Evaluation of Parallel Raytracing Strategy Improvements by Petri Nets

KOREČKO Štefan, SOBOTA Branislav, JANOŠO Radovan
Department of Computers and Informatics,
Technical University of Košice, Faculty of Electrical Engineering and Informatics,
Letná 9, 041 20 Košice, Slovakia, E-Mail: stefan.korecko@tuke.sk

Abstract ***–In this paper we deal with an evaluation of possible improvements of a parallel raytracing implementation, developed at the home institution of authors. The improvements focus on a reduction of delays caused by a communication between computational nodes. The evaluation is performed by simulation-based performance analysis using timed Coloured Petri nets (CPN) and CPN tools. We present a CPN model of our implementation and results from an analysis of proposed improvements.***

Keywords: ***simulation, coloured petri nets, parallel computing, raytracing.***

I. INTRODUCTION

Photorealistic imaging makes it possible to display 3D scenes with a high degree of realism. Techniques used in the photorealistic imaging allow to compute resulting picture following the descriptions of the environment and scene optical properties [3]. These techniques simulate the real optical effects, such as light reflection, refraction and diffusion. There are two methods of the photorealistic imaging:

- *raytracing* - this method is based on the tracing of every beam of light from an observer's position to the 3D scene trough a projection plane. In this case a computer display defines the projection plane.
- *radiosity* - this method computes the resulting scene image with respect to energy relationships inside the 3D scene.

Both methods are very time-consuming. To reduce the time needed in an extensive way a parallel raytracing computation is used. [10]. The parallelism is possible at the process or processor level. For parallel implementation we can use multicomputer (e.g. grid, cluster) or multiprocessor computer architecture (e.g. GPGPU technology) [9][12]. Because the raytracing is a recursive algorithm, the synchronisation and management of the distributed processes is relatively difficult [11]. In this case we have to solve problems of the communication and management between computation nodes. The management methods can be centralised or decentralised, deterministic or stochastic. They differ in dependency throughput on load, number of nodes, output resolution, realisation difficulty and scene complexity [2][3][4].

There are two principal methods of decomposing a raytracing computation: *demand-driven* and *data-driven* (or data-parallel), and there are research activities focused on developing a hybrid model trying to combine the best features of both methods [10]. Demand-driven parallel raytracing computes the final product of a raytracer as an image of $m \times n$ pixels, and since each pixel is computed independently, the most obvious way of decomposition is to divide the image into p parts, where p is a number of computational nodes available. Then each node will compute $m \times n/p$ pixels and, ideally, the computation would be p-times faster. A number of jobs are created, each containing different subset of image pixels and these jobs are assigned to computational nodes. Input scene is copied to local memory of each computational node. Nodes render their parts, return computed pixels, get another job if there is any, and in the end the final image is composed from these parts.

Data-driven parallel raytracing splits the input scene into a number of sections (tiles) and assigns these sections to processors. Each computational node is responsible for all computations associated with objects in this particular section, no matter where the ray comes from. Only rays passing through the node's section are traced. If a ray spawned at one node needs data from another node, it is transferred to that processor. The way the scene is divided into sections determines the efficiency of parallel computation. Determining the number of rays that will pass through a section of the scene in order to estimate the sections requiring the most processing is one of the hardest problems to overcome. Using the cost function can be helpful.

Our solution [12] implements the demand-driven model and uses a multicomputer (cluster) environment. It has a hierarchical structure, but it isn't a typical master/slave scenario. All nodes are equal, with the exception of the root (master) node. Master also controls the whole operation, allocates jobs and interacts with the user. That allows us to utilize massive parallelism. For load balancing, static or dynamic load balancing by tiling decomposition seems to be the best choice. Main benefits of this approach are easy decomposition and implementation, simple job distribution and control and the fact that a general raytracing algorithm remains unchanged and it scales well. The main disadvantage is

that the input scene has to be copied to local memory of each computational node, which poses a problem if the scene is very large.

