Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 173–187. http://www.scpe.org
ISSN 1895-1767

EXPERIENCES WITH MESH-LIKE COMPUTATIONS USING PREDICTION BINARY TREES*

GENNARO CORDASCO† BIAGIO COSENZA† ROSARIO DE CHIARA† UGO ERRA‡ AND VITTORIO SCARANO†

Abstract. In this paper we aim at exploiting the temporal coherence among successive phases of a computation, in order to implement a load-balancing technique in mesh-like computations to be mapped on a cluster of processors. A key concept, on which the load balancing schema is built on, is the use of a Predictor component that is in charge of providing an estimation of the unbalancing between successive phases. By using this information, our method partitions the computation in balanced tasks through the Prediction Binary Tree (PBT). At each new phase, current PBT is updated by using previous phase computing time for each task as next-phase's cost estimate. The PBT is designed so that it balances the load across the tasks as well as reduces *dependency* among processors for higher performances. Reducing dependency is obtained by using rectangular tiles of the mesh, of almost-square shape (i. e. one dimension is at most twice the other). By reducing dependency, one can reduce inter-processors communication or exploit local dependencies among tasks (such as data locality). Furthermore, we also provide two heuristics which take advantage of data-locality. Our strategy has been assessed on a significant problem, Parallel Ray Tracing. Our implementation shows a good scalability, and improves performance in both cheaper commodity cluster and high performance clusters with low latency networks. We report different measurements showing that tasks granularity is a key point for the performances of our decomposition/mapping strategy.

Key words: scheduling, load balancing, performance prediction, mesh-like computation, performance evaluation.

1. Introduction.

1.1. Parallel Computing. The evolution of computer science in the last two 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 processors for minimizing the completion time (that is the time when the last processor completes 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 whole computation is usually represented via a *directed acyclic graph* (DAG) $\mathcal{G} = \{\mathcal{N}, \mathcal{E}\}$ which consists of a set of nodes $\mathcal{N}$ representing the tasks and a set of edges $\mathcal{E}$ representing interactions and/or dependencies among tasks.

The number and, consequently, the size of tasks into which a given computation is decomposed determines the *granularity* of the decomposition. It may appear that the time required to solve a problem can be easily reduced, by simply increasing the granularity of decomposition, in order to perform more and more tasks in parallel, but this is not always true. Typically, interactions between tasks, and/or other important factors, limit our choice to coarse-grained granularity. Indeed, when the tasks being computed are mutually independent, the granularity of the decomposition does not affect the performances. However, dependencies among tasks incur inevitable communication overhead when tasks are assigned to different processors. Moreover, the finer is the adopted granularity by the system, the more is the generated inter-tasks communication. The interaction between tasks is a direct consequence of the fact that exchanging information (e.g. input, output, or intermediate data) is usually needed.

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* [16] but it should be emphasized that dependency can also refer to other issues such as locality of access to memory (effective usage of caching).

The mapping problem becomes quite intricate if one has to consider that: (1) task sizes are not uniform, that is, the amount of time required by each task may vary significantly; (2) task sizes are not known a

*A portion of this paper was presented at ISPDC 2008 [8].

†ISISLab, Dipartimento di Informatica ed Applicazioni "R.M. Capocelli", Università degli Studi di Salerno, Salerno (Italy), {cosenza, cordasco, dechiara, vitsca}@dia.unisa.it.

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

FIG. 1.1. *Interaction between components of a Parallel Scheduler; arrows indicate how components influence each others: (left) Traditional approach; (right) Our system with the Predictor.*

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

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

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

1.2. 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.

We introduce an iterative novel approach to implement decomposition/mapping scheduling. Our solution (see Figure 1.1) introduces a new component in the system design, dubbed *Predictor* that is in charge of providing an estimation of the computation time needed by a given tile, at each "phase".

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

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

We validate our strategy by using interactive rendering with Parallel Ray Tracing [24] algorithm, as a significant example of such a kind of computations. In this example our technique is applied rather naturally. Indeed, interactive Ray Tracing can be seen as a step-wise computation, where each frame to be rendered represents a phase. Moreover, each frame can be described as a mesh of items (pixels) and successive computations are typically characterized by temporal coherence.

For Parallel Ray Tracing, our technique experimentally exhibits good performances improvements, with different granularity (size of tiles), with respect to the static assignment of tiles (tasks) to processors. Furthermore we also provided an extensive set of experiments in order to evaluate: (1) the optimal granularity with different number of processors; (2) the scalability of our proposed system; (3) the correctness of the predictions exploiting temporal coherence; (4) the effectiveness of the locality coherence heuristics exploiting spatial coherence; (5) the impact of resolution; (6) the overhead induced by the PBT.

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

1.3. Previous Works. Decomposition/Mapping scheduling algorithms can be divided into two main approaches: list scheduling and cluster-based scheduling. In list scheduling [18], each task is first assigned a priority by considering the position of the task within the computation DAG $\mathcal{G}$. Then tasks are sorted on priority and scheduled following this order on a set of available processors. Although this algorithm has a low complexity, the quality of scheduling is generally worse than that of algorithms in other classes. In cluster-based scheduling, processors are treated as clusters and the completion time is minimized by moving tasks among clusters [4, 12, 26]. At the end of clustering, heavily communicating tasks are assigned to the same processor. While this reduces inter-processor communication, it may lead to load imbalances or idle time slots [12].

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 effective [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 these environments, performance processors are both irregular and time-varying because of uneven underlying load on the various resources. In [25] authors use a conservative load prediction in order to predict the resource capacity over the future time interval. They use expected mean and variance of future resource capabilities in order to define an appropriate data mappings for dynamic resources.

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

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

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

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

On the opposite side, two following considerations suggest a smaller m. An algorithm that has large m requires larger *communication costs* than an algorithm with smaller m, considering both the latency (more messages are required; therefore, messages may be queued up) and the bandwidth (communication overhead for each message). Other considerations that would suggest to use small m are (a) the *locality of interaction* and (b) the *spatial coherence* that are motivated because (i) computation of a task relies usually on nearby tasks

and (ii) two close tasks usually access some common data. Then, in order to make an effective usage of the local cache for each node, it is important that the tiles are large enough, so that each worker can exploit spatial coherence of tiles, having a good degree of (local) cache hits.

