  $[t_f, t_t]$ . Thus, a strategy belongs to the group of best strategies if it is not more than 10% slower than the best one. Hence, this measurement eliminates statistical errors and tells us about strategies that are equivalent to the best ones within a small interval.

We do this for a fixed number of workers for all scenes in all resolutions and count how often (out of nine possibilities) a certain strategy belongs to the group of the best ones. Figure 11 shows the result of this evaluation for the measurements with 11 workers.

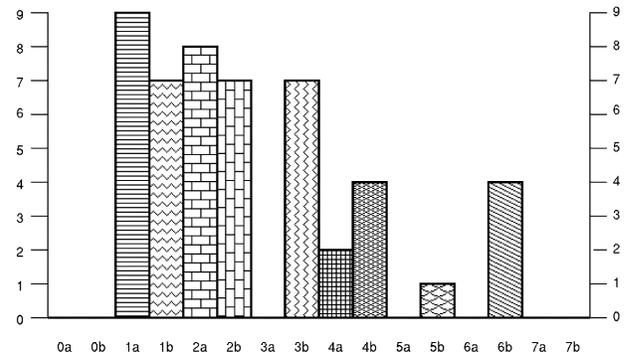


Figure 11: Strategies related to each other for eleven workers.

We will discuss our results based on the configurations with eleven workers, because efficient partitioning and scheduling strategies have their largest impact with large numbers of processes. We will later see that with less workers the effects remain the same, but are a bit harder to detect. In other words, with less worker processes, it is less important how to partition and schedule work among them. Figures 12 and 13 show the corresponding results for seven and three workers, so we can compare results for a small, a moderate, and a large number of workers (within the limits of our workstation cluster.)

Table 1 gives a more detailed picture of the information shown in Fig. 11. Here, for all nine scene/resolution combinations the sequential runtime is compared to the best and worst parallel runtimes with eleven workers. In order to compute speedup values, the heterogeneous processing speed of the workstations must be considered. Therefore, we computed

Table 1: Comparing runtimes of different strategies for 11 workers.

| Scene   | Size      | $T_s$  | Best | $T_p$ | Speedup | Worst | $T_p$ | Speedup |
|---------|-----------|--------|------|-------|---------|-------|-------|---------|
| Fish13  | 320 × 200 | 127.8  | 1b   | 12.5  | 10.2    | 5a    | 35.2  | 3.6     |
| Fish13  | 640 × 400 | 508.7  | 2b   | 42.4  | 12.0    | 7a    | 63.4  | 8.0     |
| Fish13  | 960 × 600 | 1097.4 | 2b   | 91.1  | 12.0    | 0a    | 130.2 | 8.4     |
| Shadows | 320 × 200 | 63.3   | 1a   | 6.4   | 9.9     | 5a    | 18.2  | 3.5     |
| Shadows | 640 × 400 | 243.5  | 1a   | 23.0  | 10.6    | 6a    | 44.7  | 5.4     |
| Shadows | 960 × 600 | 535.2  | 1a   | 53.8  | 9.9     | 6a    | 99.2  | 5.4     |
| Skyvase | 320 × 200 | 68.7   | 1a   | 6.4   | 10.7    | 5b    | 16.5  | 4.2     |
| Skyvase | 640 × 400 | 278.7  | 2a   | 23.8  | 11.7    | 5b    | 41.0  | 6.8     |
| Skyvase | 960 × 600 | 573.3  | 2a   | 51.8  | 11.1    | 6a    | 98.5  | 5.8     |

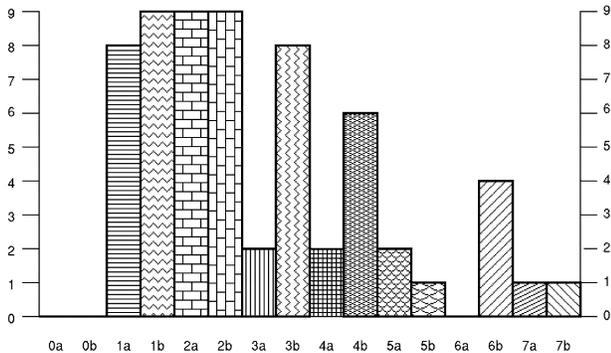


Figure 12: Strategies related to each other for seven workers.

a relative speed index for all machines involved in this setup, where we assigned the value 1.0 to the manager process. All machines are within a range of  $[0.9, 2.0]$ . The sum of relative processing speeds in our configuration with 11 workers is 14.1 which would hence be the speedup value for a perfectly linear speedup.