In short, the raytracing in our implementation proceeds as follows: First the master node sends a 3D scene data to each slave node. Then the master divides the image plane into rectangular tiles and orders each node to process (raytrace) a tile. The master itself also raytraces a tile, but the raytracing job has a lower priority than a communication with slave nodes. When a slave node finishes tile raytracing, it notifies the master. Then the master downloads the raytraced tile and orders the slave to process another one. After processing of all tiles the master assembles the final image.

The implementation is quite satisfactory for a small number of nodes, but it showed up that it will be ineffective to use more than 10-11 nodes. This is, among others, caused by a different complexity of individual tiles and the communication delay.

In the rest of this paper we deal with improvements aimed at reducing the communication delay and their evaluation using simulation-based performance analysis. We present a timed Coloured Petri net model of our implementation and results from an analysis of proposed improvements.

II. CPN MODEL OF PARALLEL RAYTRACING IMLEMENTATION

Because it is a time and money consuming process to test intended improvements using real software and hardware, we decided to use an appropriate Petri net model instead. Petri nets are a well-known formalism, are able to express non-determinism and concurrency, and have been used in a wide variety of areas including classic ones, such as verification of network protocols or not so typical ones, such as e-learning [6]. There are also Petri nets dialects suitable for discrete-event simulation. We chose timed Coloured Petri nets (CPN) [1], [7] that combine a sufficient modelling power, probability functions and time concept. Its supporting tool, called CPN tools [1], is available for free and provides facilities for collecting and basic processing of simulation data. Because of space limitations we excluded a description of CPN from this paper. Reader can find basic information about CPN and CPN tools in [7], at [1] or in a short form in [8].

Figure 1 shows a timed CPN model used to evaluate possible improvements of our current implementation of parallel raytracing. The model is a further development of a basic model, introduced in [8]. The model incorporates the whole raytracing process with a special emphasis to communication between master and slave nodes. To achieve an adequate precision the time in our model is measured in milliseconds.

The net begins its lifecycle in the initial marking with tokens in places *scStartTime*, *newScene*, *commTime*, *tilesDist* and *freeNodes*. The place *newScene* holds one token with randomly chosen value from interval 10000 to 70000 (computed by the function *discrete*). This value characterizes a complexity of a scene to be raytraced and its range is based on our practical experience. In general the scene complexity depends on its size, number of objects, objects complexity (number of polygons), objects material (opacity, mirrors,…), illumination model and camera parameters. The number of all computers (nodes) used is given by the constant *ndNo* and the place *freeNodes* holds one token for each node, where its value designates a type of the node. Albeit all the nodes are equal we have to distinguish between the slave (client) nodes (type *2*) and the master node (type *1*) that also manages the whole process. About 4% of master performance is reserved for the management. The *tilesDist* holds one token with an uninitialized tiles distribution structure, created by the function *emptyTDist*. Token from *commTime* is used in the process of sending a raytraced tile back to the master node.

There is only one transition that can be fired in the initial marking in time=0 - *sendScene*. Its firing represents a sending of the whole scene to each slave node. Sending of the scene is a sequential process and its duration is computed by the function *sendTm*

*sendTm(ndNo)=(ndNo -1)*rnNormal_int(20000,10000)*

where the function *rnNormal_int(m,v)* returns a value from exponential random distribution with mean *m* and variance *v*. The firing also saves the starting time point of scene raytracing as a token in *scStartTime* and initializes the structure in *tilesDist* with information about scene partitioning (number, dimensions and overall complexity of tiles to be raytraced).

