Load Balancing in Mesh-like Computations using Prediction Binary Trees *

Biagio Cosenza Gennaro Cordasco Rosario De Chiara
Vittorio Scarano
ISISLab, Dipartimento di Informatica ed Applicazioni "R.M. Capocelli"
Università degli Studi di Salerno, Salerno (Italy)
{cosenza, cordasco, dechiara, vitsca}@dia.unisa.it

Ugo Erra
Dipartimento di Matematica e Informatica
Università degli Studi della Basilicata, Potenza (Italy)
ugo.erra@unibas.it

Abstract

We present a load-balancing technique that exploits the temporal coherence, among successive computation phases, in mesh-like computations to be mapped on a cluster of processors. Our method partitions the computation in balanced tasks and distributes them to independent processors 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 phase) cost estimate. The PBT is designed so that it balances the load across the tasks as well as reduce 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).*

Our strategy has been assessed on a significant problem, Parallel Ray Tracing. Our implementation shows a good scalability, and improves over coherence-oblivious implementations. We report different measurements showing that granularity of tasks is a key point for the performances of our decomposition/mapping strategy.

1 Introduction

Parallel Computing

The evolution of computer science in the last 2 decades has been characterized by the architectural shift that has brought centralized computation paradigm toward distributed architectures where data processing and data storing are cooperatively performed on several nodes, interconnected by a network.

The problem of scheduling a parallel program to a set of homogeneous processor for minimizing the completion time (that is the time when the last processor complete its job) of the program has been extensively studied (see [11] for a comprehensive presentation). Indeed, 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 [12].

The number and size of tasks which a given computation is decomposed into 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, interaction between tasks, and/or other important factor, limits our choice to coarse-grained granularity. 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* (see [16]) but it should be emphasized that dependency can also refer to other issues such as locality of access to memory (effective usage of

*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.

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 [7]).

Mesh-like computations. In this paper, we focus our study on mesh-like computations, where a set of t independent tasks are represented as items in a $\sqrt{t} \times \sqrt{t}$ 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 interprocessors 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 a 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*).

Our result

In this paper 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. 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 strategies, 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 an efficient mapping.

We validate our strategy by using interactive rendering with Parallel Ray Tracing [22] 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 scalability and good performances improvements, with different granularity (size of tiles), with respect to the static assignment of tiles (tasks) to processors.

It should be said that, besides other graphical applications (e.g. image dithering), further examples of mesh-like computation where our techniques can be fruitfully used range from simple cases, such as matrix multiplication, to more complex computations, such as *Distributed Adaptive Grid Hierarchies* [17].

Previous Works

In [15], 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 [3]. 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 powerful [3] and stable [1], 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 this environments, performance processors are both irregular and time-varying because of uneven underlying load on the various resources. In [23] 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.

Organization of the paper

In the next section we present the strategy and introduce the Prediction Binary Tree, proving how effective is its updating at each phase.

Then, in Section 3, we instantiate our techniques for a parallel ray tracing application: we, first, describe the problem, the implementation and, then, show how experiments validate our statements of effectiveness and scalability for the strategy.

Finally, in Section 4, we conclude the paper with comments and further directions of research.

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 data (*Principal Data Items* (PDIs)). Auxiliary global information (*Additional Data Items* (ADIs)) are 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 tiles and sends back the partial output. If other tiles are not yet computed, the master sends another task to the worker that just finished its own. 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 considerations would ask for smaller m. In fact, an algorithm that has large m requires more *communication costs* than an algorithm with smaller m, both in latency (more messages) and bandwidth (communication overhead for each message). Other consideration that would require small m are *locality of interaction* and *spatial coherence*.

Our strategy takes into account all the considerations above, by addressing the uneven spread of the load with the Prediction Binary Tree that, with a negligible overhead, keeps the load balanced without resorting to increase the number of tiles. Thereby, our solution does not increase significantly the data-movement overhead, reduces tasks interactions, and uses effectively the local cache of workers. Therefore, we are able to address simultaneously and positively all the issues above, by providing a technique that uses a moderate amount of tiles.

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) whose 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.

We, now, 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 the horizontal (resp. vertical) axes. This assure 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 1 gives an example of a PBT, with the corresponding mesh partition on the left.

