Eurographics Symposium on Parallel Graphics and Visualization (2004)
Dirk Bartz, Bruno Raffin and Han-Wei Shen (Editors)

Tuning of Algorithms for Independent Task Placement in the Context of Demand-Driven Parallel Ray Tracing†

T. Plachetka

Department of Computer Science, University of Bristol, United Kingdom

Abstract

This paper investigates assignment strategies (load balancing algorithms) for process farms which solve the problem of online placement of a constant number of independent tasks with given, but unknown, time complexities onto a homogeneous network of processors with a given latency. Results for the chunking and factoring assignment strategies are summarised for a probabilistic model which models tasks' time complexities as realisations of a random variable with known mean and variance. Then a deterministic model is presented which requires the knowledge of the minimal and maximal tasks' complexities. While the goal in the probabilistic model is the minimisation of the expected makespan, the goal in the deterministic model is the minimisation of the worst-case makespan. We give a novel analysis of chunking and factoring for the deterministic model. In the context of demand-driven parallel ray tracing, tasks' time complexities are unfortunately unknown until the actual computation finishes. Therefore we propose automatic self-tuning procedures which estimate the missing information in run-time. We experimentally demonstrate for an "everyday ray tracing setting" that chunking does not perform much worse than factoring on up to 128 processors, if the parameters of these strategies are properly tuned. This may seem surprising. However, the experimentally measured efficiencies agree with our theoretical predictions.

Categories and Subject Descriptors (according to ACM CCS): C.1.2 [Computer Systems Organization]: Processor Architectures / Multiple Data Stream Architectures (Multiprocessors) / Parallel Processors, D.4.1 [Software / Operating Systems / Process Management / Scheduling]: , G.1.0 [Numerical Analysis]: General / Parallel Algorithms, I.3.6 [Computer Graphics]: Three-Dimensional Graphics and Realism / Raytracing

1. Introduction

Ray tracing [Whi80] computes an image of a 3D scene by recursively tracing rays from the eye through the pixels of a virtual screen into the scene, summing the light path contributions to pixels' colours. In spite of various optimisation techniques [Gla89], typical sequential computation time range from minutes to hours.

Approaches to the parallelisation of ray tracing can be roughly divided into two classes. *Object space subdivision algorithms (data-driven)* geometrically divide the 3D scene into disjoint regions which are distributed in processors' memories. A process traces a ray until the ray leaves the process' region; after that the ray is passed to the appropriate neighbour. An obvious advantage of the object space subdivision algorithms is that the scene size is only limited by the total memory of processors; however, the implementation of these algorithms may be laborious. It is also unclear how the problem of load balancing should be solved. [DS84], [Pit93] *Screen space subdivision algorithms (demand-driven)* exploit the fact that the primary rays which are sent from the eye through the pixels of the virtual screen are independent of each other. Tracing of primary rays can therefore run in parallel without a communication between processes. An important advantage of the screen space subdivision algorithms is that they can easily be incorporated into an existing sequential code. The problem of large data placement and access on distributed memory machines is not studied in this

† This work was mostly done at the University of Paderborn and was partially supported by the projects VEGA 1/0131/03 (Comenius University, Bratislava) and RoD (University of Bristol).

paper; it can be handled independently using a distributed object database. [Gre91], [Pla02a] This paper focuses on the problem of load balancing for screen space subdivision algorithms.

Previous works on demand-driven ray tracing merely present an empirical experience with chunking assignment strategies [Gre91], [BBP94], [KH95], [FHK97], [FHK98] and factoring assignment strategies [FHK97], [FHK98], [Pla98]. However, they do not explain what the optimal parameter settings for these strategies look like or how they can be determined. It seems paradoxical that a chunk size of 9 pixels is suggested in [BBP94] whereas chunk sizes of 4096 and 16384 pixels are investigated in [FHK97]. *We will show in this paper that the optimal chunk size is a function of the screen resolution, number of* `WORKER` *processes, latency of chunk assignment and pixels' (or chunks') time complexities. The knowledge of these parameters not only determines the optimal chunk size but also allows for a prediction of efficiency for a given number of* `WORKER`*s. The information on an empirical chunk size alone is insufficient.*

A diffusive load balancing strategy is advocated in [HA98] and arguments are given as to why naive static and probabilistic assignment strategies do not achieve an acceptable performance on a large number of processors. However, it remains unclear how the optimal setting of the parameters of the proposed diffusive algorithm can be determined.

This paper is organised as follows: Section 2 defines the problem of independent task assignment. Section 3 summarises known results for the probabilistic model. Section 4 presents a novel analysis for the deterministic model. Section 5 proposes tuning strategies for the deterministic model. Section 6 shows experimental results for demand-driven ray tracing and compares the measured data with theoretical predictions from Section 4. Section 7 concludes the paper.