Our strategy takes into account all the considerations above, by addressing the uneven spread of the load by using a Predictor component (the PBT), with a negligible overhead. The goal we aim to is to keep the load balanced without resorting to increase the number of tiles. Thereby, our solution does not increase significantly the data-movement overhead and it reduces tasks interactions. Therefore, we are able to address simultaneously and positively all the issues above, by providing a technique that uses a moderate amount of tiles.

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

Now we define the Prediction Binary Trees and then describe an on-line algorithm which, before each computing phase, resizes unbalanced tiles in such a way to minimize the mesh computing time. A PBT T stores the current tiling being defined as a rooted binary tree with exactly m leaves, in which each (internal) node has 2 children. The root of T, called r, represents the complete mesh. The two children of an internal node v store the two halves (more details follow on how the mesh is split) of the mesh represented by v. Consequently, each level of T represents a partition of the mesh. Moreover, each internal node v represents a tile which is the sum of the tile assigned to the leaves of the tree rooted in v and consequently, the leaves of T (henceforth $L(T)$) represents a partition of the mesh. In order to maintain a good spatial coherence and minimize tasks interaction, the children of an internal node v which belongs to an odd (resp. even) level of T are obtained halving the tile in t along two horizontal (resp. vertical) axes. This assures that tiles have an almost-square shape (i. e. one dimension is at most twice the other). Each leaf $ell \in L(T)$ also stores two variables: $e(\ell)$ that is the estimate of the time for computing tile in ℓ and $t(\ell)$ that is time used by a worker to compute (in the last phase) the tile in ℓ. Figure 2.1 gives an example of a PBT, with the corresponding mesh partition on the left.

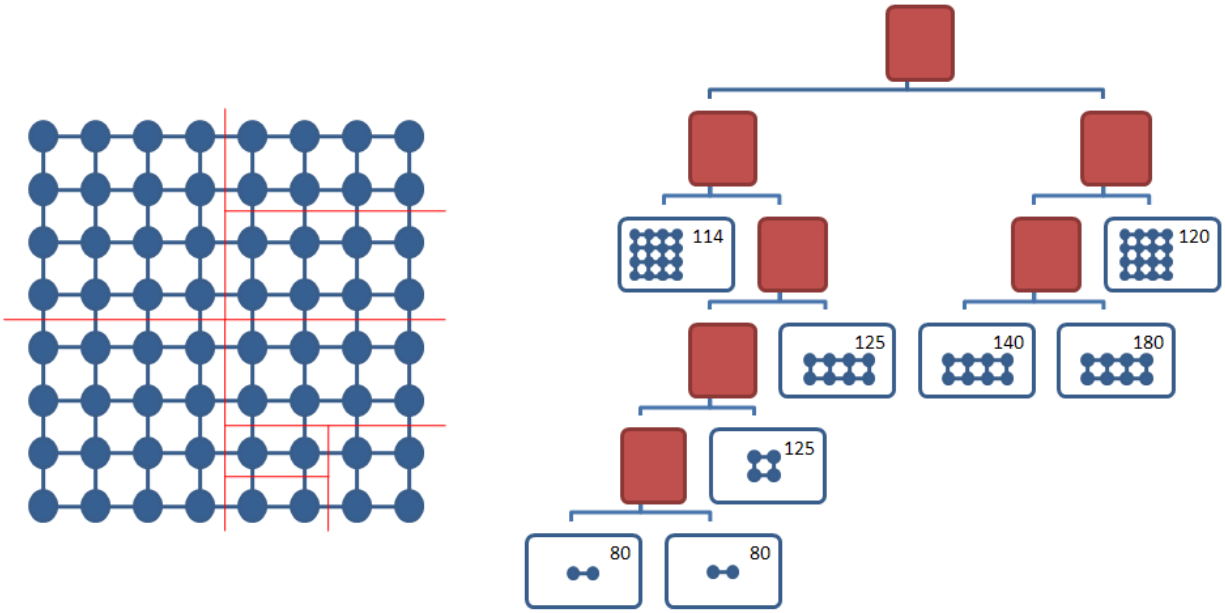

FIG. 2.1. *An example of a PBT tree: the mesh on the left has been computed with the computation times (in ms) for each tile shown on the leaves.*

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

First we define the variance as a metric to measure the (estimated) computational unbalance that is expected given the tiling provided by the PBT 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 2.1. *Algorithm PBT-Update terminates after a finite number of iterations.*

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

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

We prove first that, if $e(\ell_a)^2 > 4e(\ell_{b_1})e(\ell_{b_2})$ then $\sigma_T^2 > \sigma_{T'}^2$. So, it is not possible to improve the variance of the (estimation of the) computational time by means of a simultaneous split-merge operation if $e(\ell_a)^2 \leq 4e(\ell_{b_1})e(\ell_{b_2})$ which is the test in line 1 of the algorithm.