A firing of the transition *selectNewTile* means an assignment of a raytracing job to a free node *nt*. Of course, *selectNewTile*, can be fired only if there is some unprocessed tile of the scene (i.e. if the field *remTiles* of the structure in *tilesDist* is greater than 0). A tile to be raytraced is generated from the scene partitioning structure *td* by the function *getTile* and is sent to the place *prepTile*. To store information about the tile the colour set *TILE* is used (Fig. 2), where fields *wdt* and *hgt* store tile dimensions, *complxt* stores tile complexity, *cSuc* determines whether the tile raytracing will be successful, *nTyp* is a type of node where the tile will be raytraced and *cTm* is a time needed to return the tile to the master node after raytracing (a part of the communication delay). Tile dimensions and complexity are set by *getTile*. The tile complexity is selected randomly from an overall scene complexity using normal distribution. Fields *cSuc* and *nTyp* are set by the functions *setSuc_nTp* and *success* (Fig. 3), where a uniform random distribution is used to determine a raytracing job success. In our model we assume that

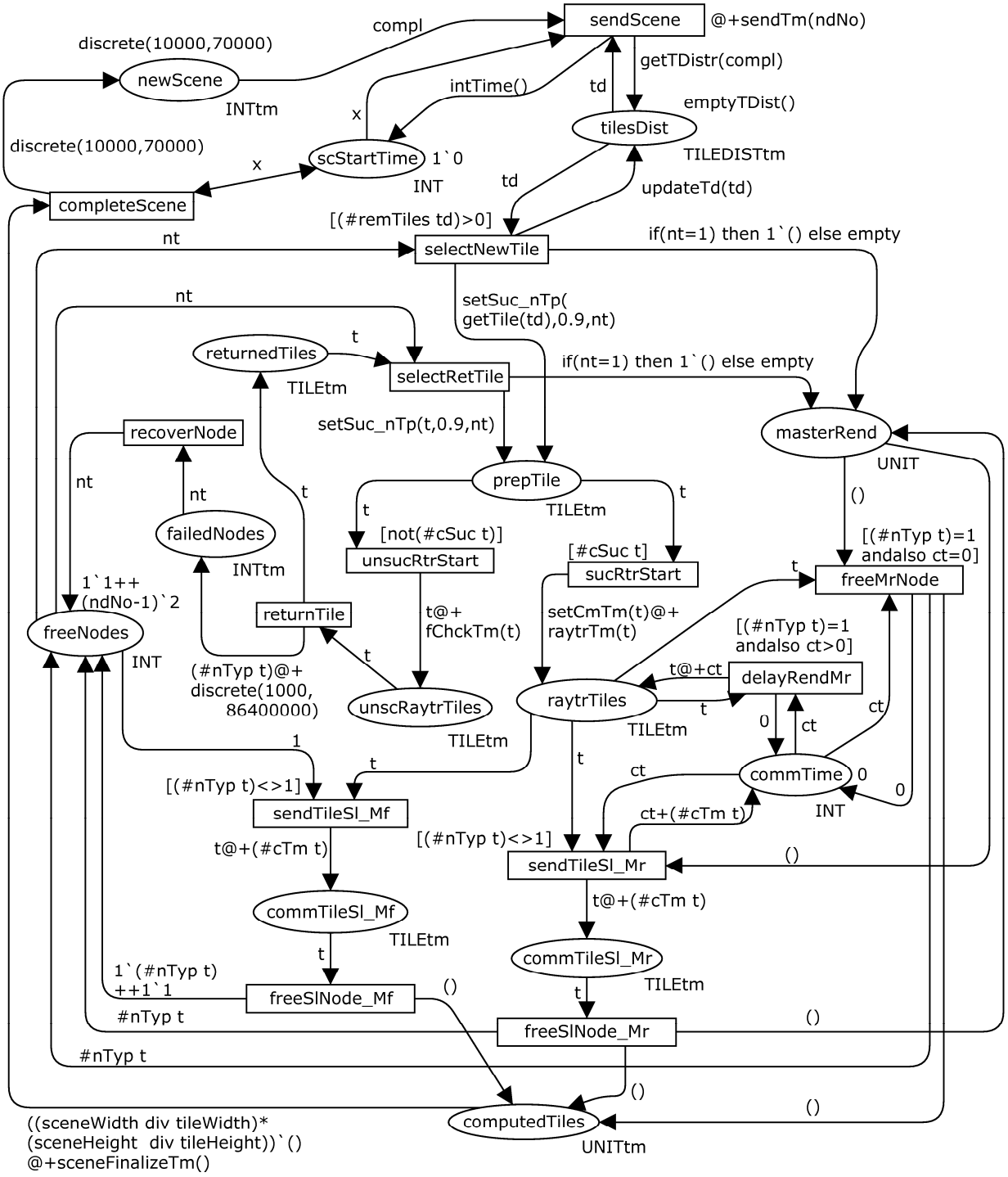

Fig. 1. CPN model of parallel raytracing implementation.

```
colset TILE = record
       wdt:INT *
       hgt:INT *
       complxt:INT *
       cSuc: BOOL*
       nTyp:INT*
       cTm:INT;

colset TILEtm = TILE timed;
```

Fig. 2. Declarations of some colour sets.

90% of all jobs on slave nodes will be successful and that the master node never fails. The firing of *selectNewTile* also updates the structure in *tilesDist* by decreasing the field *remTiles* and the overall complexity of remaining tiles. A presence of token in *masterRend* indicates that the master node has a raytracing job assigned, so the firing adds a token to *masterRend* when *nt=1*.

If the field *cSuc* of the generated tile *t* is true (i.e. *[#cSuc t]*), then a firing of *sucRtrStart* moves *t* to *raytrTiles*. The timestamp of *t* is also increased by

raytracing time and a communication delay is set to its field *cTm*. The raytracing time of the tile is computed by *raytrTm* from all fields of *t* except *cSuc* and *cTm*. The communication delay, computed by *setCmTm*, is derived from an actual network speed, hard disk speed of the master and size of *t*.

```
fun setSuc_nTp(t:TILE,rate,1) =
      {wdt=(#wdt t),
       hgt= (#hgt t),
       complxt= (#complxt t),
       cSuc = true,
       nTyp=1,
       cTm=(#cTm t)}

   |setSuc_nTp(t:TILE,rate,nt) =
      {wdt=(#wdt t),