2. Problem definition

We assume that the computational time complexities of screen pixels are not known until the computations have actually been performed; however, we assume that this unknown complexity is constant. This means that if the same pixel (*task*) is computed on any two processes then the two computational times will be exactly the same. We assume that the number of processes does not change e.g. due to faults in processors or links between them. We also assume that the computations on different pixels are independent of each other. (Some anti-aliasing techniques introduce dependencies between pixels in order to save the number of additional primary rays, but these techniques can be applied in post-processing [Pla02a]). Finally, we assume that a pixel must be processed sequentially in a whole. *The goal of load balancing is to assign subsets of screen pixels to the processes so that the time of the parallel computation of all the screen pixels (makespan) is minimal.*

We will refer to the assigned sets of pixels as *jobs*. Each assignment of a set of pixels to a process incurs a time penalty perceived by that process. This penalty, referred to as *latency*, is caused by the actual mechanism which is used for the assignment (e.g. message passing).

In terms of message passing, the model above corresponds to a process farm which consists of one `LOADBALANCER` process and N `WORKER` processes. Each `WORKER` process runs a loop in which it sends a *job request* to the `LOADBALANCER`, waits for a job and then processes the received job. The `LOADBALANCER` process runs a loop in which it waits for a job request from any `WORKER` and then assigns a job to the `WORKER` which sent the request. The `LOADBALANCER`'s loop runs until all pixels have been processed. After that the `LOADBALANCER` collects all outstanding job requests, replies them with `NO_MORE_WORK` in order to make the `WORKER`s terminate and then terminates itself.

The question is how large the jobs which are assigned by the `LOADBALANCER` in replies to job requests must be in order to process all pixels as quickly as possible. A procedure which answers this question is called a *(assignment) strategy*. Note that a strategy must work *online* because when a job request arrives, the `LOADBALANCER` must immediately decide how many pixels should be assigned in that job.

The job results must usually be assembled into a final result. In the context of ray tracing, the final compositing of the resulting image parts into a single image can involve a relatively large volume of data. This compositing burden can be moved to an additional `MASTER` process (Fig. 1). [Pla98] The addition of the `MASTER` process also amortises the compositing overhead because some of the job results are processed during the parallel computation of other jobs.

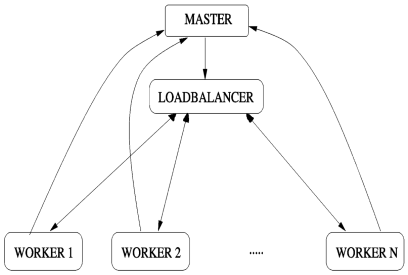

Figure 1: *Process farm*

The problem of load balancing for screen-space subdivision algorithms for parallel ray tracing is identical to the problem of scheduling parallel loops. The latter problem arises in the optimisation of loops in compilers for multiprocessor machines. A more general term which is also used to denote the same problem is "allocation of independent tasks to parallel processors". (The terms "task" and "batch" translate into terms "pixel" and "job" in the context of parallel ray tracing.)

A certain knowledge concerning the tasks' time complexities (computational times on pixels) is required in order to make claims about performance of a strategy. Two models have been proposed which characterise this knowledge. Section 3 briefly summarises the known results for chunking and factoring strategies for the probabilistic model (we recommend [Hag97] for a more extensive overview of the results). Section 4 presents a novel analysis of chunking and factoring for the deterministic model.

It is commonly assumed in both the probabilistic and the deterministic models that the latency L is a (known) constant and that the complexity of the strategy itself does not increase the latency. While this assumption simplifies the analysis of assignment strategies, the latency L is not strictly constant in practice. The network latency perceived by a process may vary e.g. when several messages arrive in the process at the same time. Moreover, L not only includes the network latency but also the time of preprocessing and postprocessing which is required for each job in the parallel implementation, whereas it only runs once in the sequential implementation.

2.1. Notation

The following notation is used throughout this paper:

N number of `WORKER` processes
W number of tasks (pixels)
L latency (the overhead of the assignment of one job)

3. Assignment strategies for the probabilistic model

The probabilistic model has been traditionally investigated in the context of optimisation of parallel loops. It assumes that the tasks' complexities are identical random variables with (known) mean μ and (known) standard deviation σ.

3.1. Fixed-size chunking

The fixed-size chunking strategy always assigns jobs of the same size to idling `WORKER` processes (the last job may be an exception). This model was investigated by Kruskal and Weiss. The following estimation of the expected makespan $E[\mathcal{M}]$ for the chunk size K is given in [KW85]:

$$E[\mathcal{M}] \approx \frac{W}{N}\mu + \frac{WL}{NK} + \sigma\sqrt{2K\ln N} \tag{1}$$

This formula has a nice intuitive interpretation. The first term describes time of executing W tasks on N processors on a system with no overhead. The second term describes the latency overhead. The third term describes the load imbalance due to the variation in tasks' durations. Unfortunately, the estimation in Equation 1 only holds if W and K are large and $K \gg \log N$. If these assumptions hold then also the optimal chunk size K^{opt} can be estimated:

$$\hat{K}^{opt} = \left(\frac{\sqrt{2}WL}{\sigma N\sqrt{\ln N}}\right) \tag{2}$$

If the assumptions above do not hold, [KW85] gives the following estimates for the expected makespan $E[\mathcal{M}]$:

$$E[\mathcal{M}] \approx \frac{W}{N}\mu + \frac{WL}{NK} + \sigma\sqrt{2K\ln\frac{\sigma N}{\sqrt{K}\mu}} \tag{3}$$

for $K \ll W/N$ and small $\sqrt{K}/N$; and

$$E[\mathcal{M}] \approx \frac{W}{N}\mu + \frac{WL}{NK} + \frac{N\sigma^2}{\mu} \tag{4}$$

for $K \ll W/N$ and large $\sqrt{K}/N$. However, a tractable analytical expression for the optimal chunk size K cannot be derived from Equations 3 and 4.