Algorithm 1 PBT-Update

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

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

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

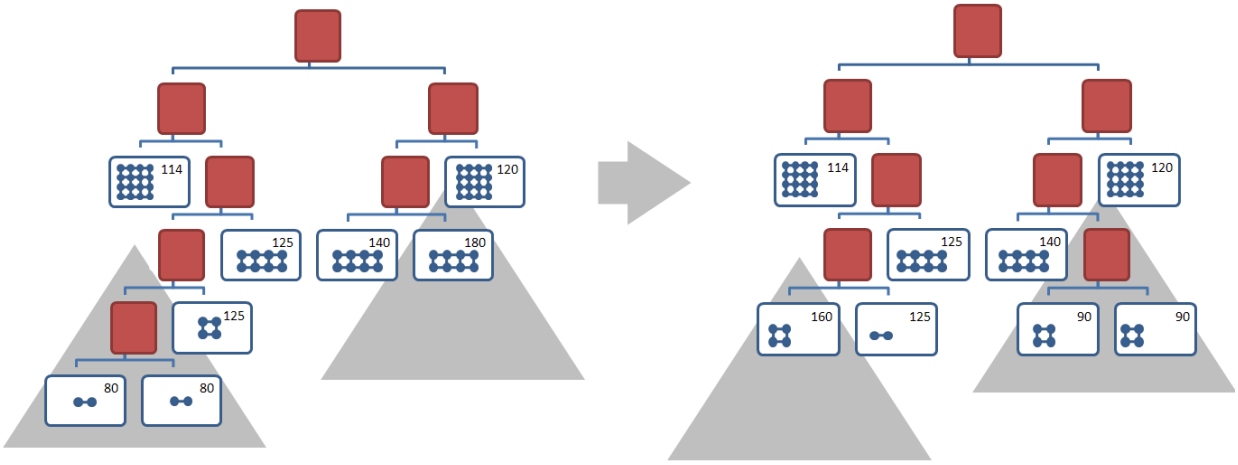

FIG. 2.2. *A merge and split operation on the PBT tree of Figure 2.1 where the estimation times $e(\ell)$ drive the updates.*

Hence, we have

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

Since, by the operations executed in lines 1-1 and 1 of the algorithm, it holds that $\sum_{\ell\in L(T)} e(\ell) = \sum_{\ell\in L(T')} e(\ell)$, then, we have that:

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

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

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 1-1) in order to have the higher (local) improvement in variance.

An example of a PBT is shown in Figure 2.1, and one of the updates of the PBT-Update algorithm is shown in Figure 2.2.

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

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

We implemented two heuristics in order to determinate affine tiles. The first heuristic is based on a greedy strategy, dubbed *PBT-Greedy*. For each computing phase, if a tile is not involved in a merge/split operation then it maintains his affinity with the processor it has been assigned to during previous phase. In case of tiles

involved in a merge/split operation, let's consider a tile a split in tiles a_1 and a_2 and tiles b_1 and b_2 merged into tile b: a_1 is assigned to processor that handled a before; b is assigned to processor that handled b_2 before; a_2 is assigned to processor that handled b_1. Just this last assignment, on a total of 3 assignments, will, probably, not exploit cache and for this reason we say that this heuristic is 2/3 effective in leveraging locality.

In the second heuristic, named *PBT-Visit* the affinity is defined by visiting the PBT: tiles are assigned following the in order visit. One can easily check that a subset of affine tiles, tiles "near" in the mesh, is assigned to the same processor phase-by-phase.

3. Case study: Parallel Ray Tracing. Ray Tracing algorithms [19, 20] are a widely used class of techniques for rendering images with the intent of achieving a high grade of realism. Ray Tracing is at the core of many global illumination algorithms. The input for Ray Tracing is a scene description that specifies the geometry of objects together with the definition of every object materials, position/orientation of the lights. The output is an image of the scene as seen through a virtual camera.

For sake of clarity we will shortly summarize the simplest form of Ray Tracing algorithm. For each pixel (x, y) in the final image a ray is casted from the virtual camera through the scene: this is called the *primary ray*. If an intersection between the primary ray and a surface is found then different parameters are considered in order to compute the light intensity at the point: the material of the surface, the position and the color of lights, whether or not the point is shadowed by other surfaces. In the Whitted-style Ray Tracing [24] a ray can be reflected and/or refracted according to surface properties and the process is repeated recursively with these new rays. At the end, the process adds the light intensities at all intersection points in order to get the final color of the pixel. Ray Tracing is considered a computationally intensive algorithm because it depends on the amount of rays shot throughout the scene and this amount can be easily increased by modifying lights properties, objects positions and materials.

Since its introduction several techniques have been explored to accelerate Ray Tracing. In animated scenes we report an interesting observation about the fact that a new frame can be very similar to the previous frame if the viewpoint did not change drastically. This similarity is an instance of the concept of *temporal coherence* (cft. Section 1) and can be exploited to reduce the amount of calculations needed for every new frame [6].

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

Parallel Ray Tracing. The Ray Tracing algorithm has been defined "embarrassingly parallel" [10] because no particular effort is needed to segment an instance of the problem in tasks considering that there is no strict dependency between parallel tasks. Each task can be computed independently from every other task in order to achieve a speed up by executing them in parallel. There are two different approaches in designing a parallel ray tracer: object-based and screen-based [5]. 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 [22]. In particular, several techniques are employed to amortize communication costs and manage load balancing. In [22] is suggested a task prefetching and work stealing, whereas [9] is presented a distributed load balancer.

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

to be mapped on the geometry and the number of light sources are elements that increase the computational complexity of a scene to be rendered.

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

The granularity of our decomposition strategy is chosen defining $m = k \cdot n$ where m is the number of tiles, n is the number of workers and k is multiplicative constant. The greater is k, the smaller is tiles sizes. We experimentally tested several values of k and found that no large k is needed since, after small values of k, performances degrades due to the higher communication cost (cf. Section 4.2).

4. Experiments and Results. Our serial implementation of Ray Tracing algorithm exploits some, but not all, optimizations techniques used by last cutting-edge ray tracers. Actually, the kind of serial implementation that is used is not relevant for our purposes. Special attention is paid to the acceleration structure. We use a Kd-tree built to minimize the number of traversal and intersection steps, done using the well-know Surface Area Heuristic [23]. Our Kd-tree implementation also provides a fast Kd-tree traversal by using a cache friendly data-layout [21].

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 would not affect much the overall performances.

We coded our system in C++, compiling it with Intel C++ Compiler 10.1 for Linux. We used MPI [13] for node communications, having care to disable Nagle algorithm [14] in order to decrease latency.

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

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 Universität Stuttgart. We used up to 64 nodes, each equipped with 2 Intel Xeon EM64T processors and 1 GB of main memory, interconnected with an Infiniband network.