       hgt= (#hgt t),
       complxt= (#complxt t),
       cSuc = success(rate),
       nTyp=nt,
       cTm=(#cTm t)}


fun success(rate)=
       let  val rn = uniform(0.0,1.0)
       in
         if(rn<=rate) then true
         else false
       end;
```

Fig. 3. Definition of functions setSuc_nTp *and* success.

After raytracing on a slave node the tile *t* is sent to the master. This is modelled by a firing of *sendTileSl_Mf* or *sendTileSl_Mr*.

The transition *sendTileSl_Mf* is used if the master doesn't perform a raytracing job when the tile is about to be sent. In this case the master node is removed from free nodes and *t* moves to *commTileSl_Mf*, where it waits for the communication delay time. Then a firing of *freeSlNode_Mf* moves the tile into already computed ones (place *computedTiles*) and frees the master and the node used for raytracing of *t*.

If the master is already performing a raytracing job, *sendTileSl_Mr* is used instead. The situation is similar to the previous case but, in addition, the communication delay of *t* is added to the value of token in *commTime*. The transitions involved also change the marking of *masterRend* to simulate that the raytracing on the master is interrupted when a communication with a slave occurs (because of a higher priority of the communication).

After raytracing on the master node there is no need to send *t* anywhere (except of the place *computedTiles*). However, in this case we have to take into account that the raytracing can be often interrupted due to communication with slaves. This is modelled by firing of *delayRendMr*, which adds a communication delay, accumulated during raytracing on the master in the place *commTime*, to the timestamp of *t* and returns it to *raytrTiles*. If there is no delay, *freeMrNode* finishes the raytracing job on the master.

The path of an unsuccessful tile begins with a firing of *unsucRtrStart*, which moves *t* to *unscRaytrTiles*. The delay computed by *fChckTm* is a time needed to detect that a given node failed and is not responding. The response of nodes is checked regularly in our implementation, so the delay computed is a randomly chosen multiple of checking period with some upper limit. Next, a firing of *returnTile* moves *t* to the place *returnedTiles* and the failed node to *failedNodes*, where it waits for recovery. We optimistically suppose that each node recovers within one day. Finally the node is returned to *freeNodes* by a firing of *recoverNode* and a raytracing of the tile *t* starts again by a firing of *selectRetTile*.