3.2. Factoring

The factoring strategy works in rounds. In each round, it assigns N jobs of equal size. The job size is decreased by a factor after each round. The idea behind this strategy is to "smooth" the high imbalance of the early rounds with smaller jobs of the later rounds. The `LOADBALANCER` process must at any time keep enough unassigned tasks for this smoothing—on the other hand, the assigned jobs should always be as large as possible in order to minimise the latency overhead.

An approximation of the optimum job size $\hat{K}_i^{opt}$ which is used in round i was determined by Flynn and Flynn Hummel [FFH90] by estimating the maximal portion of the remaining (unassigned) work which has a high probability of being completed by N processors before the optimal time, i.e. $\mu W/N$. The analysis yields the following iteration scheme (at the beginning of round i, r_i denotes the number of still unassigned tasks, $\frac{1}{Nx_i}$ is the division factor): [FHSF91]

$$r_1 = W \tag{5}$$
$$r_{i+1} = r_i - N\hat{K}_i^{opt} \tag{6}$$
$$x_1 = 1 + \frac{N^2}{r_1}\left(\frac{\sigma}{\mu}\right)^2 \tag{7}$$
$$x_{i+1} = 2 + \frac{N^2}{r_i}\left(\frac{\sigma}{\mu}\right)^2 \tag{8}$$
$$\hat{K}_i^{opt} = \frac{r_i}{Nx_i} \tag{9}$$

The latency L is ignored in the derivation of the scheme above. The latency is only used as a termination criterion—the factoring rounds stop when $\mu\hat{K}_i^{opt} \leq L$. After that the

remaining tasks are assigned in equal-sized jobs of size $\hat{K}_i^{opt}$ which was used in the last iteration i.

Note that this iteration scheme only requires the knowledge of the coefficient of variation cov of the tasks' probability distribution ($cov = \sigma/\mu$). There are two extreme cases:

1. If $cov = 0$ (no variance) then this strategy is equivalent to a static job assignment strategy which assigns N equal-sized jobs to N processors in a single round.
2. If $cov \to \infty$ (unbounded variance or negligible tasks' time complexities) then this scheme is equivalent to the dynamic job assignment which assigns the W tasks one-by-one to idle processors.

Experiments with the factoring strategy are presented e.g. in [FHSF91] and [BFH95]. However, the authors did not attempt to estimate cov. Instead of this the values of x_i in Equations 7 and 8 were assigned an empirical value of 2. This means that $\hat{K}_i^{opt}$ was halved after each iteration i. The reasoning on why halving leads to good experimental results is in our opinion not persuasive—although it performed well for the chosen problem instances and the machine used for the experiments, it would not perform well for other instances and other machines.

4. Assignment strategies for the deterministic model

Plachetka [Pla98], [Pla02a] introduced a deterministic model which does not model tasks as random variables. This model only assumes (in addition to the assumptions from Section 2) the knowledge of the maximal and the minimal tasks' complexities T_{min} and T_{max}. Whereas the goal in the probabilistic model is to minimise the expected makespan $E[\mathcal{M}]$, the goal in the deterministic model is to minimise the absolute makespan $\mathcal{M}$ for the worst possible arrangement of tasks' complexities.

4.1. Fixed-size chunking

The fixed-size chunking strategy always assigns jobs of the size K to the idling `WORKER` processes whereby K remains constant during the algorithm (the last assignment may be an exception). It can be observed that the worst makespan is obtained if one of the processes is always assigned jobs which contain the most complex tasks. As we are interested in the worst-case makespan, we can safely assume that the time complexity of all tasks is equal to this heaviest task's complexity T_{max}. Then the chunking computation depicted as a time diagram (Fig. 2) consists of $\lfloor \frac{W}{NK} \rfloor$ blocks of the width K and of the height N (both the width and the height are expressed in the number of tasks). Moreover, if $\frac{W}{NK}$ is not an integer number then there is one extra block of the width at most K, and of the depth at most $N-1$. We will assume that there indeed is such an extra block and that its width is K (the depth of this extra block is not important as it does not influence the makespan).