Updating PBT. 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

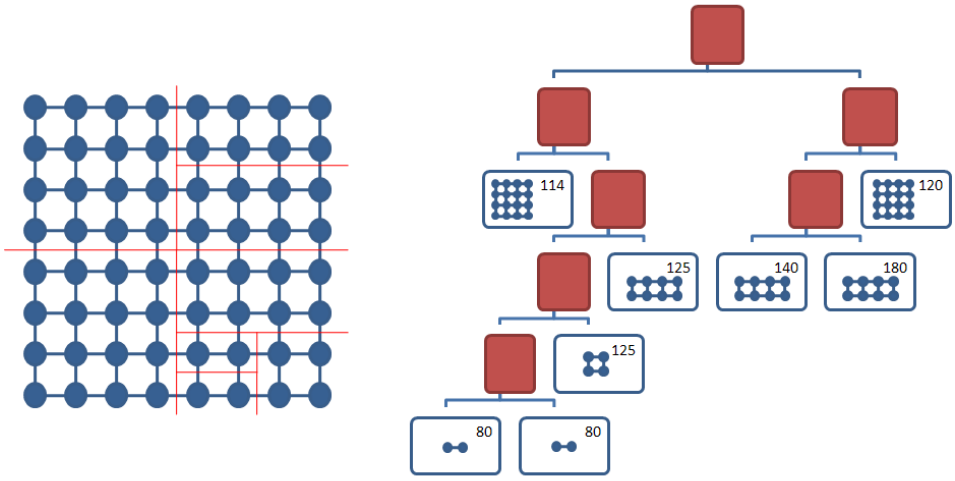

Figure 1: An example of a PBT tree: the mesh on the left has been computed with the computation times (in ms) for each tile shown on the leaves.

updated for the next phase. Here we describe a provably 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.

We, first, define the variance as a metric to measure the (estimated) computational unbalance that is expected given the tiling provided by the PBT 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 splitting a leaf ℓ_a into two leaves ℓ_{a_1} and ℓ_{a_1} and merging two sibling leaves ℓ_{b_1} and ℓ_{b_1} into ℓ_b.
We, first, prove 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.
Let us evaluate the difference between the variance on T and the variance on T'.

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

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:

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

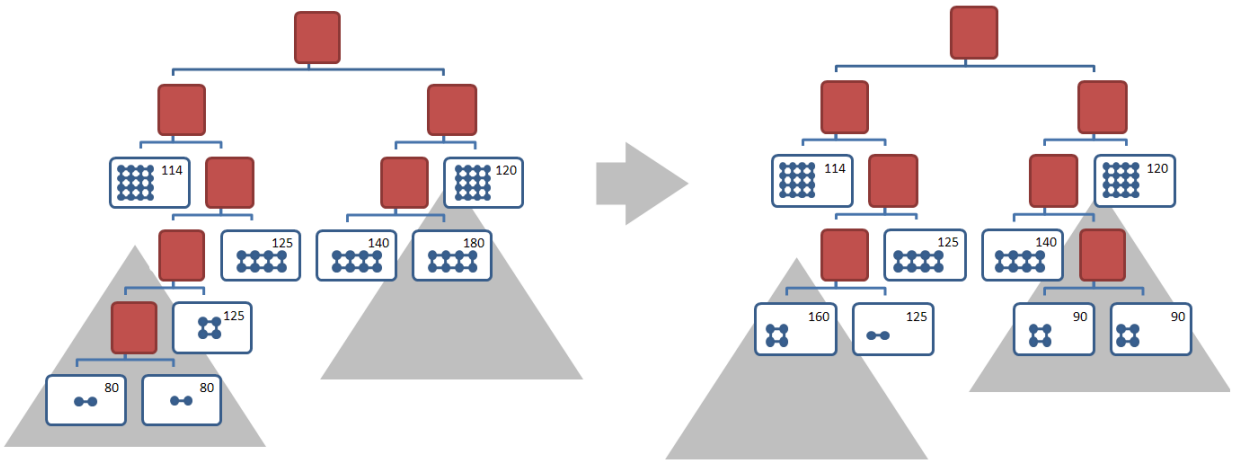

Figure 2: A merge and split operation on the PBT tree of Figure 1 where the estimation times $e(\ell)$ drive the updates.

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. □

Algorithm 1 PBT-Update

1: $T \leftarrow CurrentPBT$
2: **for all** $\ell \in L(T)$ **do**
3: copy computational time $t(\ell)$ in estimated time $e(\ell)$
4: **end for**
5: **while** true **do**
6: let ℓ_a be the leaf in T with max $e(\ell), \forall \ell \in L(T)$
7: let ℓ_{b_1}, ℓ_{b_2} be the two siblings such that $e(\ell_{b_1}) \cdot e(\ell_{b_2})$ is minimized over all the pairs of siblings in $L(T)$
8: **if** $e(\ell_a)^2 \leq 4\, e(\ell_{b_1}) \cdot e(\ell_{b_1})$ **then**
9: **return** T
10: **else**
11: Split ℓ_a in ℓ_{a_1} and ℓ_{a_2} // Now ℓ_a is internal
12: $e(\ell_{a_1}) \leftarrow e(\ell_a)/2$
13: $e(\ell_{a_2}) \leftarrow e(\ell_a)/2$
14: Merge ℓ_{b_1} and ℓ_{b_2} into ℓ_b // Now ℓ_b is a leaf
15: $e(\ell_b) \leftarrow e(\ell_{b_1}) + e(\ell_{b_2})$
16: **end if**
17: **end while**

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 1, and one of the updates of the PBT-Update algorithm is shown in Figure 2.