ENEA: an IBM HS21 Cluster with 256 nodes available at CRESCO Project computing platform, Portici ENEA Center. Each node is equipped with 2 Xeon Quad-Core Clovertown E5345 at 2.33 GHz and 16 GByte RAM. The nodes are interconnected with an Infiniband network.

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

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

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

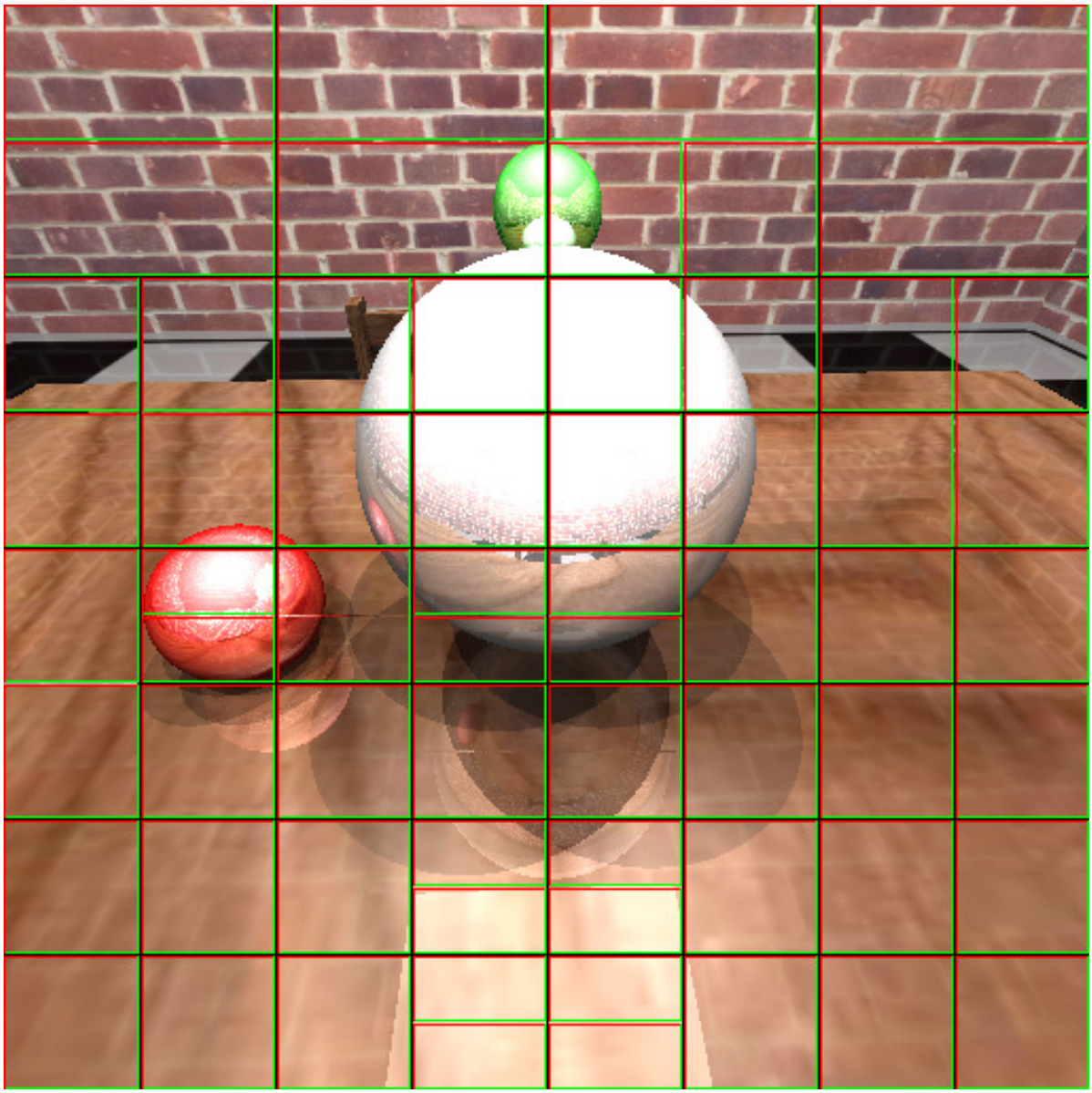

FIG. 4.1. *A frame in the walk-trough for scene ERW6-4, with the tiling shown.*

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

The results are shown in Figure 4.2 for both scenes. Results are obtained using different granularity and $n = 32$ workers. Our technique offers a speedup, ranging from 5 to 15 percent, for all the values of k tested. When k is large, the performances degrade due to two factors: the number of updates on the PBT increases. Our test shows that there are few updates for smaller values of k, but they grow quickly as k increases. The second is related to the heuristic that we have chosen. Indeed, measuring the rendering time for tiny tile has some approximation problems due to discretization. Our algorithm gives good performances for small values of m. For big values of m, the decomposition algorithm may be a bottleneck.

Optimal Granularity. We tried to estimate the optimal granularity of the schema, i. e., the optimal choice of the number of tiles m, with 8, 16, 32 and 64 processors (see Figure 4.3). The rationale behind this test is to determinate how our schema affects the tuning of the granularity in the parallel implementation, compared with the regular subdivision schema.