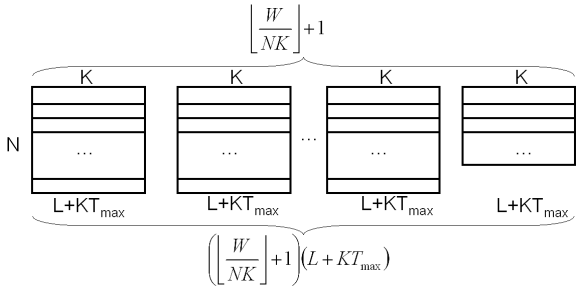

Figure 2: *The structure of the worst case for the chunking strategy (time diagram)*

The total time complexity of each block (including the extra block) is $L + KT_{max}$. Hence, the makespan $\mathcal{M}$ of this worst-case scenario can be bounded from above by $\mathcal{M}_{high}$ (which ignores the rounding of $\lfloor \frac{W}{NK} \rfloor$) as follows:

$$\mathcal{M} < \mathcal{M}_{high} = \left(1 + \frac{W}{NK}\right)(L + KT_{max}) \tag{10}$$

The optimal chunk size K which minimises $\mathcal{M}_{high}$ can be found by setting the first derivative of $\mathcal{M}_{high}$ with respect to K to zero:

$$\mathcal{M}'_{high} = T_{max} - \frac{WL}{NK^2} := 0 \tag{11}$$

Solving for K yields

$$K^{opt} = \sqrt{\frac{WL}{NT_{max}}} \tag{12}$$

The substitution of K in Equation 10 with K^{opt} from Equation 12 yields an upper bound on the optimal makespan in the worst case:

$$\mathcal{M}_{high}^{opt} = \frac{WT_{max}}{N} + L + 2\sqrt{\frac{WT_{max}L}{N}} \tag{13}$$

4.2. Factoring

The factoring strategy works in rounds. During each round, N jobs of equal size are assigned to idle $WORKER$ processes (it can happen that some of the $WORKER$ processes can be assigned more than one job within the same round). When a round finishes, the job size is decreased. During the last factoring round, single-task jobs are assigned. Due to the integer arithmetic, one extra round may be needed in order to assign the remaining tasks (the number of these remaining tasks is smaller than N).

The ratio $T = T_{max}/T_{min}$ is used to determine the job size K_i for the round i. This job size is a factor of the work remaining (the work is expressed in the number of tasks). Let W_i denote the number of still unassigned tasks at the beginning of the round i. The factoring strategy always guarantees that the computation of the job of the size K_i will not take longer than the parallel computation of the still unassigned $W_i - K_i$ tasks on the remaining $N-1$ WORKERs:

$$time(K_i) \leq \frac{time(W_i - K_i)}{N-1} \tag{14}$$

where $time(K_i)$ is the (sequential) processing time on K_i tasks. Equation 14 can be equivalently written as

$$TK_i \leq \frac{W_i - K_i}{N-1} \tag{15}$$

which yields

$$K_i \leq \frac{W_i}{1+T(N-1)} \tag{16}$$

The maximal K_i apparently minimises the number of factoring rounds (and therefore the makespan). Hence,

$$K_i^{opt} = \left\lfloor \frac{W_i}{1+T(N-1)} \right\rfloor \tag{17}$$

The maximal makespan is obtained in the case if one of the WORKERs always gets jobs which contain the most complex tasks. As we are interested in the worst case, we can safely assume that the time complexity of all tasks is equal to this maximal complexity T_{max}. We will also assume that there is one extra round (and that at least one of the tasks which are assigned in the extra round has the time complexity T_{max}). This worst case is depicted in Fig. 3. The computation consists of r factoring rounds and one extra round. The contribution of i-th factoring round to the makespan is $L + K_i T_{max}$. The contribution of the extra round is $L + T_{max}$.

The round r is the last round at the beginning of which the number of still unassigned tasks W_r is at most N (as the size of the N jobs assigned in the round r is $K_r = 1$). It can be observed that the number of still unassigned tasks W_i at the beginning of round i is equal to

$$W_i = W\left(1 - \frac{N}{1+T(N-1)}\right)^{i-1} \tag{18}$$

Solving $W_r \leq N$ for maximal r using Equation 18 yields

$$r = 1 + \left\lfloor \log_{1-N/(1+T(N-1))}(N/W) \right\rfloor \tag{19}$$

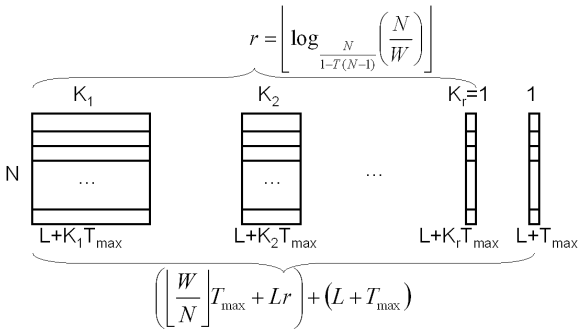

Figure 3: *The structure of the worst case for the factoring strategy (time diagram)*

Hence, the makespan $\mathcal{M}^{opt}$ in the worst case is equal to

$$\mathcal{M}^{opt} = T_{max}(\lfloor W/N \rfloor + 1) + L(r+1) \tag{20}$$

5. Tuning of the deterministic factoring algorithm

A common practical problem of the assignment strategies presented in Section 3 and Section 4 is that their parameters are not known before the parallel computation finishes. An empirical constant setting of the parameters may be close to the optimal setting for a certain input and a certain machine, but it may be far from the optimal setting for other inputs or other machines. We tackle this problem for the deterministic job assignment strategies using a self-tuning approach.

