Some Practical Aspects of Ray Tracing on Shared Memory Machine

Vlastimil Havran, Ji ra
CTU, Fac. of Electrical Eng., Dept. of Computer Science
Karlovo nm. 13, 121 35 Praha 2, Czech Republic
havran@sgi.felk.cvut.cz
zara@cs.felk.cvut.cz

Abstract
This paper compares results of selected load balancing techniques for the ray tracing algorithm implemented on a shared memory machine with eight processors. The aim is to show how theoretically expected results meet reality in practice when a lot of users cause heavy load of computing system. Special attention is paid to the question of load balancing and coherence utilization. A number of measurements were performed both on Eric Haines's PDF test scenes and on ordinary scenes obtained from practical applications.

Keywords
computer graphics, rendering, ray tracing, shared memory, load balancing, coherence, parallel implementation.

1 Introduction

Parallel implementation of sequential algorithms not only brings changes into the structure of programs, but also forces programmers to move their interest toward specific hardware platform. Already known "best sequential" algorithms have to be sometimes changed so that their efficiency is reduced by additional overhead needed for synchronization and communication among processes.

Our research team has been interested in parallelisation of the ray tracing algorithm for a several years. We started in 1993, with the implementation on a massively parallel system of T800 Inmos transputers. Later, the implementation was moved to loosely coupled network of workstations where the PVM system was used as a basic environment for programming of the parallel ray tracer. Due to the connection with other resources and research groups, the parallelisation was based on the space (scene) subdivision ([7]). Such an approach is a memory saving but on the other hand, it is coupled with a large communications overhead. A sufficient efficiency can be thus achieved only on a limited number of computers in a network. We have left the "scene subdivision" approach, although some interesting results on dynamic load balancing were obtained.

Currently, our efforts are oriented toward utilization of shared memory machines. Parallel ray tracer is based on "image subdivision". This principle is very well suited to shared memory machines, when the whole scene is stored in main memory only once. Parts of image are rendered by processors independently. Theoretically, the speed up is linear, e.g. the efficiency is slightly below 1. However, the real efficiency depends on several factors - a global load of shared memory multiprocessor machine, utilization of coherence, number of processes and the distribution of work among them.

2 Hardware platform

We have used the AlphaServer 8400 System made by Digital. It is a 64-bit RISC machine with symmetric multiprocessing (SMP) extendibility. The system is equipped with eight 64-bit processors Alpha 21164 /300 MHz, 2 GBytes RAM memory, 4 GBytes swap space. The internal bandwidth of the data bus is 1.2 GB/sec and the processors are provided with a three-level cache. The internal cache on the chip is 8 KB, write back secondary cache is 96 KB and tertiary on board cache is 4 MB. Typical benchmarks for one processor are 341.4 SPECint92 and 512.9 SPECfp92 (7.43 SPECint95 and 12.4 SPECfp95).

From the programmer's point of view, the SHM library (based on IPC library from AT&T) was used for the implementation of parallel ray tracer.

3 Implemented techniques

The sequential ray tracer already developed by our team was modified for shared memory parallel implementation. This ray tracer was also used as a reference for measuring the speed up and other characteristics of parallel solutions.

Although a very simple and well known approach called "process farming" was used for parallelisation, there was still a number of possibilities, how to subdivide an image and how to distribute the work among processes. To achieve the best efficiency, one has to pay attention both to load balancing and good utilization of coherence. The following notes are dedicated to these problems:

Load balancing

To fully utilize all processors available, the work load has to be distributed among them for the whole computing time. Unfortunately, the characteristics of the ray tracing algorithm doesn't allow to determine the complexity or the cost of computation for certain parts of an image. The amount of computations for each subimage depends on geometrical and optical features of objects hit by rays, which can be reflected and refracted into arbitrary directions. If the computation requirements are practically unforeseeable, the load balancing can be solved either dynamically (the work is distributed on demand) or statically by image subdivision with very fine granularity (all processes shall obtain almost the same amount of simple tasks, thus the loads shall be comparable from the point of view of statistics).