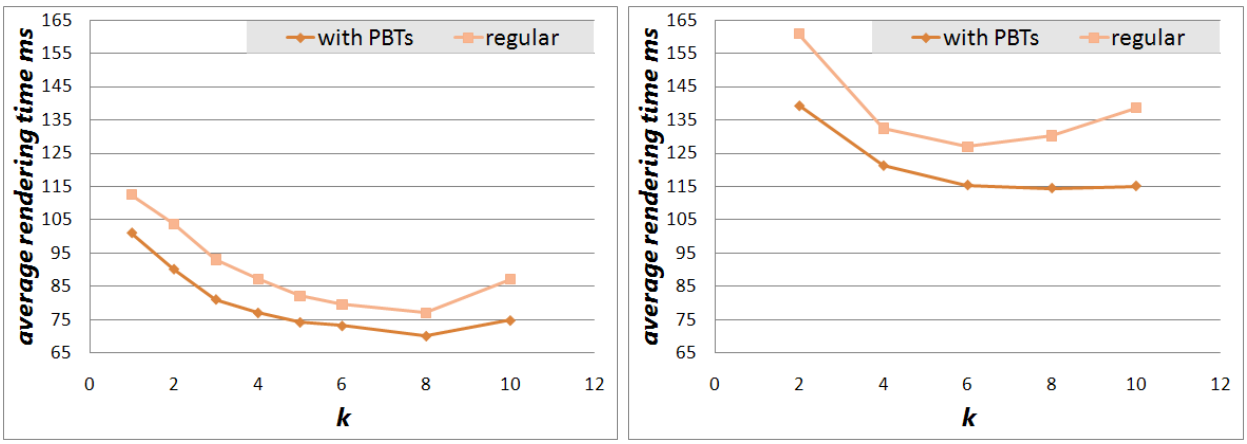

FIG. 4.2. *Average per frame rendering time on increasing k, comparing regular and PBT-based job assignments. (Left: ERW6. Right: ERW6-4)*

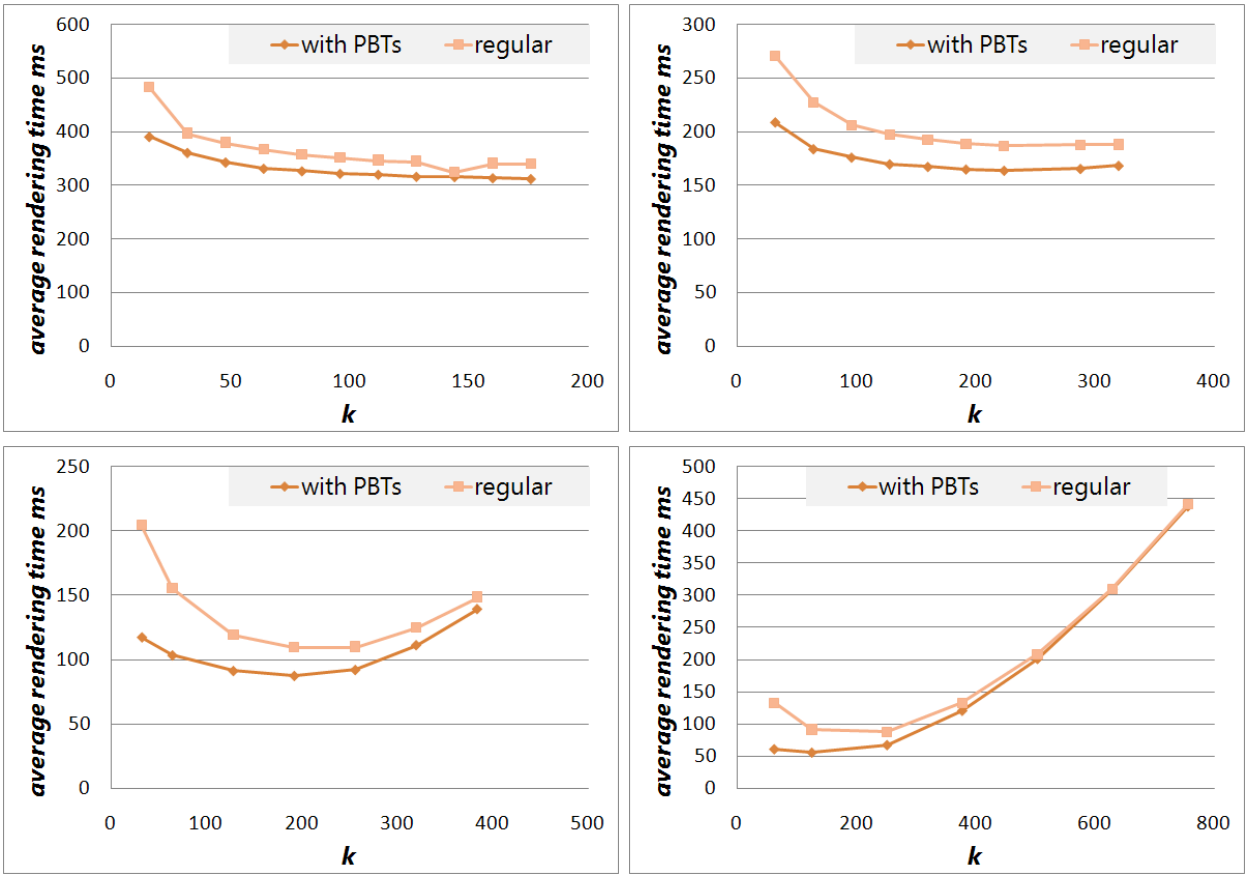

FIG. 4.3. *Optimal subdivision granularity with both regular and PBT subdivision. Test done with 8 (top-left), 16 (top-right), 32 (bottom-left) and 64 (bottom-right) processors. The horizontal axis represents the number of tiles, whereas vertical axis represents rendering time. ERW6-4 test scene.*

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

Scalability. We compared our schema based on the PBT against the regular subdivision schema with 2, 4, 8, 16, 32 and 64 processors, in order to evaluate the efficiency of our strategy (see Figure 4.4). The tests, ran on ENEA, shows that our schema works always better than the regular one, and presents almost linear scalability. In all the tests the granularity coefficient k is fixed to 4. However, the test with 64 processors also shows that when the number of tiles increases the use of an adaptive subdivision is still not enough in order to assure a good scalability. We conjecture that, in order to improve scalability, the value of k should be tuned in such a way that tile's sizes do not become extremely small.