The parameters which are known or can be measured beforehand for a given machine and for one run are W, L and N. The only unknown parameter which is required by the factoring strategy is T. (In order to predict the makespan, the a priori knowledge of T_{max} is also required—which implies the a priori knowledge of T_{min}.)

The parameter T controls the trade-off between the fixed-size chunking with maximal chunks ($T \to 1$ yields the chunk size of $\lfloor W/N \rfloor$ tasks) and fixed-size chunking with minimal chunks ($T \to \infty$ yields the chunk size of 1 task). Note that if the latency exceeds the computational times of tasks then ending up with single-task jobs is not desirable—the factoring rounds should be terminated sooner and the yet unassigned tasks should be distributed in N equal-size chunks. In order to achieve this, we introduce a parameter A which determines the minimal (atomic) job size which is assigned during the factoring rounds. The resulting factoring algorithm is shown in Fig. 4.

5.1. Tuning of the atomic job size A

The parameter A determines the minimal job size. An intuitive approximation of the optimal setting of this parameter is $\hat{A}^{opt} = \max(L/T_{max}, 1)$ (a more precise setting of A^{opt}

```
LOADBALANCER(float T, int A, int W, int N)
  int job_size;
  int work = W;
  while (work > 0)
    job_size = max (A, ⌊work/(1 + T · (N − 1))⌋);
    for (counter = 0; counter < N; counter++)
      wait for a job request from an idle WORKER;
      if (work > 0)
        send job of size job_size to the WORKER;
        work = work − job_size;
  reply job requests with NO_MORE_WORK;
```

Figure 4: *The factoring algorithm used in the* `LOADBALANCER` *process*

which minimises the worst-case makespan can be probably derived). Still, the optimal setting of A requires an a priori knowledge of T_{max}.

If $A < A^{opt}$ is used in the algorithm in Fig. 4 then communication costs will dominate the computational times of some jobs. If $A > A^{opt}$ then an unnecessary imbalance will be observed.

We suggest to run the first factoring round with $A = 1$ and adapt A according to the measurements performed in the run-time. Before the `LOADBALANCER` process assigns a job J_i, it starts a stopwatch. This stopwatch stops when `LOADBALANCER` receives another job request from the same `WORKER`. Let $t_{total}(J_i)$ denote the time measured by `LOADBALANCER`. The `WORKER` process measures the processing time of the job $t_{worker}(J_i)$ and reports this time to `LOADBALANCER` (together with the job request). The difference $t_{total}(J_i) - t_{worker}(J_i)$ is used to estimate the latency L. If

$$\frac{t_{total}(J_i) - t_{worker}(J_i)}{t_{worker}(J_i)} \geq 1 \qquad (21)$$

has been measured e.g. for all jobs J_i of the same round then this is a good indication of that the job size should not be further decreased. The factoring rounds should be then replaced with the final chunking.

Remark. Note that the chunking strategy is a special case of the extended factoring strategy. If $T \to \infty$ then the algorithm in Fig. 4 assigns jobs of the constant size A. The tuning procedure above can therefore be used in order to determine the chunk size for the chunking strategy. •

5.2. Tuning of the factor T

The factor $T^{opt} = T_{max}/T_{min}$ is generally unknown before the actual computation has finished. However, T can also be tuned in run-time. This tuning is independent on the tuning of the atomic size A. We propose two tuning approaches:

- **Conservative approach.** T is always set to the maximal ratio over all jobs which have already been processed. This statistics can be measured in the `WORKER` processes and reported to the `LOADABALANCER` process when a job request is being sent.
  Note that an underestimation of T can still occur. However, `LOADBALANCER` can set a deadline for each job which it assigns and it can detect if a job has not been processed in time. If this happens, the `LOADBALANCER` sends an `ABORT` message to the `WORKER`, making the `WORKER` finish its job with only a partial result. The tasks of the aborted job which have not been processed are returned back to the task-pool in the `LOADBALANCER`. After that `LOADBALANCER` adjusts (increases) its estimation of T (according to the number of unprocessed tasks in the aborted job) and continues.
- **Optimistic approach.** T is set to an empirical constant and remains constant. Both overestimation and underestimation of T^{opt} lead to an unnecessary increase of the makespan. If $T > T^{opt}$ then the number of rounds will be greater than the optimal number of rounds—however, this performance loss can be predicted (and it can be usually neglected, especially if the overestimation is not large). The underestimation $T < T^{opt}$ leads to an increase of imbalance. In this case a job request will be replied by the `LOADBALANCER` with `NO_MORE_WORK`, while other `WORKER`s may have several yet unprocessed tasks. When this happens, the `WORKER` which received the `NO_MORE_WORK` message immediately begins with *work stealing*, trying to find some other `WORKER` with enough yet unprocessed tasks. A part of these yet unprocessed tasks will be then passed to the idling `WORKER`.