The dynamic solution brings some communication overhead, which is needed for the distribution of work. Also the coherence utilization is not high, because processes have to work with new data directly obtained from the main control process, so they cannot use already processed scene data (geometry, textures) which could be currently available in a cache.

The static load balancing based on the image subdivision into very small areas (rectangular blocks or rows of pixels) ensures almost regular load for all processes. The problem is, that high task granularity (e.g. processing one pixel as a one task) decreases effect of data coherence.

Coherence utilization

There are a number of methods in the ray tracing literature, which were designed with the aim to fully exploit some kind of coherence. We have not used any special method like ray coherency (for antialiasing within one pixel), light buffers (for shadow rays), ray beams or safety zones (to minimize intersection computations).

However, a *data coherence* was taken into account. Since each process has to operate (theoretically) with the whole scene consisting of thousands of objects, there

must be a time difference between processing data taken from the fast but small **cache** and the large but slower **main memory**. Although the data transfer between cache and main memory is up to a processor, a kind of software optimization can be also achieved by selection of primary rays to be computed. There is a high probability that adjacent primary rays will hit the same objects ([1]), thus the optimal area for rendering has a squared shape ([8]).

Notice, that this kind of data coherence has the larger impact to speed up in case of large number of simple objects to be tested for intersection. If a geometry of an object is complex (e.g. spline surfaces), a processor spends all time with arithmetic instructions on small set of geometrical data. A difference between cache and main memory is then hidden and a data coherence has very low influence. That is the reason why our testing scenes have been built mostly from spheres and triangle patches, which are the simplest shapes for ray tracing.