Our measurement results clearly give evidence for the superiority of our adaptive partitioning schemes compared to static or classical dynamic partitionings. The schemes 1a – 3b are seven to nine times among the set of best strategies, except for 3a which produces huge amounts of tiny messages which compensate the benefits of adaptive partitioning. The static partitioning never belongs to the best strategies, and some classical dynamic schemes show sometimes, but significantly less often than adaptive schemes, performance coming close to the best results. The very best results are always produced by adaptive strategies. Strategies 1 and 2 are almost equivalent, whereas strategy 3 suffers from the data structure mismatch between line-wise organization in POV-Ray and the column-wise partitioning. Nevertheless, 3b is still significantly

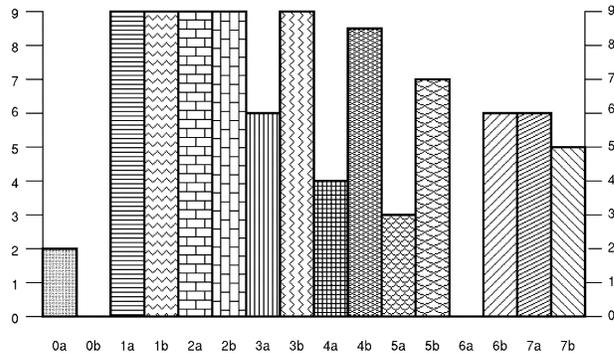


Figure 13: Strategies related to each other for three workers.

better than classical dynamic partitionings. Surprisingly, the static partitioning never (except in one case) produces the worst parallel runtimes.

Another interesting aspect of the results is that – except for the scene *Fish13* – the message scheduling strategy which sends the results line-wise to the manager produces the best results, and hence outperforms its block-wise sending counterpart. This clearly indicates that overlapping computation between manager and workers compensates the overheads introduced by the larger number of messages. In the case of *Fish13*, almost all parts of the scene have high and almost equal computational load requirements, leading to congestion effects on the manager side when several workers almost simultaneously want to transmit their results. Hence, the fewer messages generated by the block-wise sending mode lead to less manager load and thus to better runtimes.

## 5 Network Impact

As explained in the previous section, we have performed our measurements on all available equipment,

including an Ethernet as well as an FDDI network. Because only six of our workstations are connected to both Ethernet and FDDI, we have to restrict our comparisons to configurations with up to five workers.

At first glance, comparisons between both networks are simply disillusioning. Fig. 14 shows runtimes for *Fish13* ( $960 \times 600$  pixels) with five workers compared on Ethernet vs. FDDI. Although FDDI's network capacity is with 100 MBit/sec ten times as fast as Ethernet, runtime results are close to each other. More precisely, FDDI runs show average runtimes that are about 8 seconds better than the same runs performed over Ethernet.

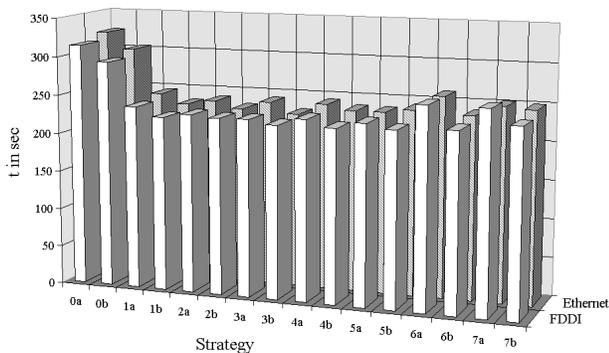


Figure 14: *Fish13* ( $960 \times 600$  pixels) with five workers compared on Ethernet vs. FDDI