Temporal coherence tests. In order to explore the performances of the PBT in exploiting temporal coherence, we checked it under different conditions. We ran a set of tests on Hydra in order to evaluate the correctness of the prediction made using the PBT. First we tested two different task granularity (we recall that we consider $m = k \cdot n$, where m is the number of tiles and n is the number of worker): we have chosen $k = 1$ (one tiles for each worker) and $k = 4$ (four tiles, on average, for each workers). Moreover, we considered two different camera speeds (1x and 2x). In all the tests the number of workers n is 32. To make a comparison, we measured the total amount of tiles which has been estimated correctly using 85^{th}, 90^{th} and 95^{th} percentile (see Table 4.1). As an example row 2 (Perc. 90^{th}) represents the percentile of estimations having an error up to 10%. In other words, when $k = 1$ and the camera speed is 1x the 93.2% of estimations have an error smaller than 10%, while when $k = 4$ and the camera speed is 2x the 79.8% of estimations have an error smaller than 10%.

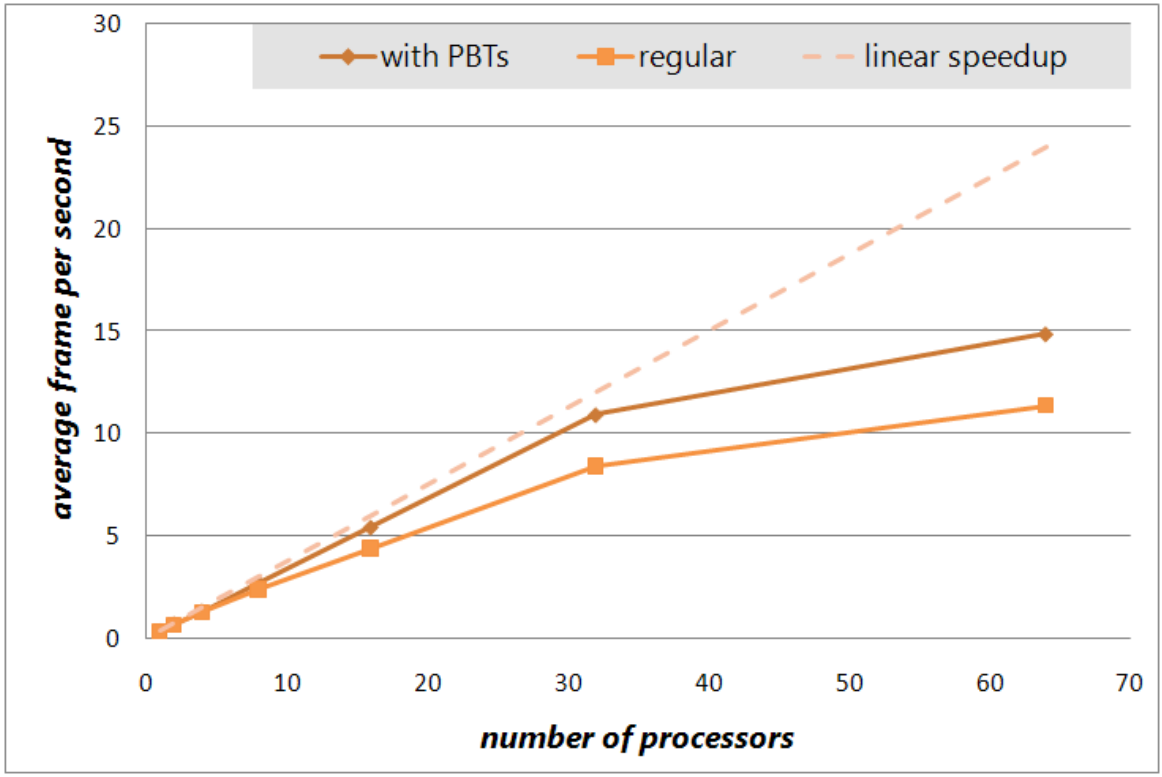

FIG. 4.4. *Scalability. Frame rates on increasing number of processors comparing the regular subdivision, our PBT-based subdivision and the optimal linear speedup. ERW6-4 test scene.*

Spatial coherence tests. In this paragraph we investigate how the PBT exploits data locality using the two locality-aware heuristic described in Section 2.1. The idea is to let workers to better use their CPU cache.

The test considers 4 different scenes with an increasing number of triangles, ranging from less than 30000 to about 950000. In Table 4.2 are reported the number of triangles and the number of nodes in the Kd-tree. The difference between scenes is an increasing number of amphitheaters, all of them with a simple diffusive shader (see Figure 4.5). Scenes are animated by a predefined walk-through of the camera.

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

TABLE 4.1
Results of the predictions in 85^{th}, 90^{th} and 95^{th} percentile.

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

TABLE 4.2
A simple description of the test scenes used for spatial coherence tests. For each scene, we report the corresponding number of primitives and the Kd-tree's size.

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

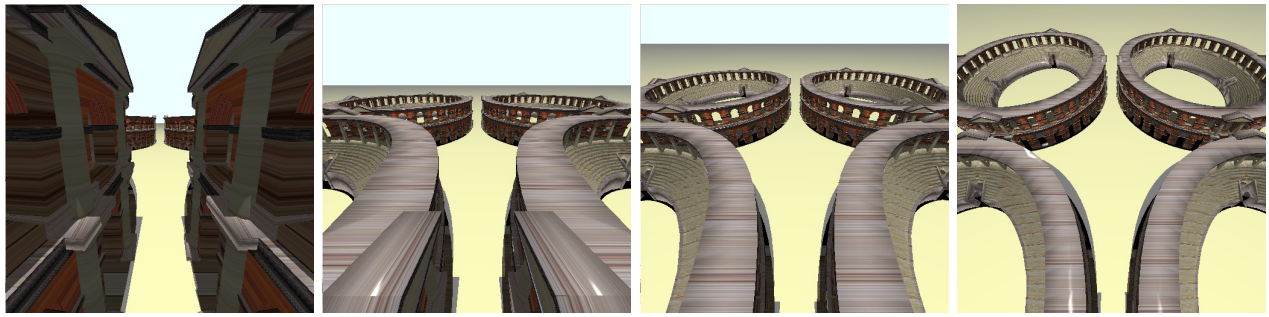

FIG. 4.5. *Amphitheater test scene, used in spatial coherence test set.*

All tests shows that improvements obtained by using the PBT (with both the *locality-aware* heuristics and a random assignment named *PBT-Random*) with respect to the regular assignment (cf. *Exploiting Local Coherence*, Subsection 2.1). With more details, on small scenes, the whole scene fits into the cache and then the assignment strategy does not matter. When the scene becomes larger, so that it does not fit into the cache, the data locality also provides a modest improvement of the performances of the system (around $1-5$ ms) using both the *locality-aware* heuristics. This test has been performed on Hydra.