3 Case study: Parallel Ray Tracing

Ray Tracing [18, 20] is a widely used algorithm for rendering images aiming at a high realism. It is the core technique underlying several 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 ray tracing algorithm. For each pixel (x, y) in the final image a ray is casted from the virtual camera through the scene, it is called *primary ray*. If exists, the first object is determinate. Based on the intersection point, the surface properties, the position and the color of lights, the light intensity at the intersection point is computed. In the Whitted-style ray tracing [22] 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. Then, it is quite obvious that ray tracing is computationally intensive which is directly bound to the amount of rays shot throughout the scene.

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 1) and can be exploited to reduce the amount of calculations needed for every new frame [5] .

A common way of exploiting the temporal coherence is the *interpolation* [19, 6]: the amount of calculations needed to render pixel p' is reduced re-using (i.e. interpolating) information calculated for pixel p.

Parallel Ray Tracing. Ray tracing has been defined "embarrassingly parallel" [9] because no particular effort is

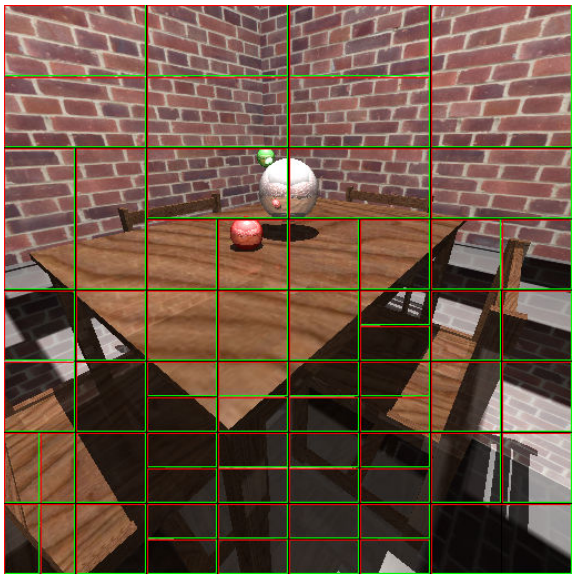

Figure 3: A frame in the walk-trough for scene ERW6, with the tiling shown.

needed to segment the problem in tasks and 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. There are two different approaches in designing a parallel ray tracer: object-based and screen-based [4]. 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 paper 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 [2] and the use of commodity PC clusters [21]. In particular, several techniques are employed to amortize communication costs and manage load balancing. In [21] is suggested a task prefetching and work stealing, whereas [8] is presented a distributed load balancer.

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 $\sqrt{t} \times \sqrt{t}$ mesh that represent 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, that is rectangular areas of pixels, that 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.

Implementation

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.

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 small changes to the tree that do 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 [13] for node communications, having care to disable Nagle algorithm [14] in order to decrease latency.

Experiments and Results

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 unbalancing between tiles.

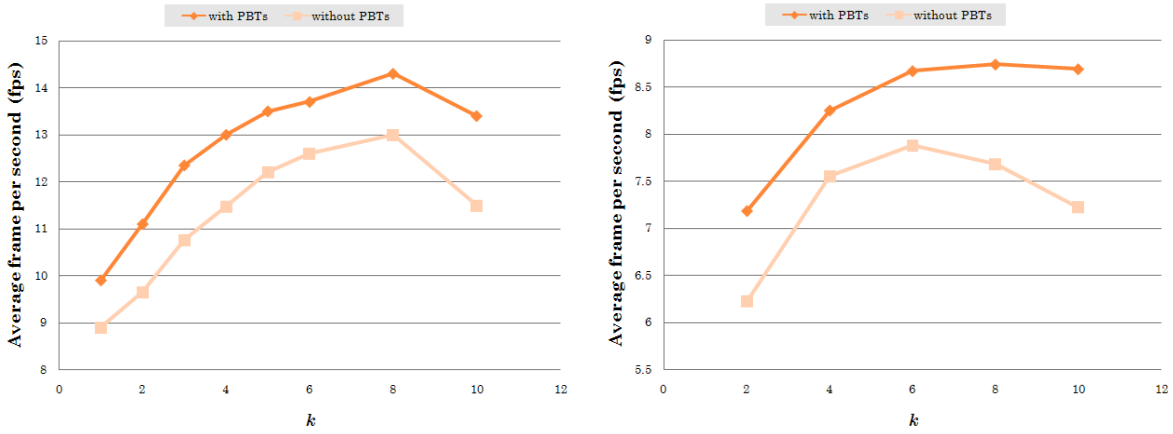

Figure 4: Frame rate on increasing k comparing static and semi-static (i.e., using the PBT) job assignments. (Left ERW6. Right ERW6-4 test scene)

Setting of the experiments. We ran several tests on two hardware platform:

Hydra: an IBM BladeCenter Cluster of 33 nodes (1 master node, 32 worker nodes). Each node has an Intel Pentium IV processor running at 3.20 GHz, with 1 GB of main memory and CentOS 5 Linux as operating system with OpenMPI version 1.1.1 for message passing. All the nodes are interconnected with a Gigabit Ethernet network.