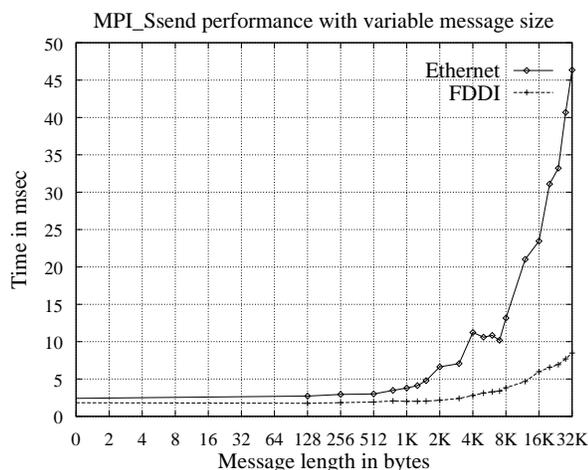


Figure 15: MPI\_Ssend performance compared on Ethernet vs. FDDI

The explanation for this behaviour can be found in Fig. 15 which provides runtime measurements for the plain send operation we used from the MPI package.

These measurements have been performed as ping-pong tests in order to eliminate effects of overlapping executions on the sender and receiver sites [20]. The figure shows that the message latency is almost identical on Ethernet and on FDDI, indicating the high software-related overhead. Up to message lengths of about 1 KBytes, transmission time is nearly the same, too. Message lengths of about 32 KBytes have been used in the better strategies shown in Fig. 14, so we can adequately calculate the network impact based on network throughput achievable with messages of this size. On Ethernet, MPI\_Ssend (MPI's synchronous send) takes 46.4 msec, resulting in a throughput of about 700 KBytes/sec. FDDI takes 8.5 msec, yielding a throughput of about 3.8 MBytes/sec.

The trace data generated in the runs shown in Fig. 14 has a size of about 18 MBytes. Since Ethernet is a bus-based system, trace data has to be sent to the manager sequentially. Based on the measured throughput of 700 KBytes/sec, Ethernet needs about 25 seconds to transfer the trace data to the manager. With FDDI's 3.8 MBytes/sec, the transmission time is only about 5 seconds. Although this is just 20% of the Ethernet transmission time, 25 seconds are only 10.8% of the total runtime of 230 seconds of the best partitioning/scheduling strategy run over the Ethernet. Because all trace data has to be sent to a single process (the manager), our FDDI network cannot make any use of the crossbar switch that provides the full speed of 100 MBit/sec simultaneously between any pair of connected machines. Hence, in this case the FDDI network is only as fast as a simple FDDI ring where data is transmitted sequentially. If we compare only the plain transmission times, the FDDI runs could theoretically be at most 20 seconds faster than our Ethernet runs. Since our measured configurations gain in average 8 out of 20 seconds, we come close to the theoretical maximum. Unfortunately, fast network connections do not seem to have a significant influence on parallel application performance in embarrassingly parallel problems approached by manager/worker schemes. But nevertheless, Fig. 14 again gives evidence to our observations made on Ethernet runs in which the adaptive partitioning strategies are superior to the other ones.

## 6 Conclusion

In this paper, different partitioning and scheduling strategies for implementing manager/worker schemes on clusters of workstations were investigated. Ray-tracing was used as the testbed application, because its task-parallel parallelization constitutes an embar-

rassingly parallel problem in which runtime behaviour can be studied without interferences of complex inter-process communication patterns. In the experiments conducted, 16 different strategies were evaluated. The measured runtime results indicate that static as well as widely known dynamic partitioning schemes perform worse than newly introduced adaptive schemes which automatically assign larger tasks to faster processors and smaller tasks to slower processors without any knowledge of the processors or the computational complexity of the tasks. Furthermore, it was shown that message scheduling is an important design issue influencing application speed on workstation clusters, but the underlying network (Ethernet or FDDI) does not have a significant impact on the application runtime in manager/worker implementations of embarrassingly parallel problems.