There is a good reason for the favouring of the optimistic approach: although the empirical T can be much smaller than the actual T^{opt} for individual *tasks*, the factor between the computational times of any two assigned *jobs* may be still close to the empirical T. Using the empirical T instead of the theoretically correct T^{opt} can significantly decrease the number of factoring rounds and therefore the latency overhead. The additional work-stealing phase helps to decrease the imbalance in the unlucky case when too many tasks with high complexity were assigned in one job.

6. Experiments with demand-driven parallel ray tracing

We used the Persistence of Vision Ray Tracer (POV-Ray) version 3.1g as a base for our demand-driven parallelisations. POV-Ray is a state of the art (sequential) ray tracer which implements all important optimisation techniques. For all the following experiments, we used a fairly complex "everyday scene" with ca. 600 objects and 8 point light

sources. We rendered an image of this scene in PAL resolution (720x576 pixels) with no anti-aliasing (Fig. 5).

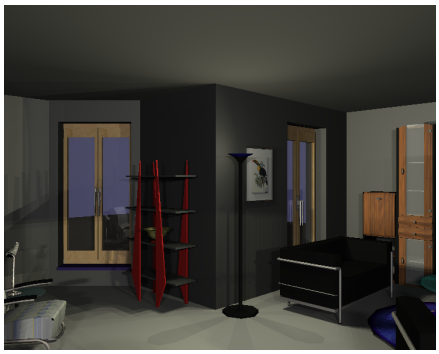

Figure 5: *The rendered image of the "everyday scene"*

All the experiments were running on a partition of Siemens-Fujitsu *hpcLine* cluster in the Paderborn Center for Parallel Computing (PC^2) at the University of Paderborn. Each process was mapped onto one processor of the allocated partition. The cluster consists of 96 Siemens Primergy nodes with two 850 MHz Intel Pentium III and 512 MBytes RAM per node, running Linux Redhat. The nodes are connected via two independent networks: SCI (500 MBit/second Scalable Coherent Interface by Scali/Dolphin) and Fast Ethernet (100 MBit/second). We used our own message passing library TPL, which uses TCP and the Fast Ethernet network. [Pla02b] The highest network latency (roundtrip time) perceived by a process, measured on a `PINGPONG` benchmark which uses TPL was ca. 0.6 millisecond.

When we refer to sequential time, we mean the running time of the original sequential POV-Ray—not the parallel time with 1 `WORKER` process. Efficiencies reported in this paper relate to the sequential time.

6.1. Pessimistic (worst-case) predictions using optimistic (average) estimations

The formulas from Section 4 allow us to predict the results of the efficiency experiments for the worst case if L, T_{max} and T are known. Unfortunately, at the time of these experiments these statistics for individual tasks (pixels) were not collected during the runs. However, we were able to roughly reconstruct the parameters as follows:

- $L = 0.007$ second. This time includes the pre- and post-processing such as packing/unpacking of the frame buffer, repetitive computation of vista buffer etc. We got this estimation from the difference between the sequential time and the parallel chunking time with 1 `WORKER` process.
- $T_{max} = 0.0022591$ second. This is actually the average computational time on one pixel estimated from the sequential time (ca. 937 seconds), not the maximal computational time on one pixel.
- $T = 3$. This is roughly the factor between the computational times of blocks of pixels of size at least 360 pixels (see Section 6.2), not the factor of maximal and minimal computational times on individual pixels (which is much higher). This factor came out from our empirical measurements, see Fig. 8 and it is relatively low thanks to the quasi-random mixture of high-complexity and low-complexity tasks in the jobs.

Equation 13 and Equation 20 can be used to obtain upper bounds on the *expected makespan* (i.e. lower bounds on the expected efficiency). In such a case T_{max} should be interpreted as the maximal *job's* time complexity, where a job's time complexity is defined as an average of time complexities of the tasks included in that job; and T should be interpreted as the maximal ratio of *jobs'* time complexities. The meaning of the reconstructed parameters above is not much different from this interpretation.

6.2. Chunking

In our first experiment, we manually tuned the optimal chunk size for the chunking assignment strategy. We ran the parallel program with 90 `WORKER`s and measured the absolute parallel times as a function of the chunk size K. The values of K were 22, 45, 90, 180, 360 and 720 (we had already known from our previous experience that the empirical optimal chunk size for this setting is smaller than 720). The results of the measurements are shown in Fig. 6. The empirical optimal chunk size $\hat{K}^{opt}$ is in the interval 45–360 pixels (the differences between these parallel times are negligible and subject to external factors which are beyond our control). We set $\hat{K}^{opt} = 360$. (Expressed informally, this is a chunk size which we can safely "afford" in order to not let the latency dominate the computational times of the chunks. Note that the non-constant latency will be lower for fewer `WORKER`s than 90.) The predicted K^{opt} for the worst-case scenario (Equation 12) is ca. 800 for 2 `WORKER`s, ca. 280 for 16 `WORKER`s and ca. 100 for 128 `WORKER`s. Our choice $\hat{K}^{opt} = 360$ is a reasonable constant approximation of K^{opt} for the number of `WORKER`s in the given range.