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

The tests show that the PBT is effective with all the workloads and its ability in balancing the work between nodes is beneficial with heavy loads (see Figure 4.6).

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

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

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

Figure 4.7 shows the test results for 16 and 32 processors at different granularity. The time spent in updating the PBT is proportional to m, hence to the granularity and the number of processors, but in our test is always

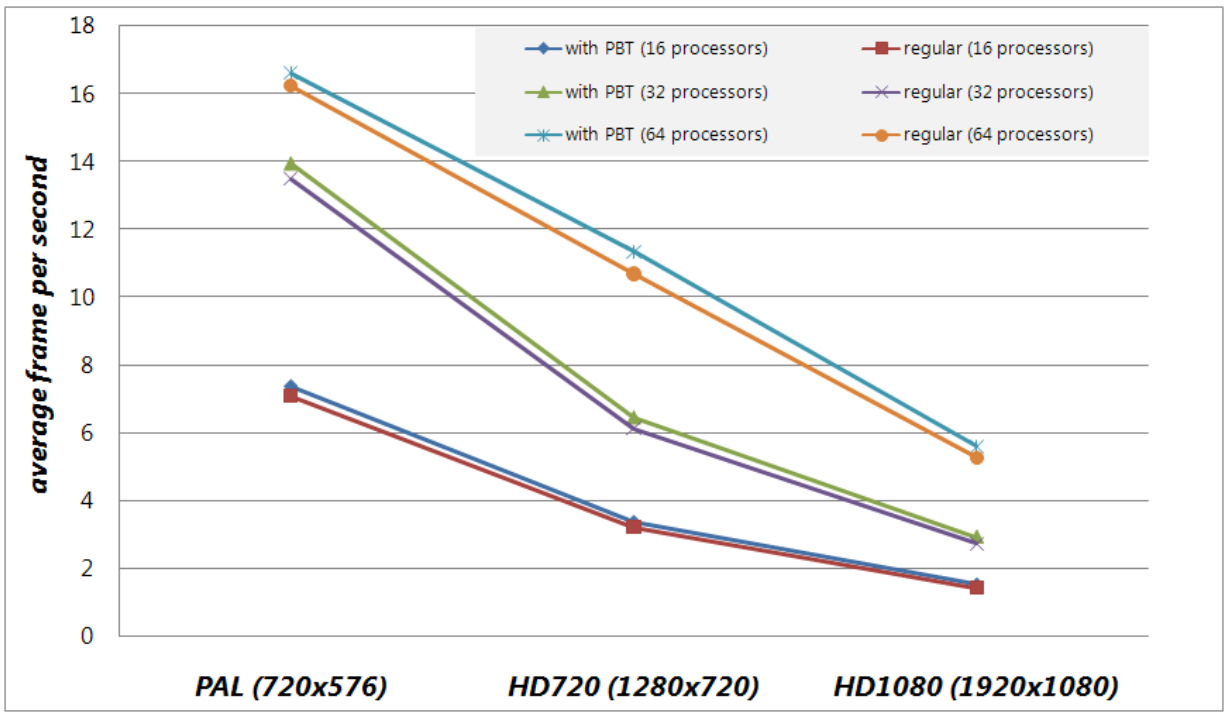

FIG. 4.6. *Average frame rate using different resolutions: PAL(*720×576*), HD720(*1280×720*), HD1080(*1920×1080*), comparing regular and PBT-based job assignments. ERW6-4 test scene.*

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

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

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

The predictions used in our approach are based on temporal coherence. We proved that for simple scenes (e.g. a predefined walk-through of the camera around the scene), the proposed estimation heuristic is affordable. Spatial coherence for data locality has been exploited by using two locality-aware heuristic during assignment. However we showed that such heuristic provides an improvement on performance only with larger scene (at least 1 million of triangles).