Cacau: a NEC Xeon EM64T Cluster available at HLRS High Performance Computing Center at the Stuttgart Universität. We used up to 64 nodes, each equipped with 2 Intel Xeon EM64T processors and 1 GB of main memory, interconnected with a Infiniband 1000 MB/s.

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 3) (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 a walk-through of the camera around the scene, with movements in all directions and rotations too. The image resolution is 512×512 pixels.

Effectiveness of Prediction Binary Tree. In these tests, ran on Hydra, we evaluate the improvement provided by using the PBT instead of a static demand-driven balancing strategy.

The results are shown in Figure 4 for both scenes. Results are obtained using different granularity and $n = 32$ workers. Our technique offers a speedup 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 along 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, tiling algorithm may be a bottleneck. For this reason we implemented a simple limitation on the number of split/merge operations that gives a trade off between tile balancing and tile tree updating time.

Scalability. We compared our schema based on the PBT against a regular subdivision schema with 2, 4, 8, 16, 32 and 64 processors, in order to evaluate the efficiency of our strategy (see Figure 5). The tests, ran on Cacau, shows that our schema works always better than the regular one, and presents almost linear scalability. 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.

4 Conclusions and Future Works

We present a scheduling strategy that: (i) improves load balancing; (ii) allows to exploit temporal coherence among successive computation phases; (iii) minimizes the interprocessors 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. The PBT is effectively used to evaluate the load balance of each phase and, eventually, to update tasks assignment in order to reduce

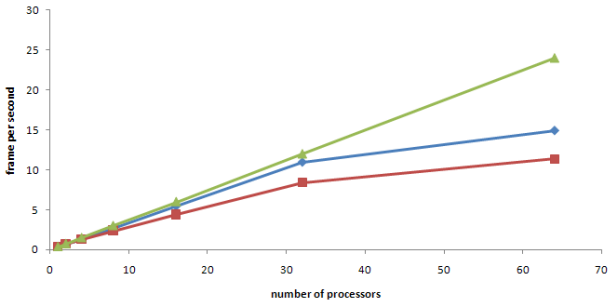

Figure 5: Scalability. Frame rates on increasing number of processors comparing our adaptive subdivision (blue), a regular subdivision (red) and the optimal linear speedup (green). ERW6-4 test scene.

their completion time.

We have proved that our strategy shows a good scalability, on a significant problem: Parallel Ray Tracing; it improves on the same problem without load balancing. We reported different measurements showing that the number of tiles is a key point for the performances of the system. Our proposed implementation is quite simple and leaves room for further investigations.

Possible improvements of the system can be obtained tweaking the PBT data structure. Our current implementation splits in halves unbalanced tiles; a fine grained split, based on the estimations, may give better results. Another possible extensions of our schema is to move toward a distributed PBT. This can also decrease the time required by PBT-Update that currently is serial. Indeed, the state of the art in load balancing [8, 3] has abandoned master-workers paradigm preferring more scalable distributed load balancing schema.

Finally, it maybe worth interesting to integrate our PBT based strategy with online approaches, like work-stealing: the purpose is to verify that such strategies can get benefit from a better load balancing (obtained by exploiting the PBT) and, hence, introduce smaller overheads.

References

[1] P. Berenbrink, T. Friedetzky, and L. A. Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32(5):1260–1279, 2003.

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

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

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

[5] 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.

[6] C. Chevrier. A view interpolation technique taking into account diffuse and specular inter-reflections. *The Visual Computer*, 13(7):330–341, 1997.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[8] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker. Memory sharing for interactive ray tracing on clusters. *Parallel Computing*, 31(2):221–242, 2005.

[9] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!* Morgan Kaufmann, May 1994.

[10] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, 1998.

[12] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

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

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

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

[16] M. Parashar and J. Browne. Distributed dynamic datastructures for parallel adaptive meshrefinement. In *Proceedings of the International Conference on High Performance Computing*, 1995.

[17] M. Parashar and J. 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.

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

[19] J. Sig Badt. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4(3):123–132, 1988.

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

[21] I. Wald, C. Benthin, A. Dietrich, and P. 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.

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

[23] L. Yang, J. M. Schopf, and I. 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.