We then measured the efficiency of the chunking strategy with constant $\hat{K}^{opt} = 360$ for a varying number of `WORKER`s (powers of 2). The results of these measurements and the predicted efficiency (using Equation 10) are shown in Fig. 7.

6.3. Factoring

The goal of the experiments with factoring was to empirically find the factor T which yields the maximal efficiency for the given setting. This empirical optimal $\hat{T}^{opt}$ is the ratio of the maximal and minimal times of the *jobs* assigned during the run, not the ratio of the maximal and minimal times of single *tasks* (pixels). The actual T^{opt} may be much higher than the empirical $\hat{T}^{opt}$. We also used the previously found

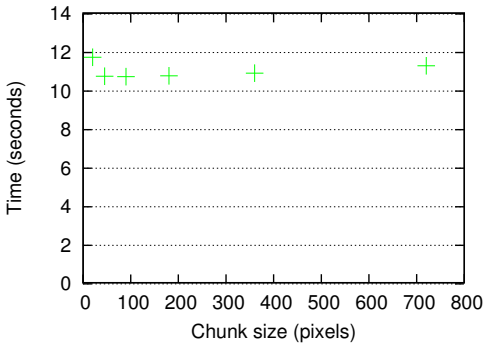

Figure 6: *Absolute parallel times of the chunking strategy with 90 WORKERs, for a varying chunk size*

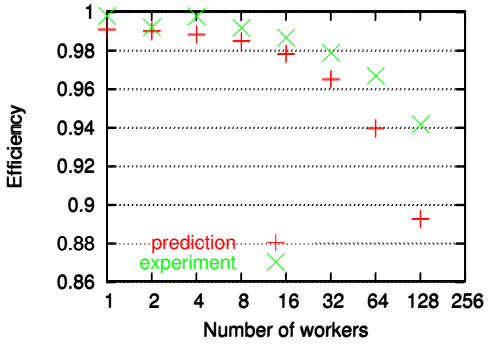

Figure 7: *Efficiency of the chunking strategy with $\hat{K}^{opt} = 360$, for a varying number of WORKERs (note the logarithmic scale on x axis)*

chunk size as the atomic job size for the factoring algorithm from Fig. 4 ($A = 360$).

Fig. 8 shows the results of the measurements for $T = 2$, $T = 3$ and $T = \infty$ (a sufficiently large T) for a varying number of WORKERs (powers of 2). This graph also includes the predicted efficiency for $T = 3$ (using Equation 20). The efficiencies measured for $T = 1$ (static chunking) are not included in the graph; they dropped down quickly from 1.0 to 0.6 for 1–8 WORKERs and ranged between 0.6–0.4 for more than 16 WORKERs. Note that the empirical $\hat{T}^{opt}$ is between 3 and ∞ (whereby $\hat{T}^{opt} = \infty$ yields the chunking strategy with the chunk size $A = 360$).

7. Conclusions

We presented a worst-case analysis for chunking and factoring assignment strategies for the independent task placement problem, using a deterministic model. This analysis gives the best setting of the parameters for the worst-case arrangement, if a certain a priori information on the tasks' time complexities is known. In the context of demand-driven

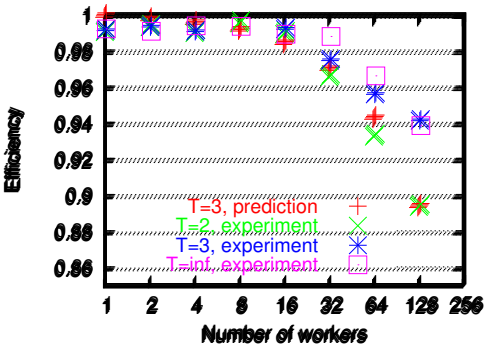

Figure 8: *Efficiency of the factoring strategy with $A = 360$, for a varying number of WORKERs (note the logarithmic scale on x axis)*

ray tracing, this information is unknown. Therefore we proposed self-tuning strategies which estimate the missing information in run-time. Our experiments show that even the very simple chunking assignment strategy yields a very good performance (more than 94%) for an "everyday ray tracing setting" on up to 128 WORKERs—provided that the chunk size is properly set. The factoring strategy never significantly outperformed the chunking strategy in the experiments; also our prediction models only slightly favour factoring over chunking (the predicted difference in their efficiencies is less than 2% for 1–128 WORKERs). If we keep the parameters intact in our prediction models, and only increase the number of WORKERs to 1024 then the predicted efficiency for the chunking strategy will be 85%. This seems to contradict the previously reported bad experience with chunking (e.g. in [FHK97]), but in fact it does not. None of the previous publications we know of tuned the chunk size—they used an empirical chunk size which could be far from the optimal one. We also disagree with Heirich and Arvo [HA98] who claim that chunking is insufficient for more than 128 WORKERs and seek a solution among diffusive strategies.