After successful processing of all tiles the final image can be assembled and the transition *completeScene* fired. Its firing removes all tokens from computedTiles and generates a new one in *newScene*, so a raytracing process can start over again. There is a data collecting monitor associated with firing of *completeScene*, which saves information about raytracing duration and number of used nodes into the text file for further processing.

III. EVALUATION OF MASTER/SLAVE BUFFERING USING SIMULATION

As it was mentioned earlier, one of possible improvements of our current parallel raytracing solution is to reduce the communication delay. This can be done by adding some memory buffers to the master or slave nodes. We decided to deal with a buffer on the master first, because it is easier to implement and we don't need to consider a risk of losing computed data, which is not the case when we have buffers on the slaves.

In general, use of a memory buffer on the master reduces the communication delay by approximately the time needed to save a raytraced tile to hard disk of the master node. To determine whether an implementation of master node buffer will be of any advantage, we performed a simulation experiment comparing time of a scene raytracing with and without the master hard disk saving time in the communication delay. The experiment was conducted on a slightly modified version of the model introduced in section II. The modifications reduced side effects by making all nodes equal (the master can use all of its performance for raytracing) and by ensuring that no computation fails (each tile has *cSuc=true*). They also guarantee that in both cases the same tiles (with same dimensions and tile complexity) are processed in the same order. In addition, we changed the place *computedTiles* from the collector of raytraced tiles to their counter. This change reduced the simulation time significantly. The model used for the simulation experiment is shown in Fig. 4 and it can be divided into 3 parts. The upper part simulates the raytracing with the buffer the bottom part simulates the raytracing without the buffer and the middle part deals with synchronized tile production and distribution.

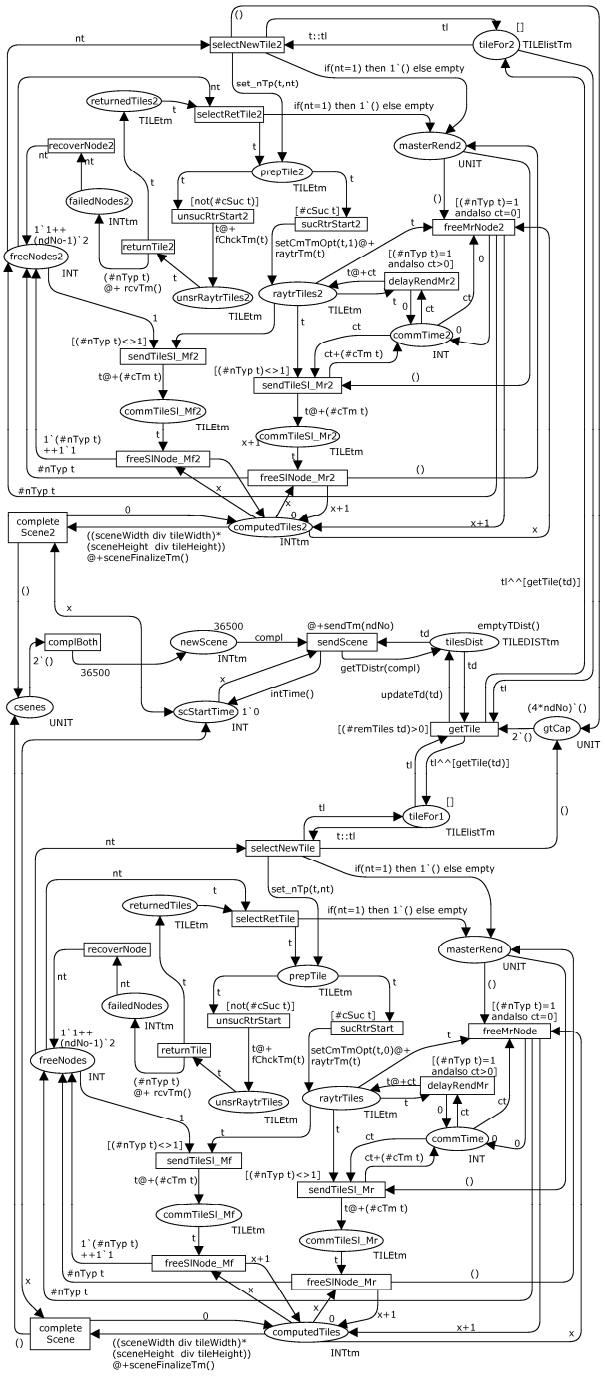

Fig. 4. Timed CPN model used for simulation experiments.