Based on the experiences gained in this work, we are currently investigating how to improve our results. An interesting approach is to fine-tune the adaptive algorithms introduced with different division ratios or variations of initial task sizes. Another promising approach would be to introduce self-learning partitioning and scheduling strategies by employing techniques from fuzzy logic and neural networks [26].

## References

- [1] Argonne National Laboratory. MPICH implementation home page. <http://www.mcs.anl.gov/Projects/mpi/mpich/index.html>, 1995.
- [2] M.J. Atallah, C.L. Black, D.C. Marinescu, H.J. Siegel, and T.L. Casavant. Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations. *Journal of Parallel and Distributed Computing*, 16(4):319–327, 1992.
- [3] D. Badouel, K. Bouatouch and T. Priol. Distributed Data and Control for Ray Tracing in Parallel. *IEEE Computer Graphics & Applications*, 14(4):69–77, 1994.
- [4] A. Dilger. PVM Patch for POV-Ray, Home Page. <http://www.mddsp.enl.ucalgary.ca/People/adilger/povray/pvmpov.html>, 1995.
- [5] A. Enzmann, L. Kretzschmar, and C. Young. *Ray Tracing Worlds with POV-Ray*. The Waite Group, 1994.
- [6] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
- [7] G. Fox, R. Williams and P. Messina. *Parallel Computing Works!*. Morgan Kaufmann, 1994.
- [8] G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.
- [9] A. Glassner (Editor). *An Introduction to Ray Tracing*. Academic Press, 1989.
- [10] I. J. Grimstead and S. Hurley. Accelerated Ray Tracing Using an NCUBE2 Multicomputer. *Concurrency: Practice and Experience*, 2(6):571–586, 1995.
- [11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1995.
- [12] W. Gropp and E. Lusk. MPICH Working Note: Creating a New MPICH device using the Channel Interface. Technical Report ANL/MCS-TM-000, Argonne National Laboratory, 1995.
- [13] A. Hermida. *Adventures in Ray Tracing*. Que Corporation, 1993.
- [14] D. Judd, N.K. Ratha, P.K. McKinley, J. Weng, and A.K. Jain. Parallel Implementation of Vision Algorithms on Workstation Clusters. In: *Proceedings of the International Conference on Pattern Recognition*, pp. 317–321, Jerusalem, Israel, 1994.
- [15] H.-J. Kim and C.-M. Kyung. A New Parallel Ray-Tracing System Based on Object Decomposition. *The Visual Computer*, 14:244–253, 1996.
- [16] K.-L. Ma, J.S. Painter, C.D. Hansen and M.F. Krogh. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.
- [17] N. Magnenat-Thalmann, I.S. Pandzic, J. Moussaly, D. Thalmann, Z. Huang, and J. Shen. The Making of Xian Terra-Cotta Soldiers. In: *Computer Graphics: Developments in Virtual Environments*, pp. 281–295, Academic Press, 1995.
- [18] K. Menzel. Parallel Rendering Techniques for Multiprocessor Systems. In *International Spring School on Computer Graphics*, Bratislava, 1994.
- [19] S. Molnar, M. Cox, D. Ellsworth and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.
- [20] N. Nupairoj and L. M. Ni. Benchmarking of Multicast Communication Services. Technical Report MSU-CPS-ACS-103, Dept. of Computer Science, Michigan State University, 1995.
- [21] L.L. Peterson and B.S. Davie. *Computer Networks*. Morgan Kaufmann, 1996.
- [22] M. Stumm. The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pp. 12–22, 1988.
- [23] S. Whitman. Dynamic Load Balancing for Parallel Polygon Rendering. *IEEE Computer Graphics and Applications*, 14(4):41–48, 1994.
- [24] N. Wilt. *Object-Oriented Ray Tracing in C++*. The Waite Group, 1994.
- [25] R. Yagel and R. Machiraju. Data-Parallel Volume Rendering Algorithms. *The Visual Computer*, 11:319–338, 1995.
- [26] R.R. Yager and L.A. Zadeh (Eds.), *Fuzzy Sets, Neural Networks, and Soft Computing*. Van Nostrand Reinhold, New York, 1994.