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

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

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

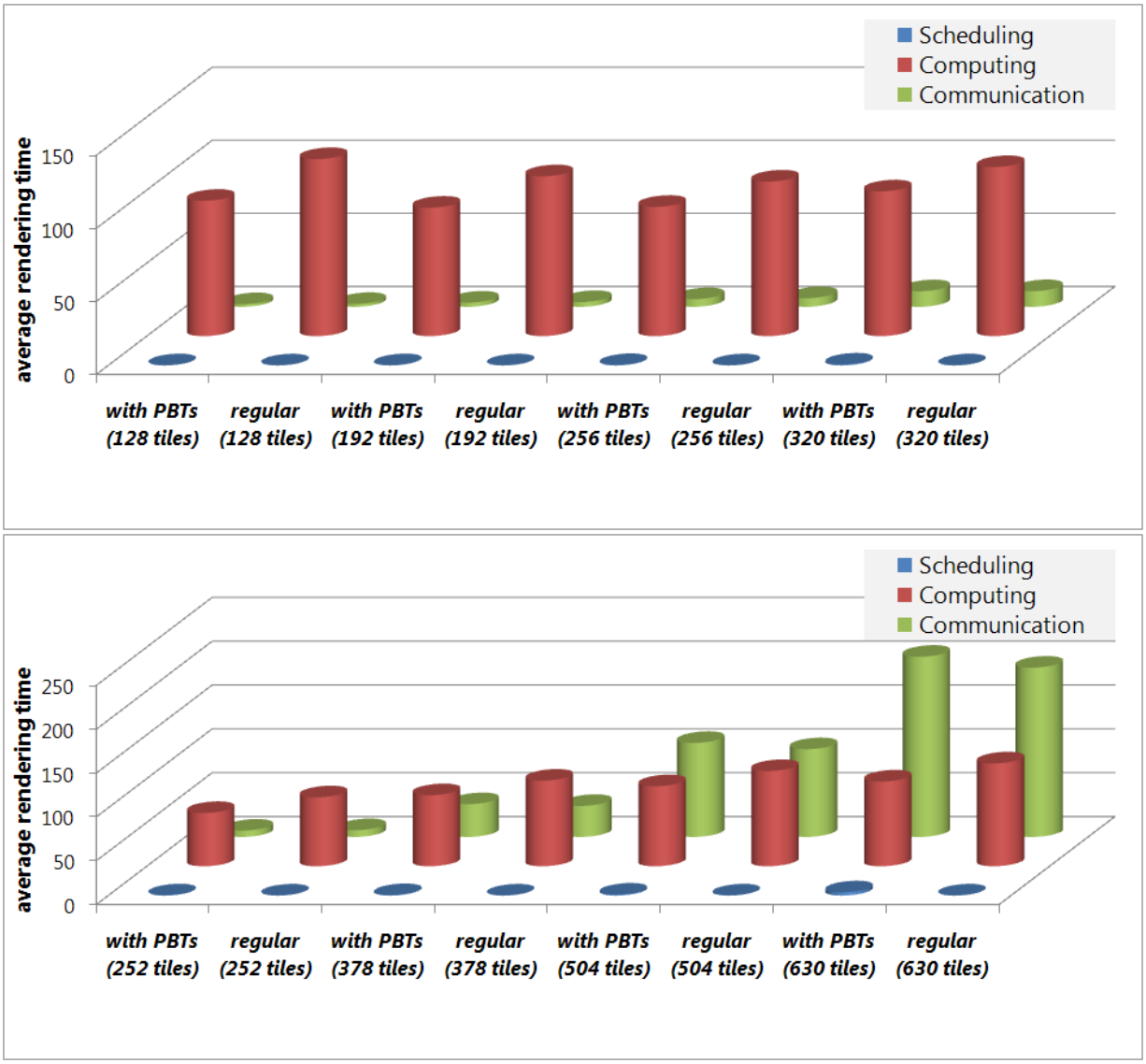

FIG. 4.7. *Contributions in rendering time with 32 (up) and 64 (down) processors, at different granularities, with both adaptive and regular subdivision techniques. The total rendering time is split in: the time spent in subdivision (i. e. update the PBT for adaptive subdivision); the maximum per-node compute time, such as an estimate of the load balancing; the remaining time, mostly due by communications. Rendering times shown are the average in a test scene (ERW6-4) of 600 frames.*

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

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

REFERENCES

[1] P. BERENBRINK, T. FRIEDETZKY, AND L. A. GOLDBERG, *The natural work-stealing algorithm is stable*, in SIAM J. Comput., 32(5):1260–1279, 2003.

[2] J. BIGLER, A. STEPHENS, AND S. G. PARKER, *Design for parallel interactive ray tracing systems*, in IEEE Symposium on Interactive Ray Tracing, 0:187–196, 2006.

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

[4] C. Boeres and V. Rebello, *Cluster-based static scheduling: Theory and practice*, in Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing (SCAB-PAD'02), page 133, Washington, DC, USA, 2002. IEEE Computer Society.
[5] A. Chalmers and E. Reinhard, editors. *Practical Parallel Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.
[6] 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.
[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
[8] B. Cosenza, G. Cordasco, R. De Chiara, U. Erra, and V. Scarano, *Load Balancing in Mesh-like Computations using Prediction Binary Trees*, in International Symposium on Parallel and Distributed Computing (ISPDC '08), pages 139–146, ISBN: 978-0-7695-3472-5, 2008. IEEE Computer Society.
[9] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker, *Memory sharing for interactive ray tracing on clusters*, in Parallel Computing, 31(2):221–242, 2005.
[10] G. C. Fox, R. D. Williams, and P. C. Messina, *Parallel Computing Works!*, Morgan Kaufmann, May 1994.
[11] K. Hwang and Z. Xu., *Scalable Parallel Computing: Technology, Architecture, Programming*, McGraw-Hill, 1998.
[12] Y. Kwok and I. Ahmad, *Benchmarking and comparison of the task graph scheduling algorithms*, in 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*, in IEEE Transaction on Computer, 37(9):1073–1087, 1988.
[16] M. Parashar and J. C. Browne, *Distributed dynamic data-structures for parallel adaptive mesh refinement*, 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 Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture. IEEE Computer Society, 1996.
[18] G. L. Park, B. Shirazi, and J. Marquis, *Mapping of parallel tasks to multiprocessors with duplication*, in Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS'98) Volume 7, page 96, Washington, DC, USA, 1998. IEEE Computer Society.
[19] P. Shirley and R. K. Morley, *Realistic Ray Tracing*, A. K. Peters, Ltd., Natick, MA, USA, 2003.
[20] K. Suffern, *Ray Tracing from the Ground Up*, A. K. Peters, Ltd., Natick, MA, USA, 2007.
[21] I. Wald, *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.
[22] 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.
[23] I. Wald and V. Havran, *On building fast kd-trees for ray tracing, and on doing that in O(N log N)*, in Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pages 61–69, 2006.
[24] T. Whitted, *An improved illumination model for shaded display*, in Communications of the ACM, 23(6):343–349, 1980.
[25] L. Yang, J. M. Schopf, and I. Foster., *Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments*, in Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC'03), page 31. IEEE Computer Society, 2003.
[26] T. Yang and A. Gerasoulis, *Dsc: Scheduling parallel tasks on an unbounded number of processors*, in IEEE Transactions on Parallel and Distributed Systems, 5:951–967, 1994.

Edited by: Marek Tudruj, Dana Petcu
Received: February 5th, 2009
Accepted: June 28th, 2009