In the experiments we fixed the scene complexity to 36500 and used two different scenes

- a big scene, SC.1, with 30000×22500 pixels and
- a smaller one, SC.2, with 10000×7500 pixels.

Tile dimensions were 30000×10 pixels for SC.1 and 10000×10 pixels for SC.2. Network speed was set to 100Mbps and hard disk speed of the master to 300Mbps, according to the SATA2 specification. Number of the nodes varied from 3 to 16. The results, which are averages from 100 multiple simulation runs, can be seen in Table 1 and figures 5, 6. As it is clear from the results, use of the memory buffer on the master doesn't bring the desired improvement. It saves only a few seconds from the processing that takes a lot of hours. In some cases the raytracing duration with the buffer was even longer than without it. Our simulation experiments with buffers on the slave nodes showed similar results.

TABLE 1. Time saved by elimination of the hard disk saving time (in seconds) compared with scene raytracing duration (in hours)

no-des	SC.1 raytracing		SC.2 raytracing	
	duration (h)	*saved (s)*	duration (h)	*saved (s)*
3	228,8	*7,6*	25,5	*2,6*
4	171,9	*16,3*	19,2	*1,9*
5	137,7	*32,8*	15,4	*3,3*
6	114,9	*29,5*	12,9	*3,3*
8	98,8	*7,6*	11,2	*1*
7	86,6	*7*	9,8	*1,6*
9	77	*27,24*	8,8	*0,8*
10	69,4	*34,4*	7,9	*1,4*
11	63,3	*2,1*	7,3	*0,4*
12	58,1	*2,5*	6,7	*0,4*
13	53,8	*3,7*	6,2	*0,9*
14	49,9	*2,2*	5,8	*0,6*
15	46,7	*46*	5,5	*2,6*
16	43,9	*3*	5,1	*0,5*

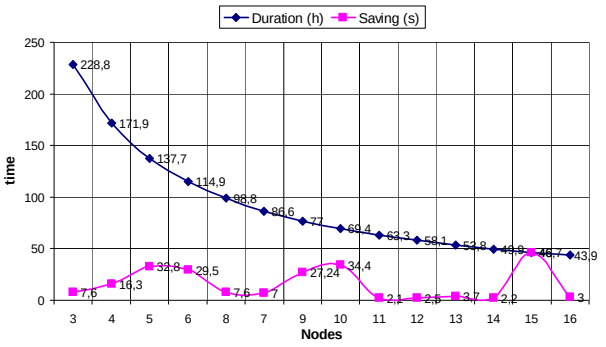

Fig. 5. Results of parallel raytracing simulation for SC1.

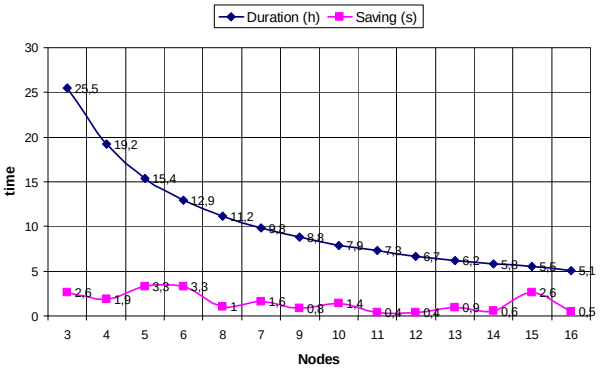

Fig. 6. Results of parallel raytracing simulation for SC2.