We stress that the chunk size $\hat{K}^{opt}$ which we empirically found in Section 6.2 is only valid for the given machine and communication library, given input, given screen resolution, given number of WORKER processes etc. A different setting would yield a different optimal chunk size K^{opt} which minimises the worst-case makespan. This is the reason why the chunk size must be tuned. According to Equation 12, the optimal chunk size K^{opt} is a function of W (number of tasks), N (number of WORKERs), L (latency) and T_{max} (maximal task's or job's time complexity). All these parameters should be reported in experiments with assignment strategies.

Note that our model does not depend on the mechanism which is used for the job assignment. Even though we assumed message passing throughout this paper, our prediction model and tuning strategies are also valid for shared memory architectures. [PMS*99]

A prefetching technique (combined with chunking) is proposed e.g. in [WBDS03]. This technique assigns more than one job to an idle `WORKER` at the beginning and immediately assigns another job to the `WORKER` when the first job has already been computed. Prefetching leads to a reduction of the latency (the `WORKER` does not wait for a job when it finishes the computation of the previous job) but increases the work imbalance. An analysis similar to ours is needed in order to justify the prefetching technique and to determine the optimal chunk size and the optimal number of prefetched jobs.

References

[BBP94] BADOUEL D., BOUATOUCH K., PRIOL T.: Distributing data and control for ray tracing in parallel. *IEEE Computer Graphics and Applications 14*, 4 (1994), 69–77. 2

[BFH95] BANICESCU I., FLYNN-HUMMEL S.: *Balancing Processor Loads and Exploiting Data Locality in Irregular Computations*. Tech. Rep. RC 19934, IBM Research, 1995. 4

[DS84] DIPPÉ M., SWENSSEN J.: An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics 18*, 3 (1984). 1

[FFH90] FLYNN L. E., FLYNN-HUMMEL S.: *Scheduling Variable-Length Parallel Subtasks*. Tech. Rep. RC 15492, IBM Research, 1990. 3

[FHK97] FREISLEBEN B., HARTMANN D., KIELMANN T.: Parallel raytracing: A case study on partitioning and scheduling on workstation clusters. In *Proc. of Hawaii International Conference on System Sciences (HICSS-30)* (1997), vol. 1, IEEE Computer Society Press, pp. 596–605. 2, 8

[FHK98] FREISLEBEN B., HARTMANN D., KIELMANN T.: Parallel incremental raytracing of animations on a network of workstations. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)* (1998), Arabnia H., (Ed.), vol. 3, CSREA Press, pp. 1305–1312. 2

[FHSF91] FLYNN-HUMMEL S., SCHONBERG E., FLYNN L. E.: Factoring: A practical and robust method for scheduling parallel loops. In *Proc. of Supercomputing '91* (1991), IEEE Computer Society / ACM, pp. 610–619. 3, 4

[Gla89] GLASSNER A. S.: *An Introduction to Ray Tracing*. Academic Press, 1989. 1

[Gre91] GREEN S.: *Parallel Processing for Computer Graphics*. Research Monographs in Parallel and Distributed Computing. Pitman Publishing, 1991. 2

[HA98] HEIRICH A., ARVO J.: A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing 12*, 1–2 (1998), 57–68. 2, 8

[Hag97] HAGERUP T.: Allocating independent tasks to parallel processors: An experimental study. *Journal of Parallel and Distributed Computing 47*, 2 (1997), 185–197. 3

[KH95] KEATES M. J., HUBBOLD R. J.: Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum 14*, 4 (1995), 189–202. 2

[KW85] KRUSKAL C. P., WEISS A.: Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering 11*, 10 (1985), 1001–1016. 3

[Pit93] PITOT P.: The Voxar project. *IEEE Computer Graphics and Applications* (1993), 27–33. 1

[Pla98] PLACHETKA T.: POV||Ray: Persistence of Vision parallel raytracer. In *Proc. of Spring Conference on Computer Graphics* (1998), Szirmay-Kalos L., (Ed.), Comenius University, Bratislava, pp. 123–129. 2, 4

[Pla02a] PLACHETKA T.: Perfect load balancing for demand-driven parallel ray tracing. In *Proc. of Euro-Par 2002* (2002), Monien B., Feldman R., (Eds.), vol. 2400 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 410–419. 2, 4

[Pla02b] PLACHETKA T.: (Quasi-) thread-safe PVM and (quasi-) thread-safe MPI without active polling. In *Proc. of the 9th EuroPVM/MPI User's Group Conference* (2002), Kranzlmüller D., Kacsuk P., Dongarra J., Volkert J., (Eds.), vol. 2474 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 296–305. 7

[PMS*99] PARKER S., MARTIN W., SLOAN P. P. J., SHIRLEY P., SMITS B., HANSEN C.: Interactive ray tracing. In *Proc. of the 1999 Symposium on Interactive 3D Graphics* (1999), pp. 119–126. 8

[WBDS03] WALD I., BENTHIN C., DIETRICH A., SLUSALLEK P.: Interactive ray tracing on commodity PC clusters—state of the art and practical applications. In *Proc. of Euro-Par 2003* (2003), Kosch H., Böszörmenyi L., Hellwagner H., (Eds.), vol. 2790 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 499–508. 9

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (1980), 343–349. 1