IV. CONCLUSIONS

Albeit we have to conclude that some of the intended improvements turned out to be unsatisfactory, it is needed to say that the use of discrete event simulation of

the CPN model instead of experiments on real hardware and software saved us a lot of time and money.

It should be also mentioned that to verify functional properties of the CPN model we also designed similar low-level Petri net model (PT net), where we used place invariants to check that a model has desired properties. In addition, we used our own tools and results [5] to transform the PT net model to the language of B-Method, where additional properties, such as deadlock freedom, has been verified.

Our future research will focus on the inefficiency caused by a different complexity of individual tiles, namely on the development of a method to estimate the tile complexity in a time-effective manner. Here we proposed an approach that uses the raytracing of the tile with a recursion depth 1 and very low resolution (approx. 32x32 pixels). We also intend to move our raytracing implementation from the cluster environment to a graphic card with GPGPU functionality.

ACKNOWLEDGMENTS

This work is supported by VEGA grant project No. 1/0646/09: “Tasks solution for large graphical data processing in the environment of parallel, distributed and network computer systems”

REFERENCES

[1] CPN tools homepage, *http://wiki.daimi.au.dk/cpntools*.

[2] A. Dietrich, E. Gobbetti, S.-E. Yoon, *"Massive-Model Rendering Techniques: A Tutorial"*, IEEE Computer Graphics and Applications, vol. 27, no. 6, pp. 20-34, 2007.

[3] I. Georgiev, P. Slusallek, RTfact: *"Generic Concepts for Flexible and High Performance Ray Tracing"*, Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing 2008, Los Angeles, USA, pp. 115-122, 2008.

[4] A. Heirich, J. Arvo, *"A competitive analysis of load balancing strategies for parallel ray tracing"*, The Journal of Supercomputing, vol. 12, no. 1-2, pp. 57-68, 1998.

[5] Š. Hudák, Š. Korečko, S. Šimoňák, *"A Support Tool for the Reachability and Other Petri Nets-Related Problems and Formal Design and Analysis of Discrete Systems"*, Problems in Programming, vol. 20, no. 2-3, pp. 613-621, 2008.

[6] H. Indzhov, D. Blagoev, G. Totkov, “*Executable Petri Nets: Towards Modelling and Management of e-Learning Processes*”, International Conference on Computer Systems and Technologies - CompSysTech’09, Ruse, Bulgaria, 2009.

[7] K. Jensen, L.M. Kristensen, L. Wells, *"Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems"*, International Journal on Software Tools for Technology Transfer, vol. 9, no. 3-4 pp. 213-254, 2007.

[8] Š. Korečko, B. Sobota, *"Using coloured petri nets for design of parallel raytracing environment"*, Acta Universitatis Sapientiae Informatica, Romania, 2010, in press.

[9] O. Látka, B. Madoš, J. Perháč, A. Kleinová: *Parallel system module for prepare photorealistic rendering in grid and cluster*, 6th International Symposium on Applied Machine Intelligence and Informatics SAMI 2008, january 21-22, 2008, Herľany, Slovakia. - Budapest : Budapest Tech, 2008, pp. 317-320, ISBN 978-1-4244-2106-0.

[10] I. Notkin, C. Gotsman, *"Parallel Progressive Ray-Tracing"*, Computer Graphics Forum, vol. 16, no. 1, pp. 43-55, 1997.

[11] M. Paralič, M. Krokavec, M. Tomášek: *Perspective Methods and Tools for the Design of Distributed Software Systems*, Computer Science and Technology Research Survey, Košice, Elfa s.r.o., 2007, 1, pp. 16-19, 978-80-8086-046-2

[12] B. Sobota, J. Perháč, Cs. Szabó, Š. Schrötter: *High-resolution visualisation in cluster environment*, Grid Computing for Complex Problem – GCCP 2008, Bratislava, 27.10.-29.10.2008, Bratislava, Slovakia,, Institute of Informatics of Slovak Academy of Sciences (SAS), 2008, pp. 62 - 69, ISBN 978-80-969202-9-7