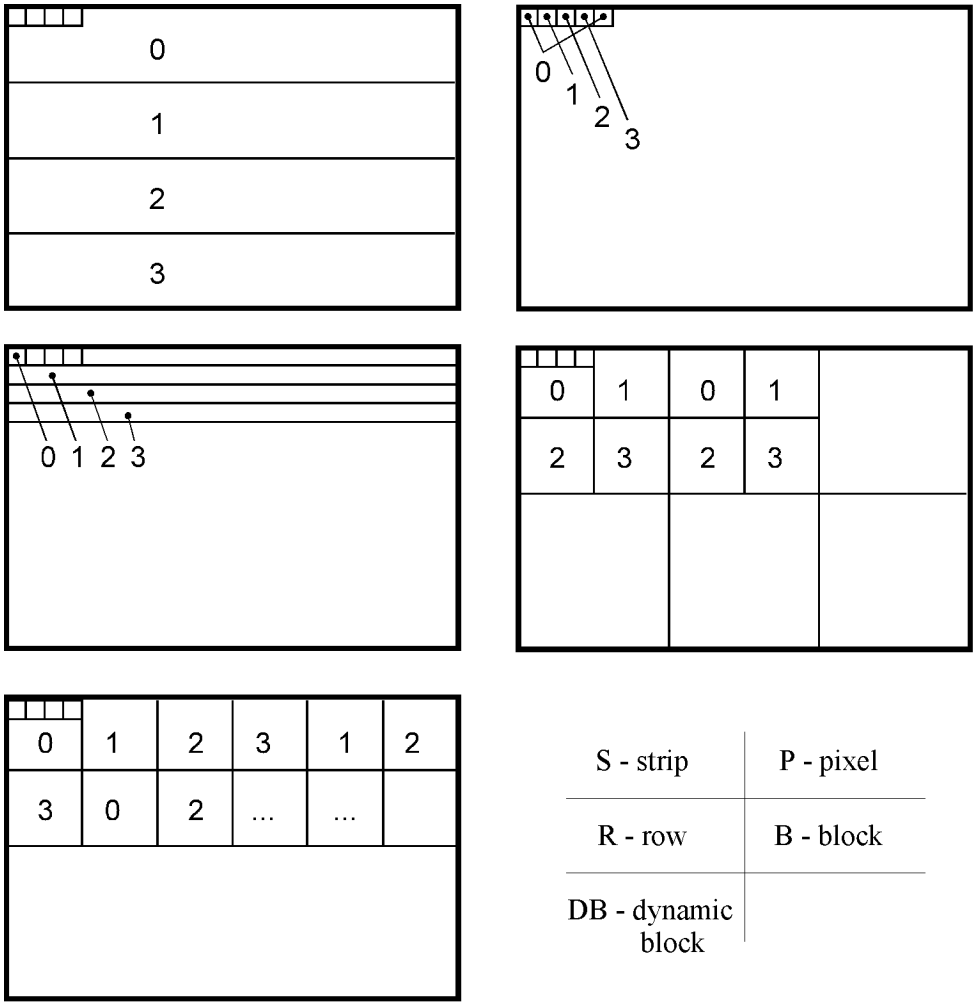

Figure 1: One dynamic and four static load balancing techniques

3.1 Process mapping

Five methods for image subdivision were tested. The principles are shown on fig. 1, where only four processes (0, 1, 2 and 3) are used for simplification:

(S) Strip subdivision – the number of image parts is equal to a number of processes. The best coherence is achieved, but the load balancing is the worst.

(P) Pixel subdivision – the finest granularity gives the best static load balancing, but doesn't use any advantage of coherence.

(R) Row subdivision – the compromising solution between static load balancing and coherence. The number of rows should be much higher than the number of processes.

(B) Block subdivision – an image is subdivided into the fields, which are further divided into small blocks. The number of blocks within one field is equal to the number of processes. Relatively small area of fields ensures approximately the same scene complexity for all blocks within one field.

(DB) Dynamic block distribution – similar small blocks as in the previous method are sequentially distributed among processes on demand. A synchronization and communication during rendering is required. A coherence is exploited within one block, but can be also extended to the adjacent blocks in some cases as shown on fig. 1 (process No. 2).

Only the last technique provides dynamic load balancing. The others are based on initial static subdivision and completely independent parallel computations.

4 Measurements

Our aim was to measure rendering times depending on image subdivision methods (**S, P, R, B, DB**) as well as on the number of processes.

Let us describe the values that were in our interest:

Real time is the time, which is important for the end user. It is a time interval between the beginning and the end of the whole computational process, regardless of the number of processors involved. **Real** time depends on the load of the system, for instance on the number of users in the moment of the rendering.

User time is bundled with a rendering process. It expresses a time, when the process is active, i.e. when a processor performs the code of the process. A **user** time doesn't depend on the load of machine.

In case of parallel computation, the set of **user** times is obtained (one for each process). The better the load balancing is, the smaller the differences among user times.

To compare efficiency of parallel implementation in the following text, we have selected the worst **user** time from each set of user times from one experiment.

Speed up is done as a fraction T_1/T_n, where T_1 is sequential time (time for rendering by one process only) and T_n is the time achieved by n rendering processes. Optimal **speed up** is equal to n.

Speed up can be expressed both in terms of **real** and **user** times.

4.1 Scene characteristics

There have been five scenes used for testing. Three of them are standard PDF ([2]) ray tracing benchmark data (*Balls, Tetra* and *Shell*), next two are taken from user

applications. The scene *Room* comes from CAAD system, *Fluid* is the result of visualization of a fluid dynamics simulation.

Name	Geometry	# of elems.	# of light sources	# of inters. tests [10^6]	depth of recursion
Balls	spheres	7382	3	193.0	2
Tetra	triangles	4096	1	7.6	2
Shell	spheres	5761	1	212.2	2
Room	polyg. meshes	7786	3	114.4	4
Fluid	spheres	2899	2	29.3	5

All scenes were rendered with resolution 512×512 pixels, one ray per pixel. The binary space subdivision (BSP) technique was used for decreasing a number of intersection tests.

The following table shows real and user times for rendering scenes using one process only. All five methods for process mapping were used, although some of them have no impact in case of sequential processing. Strip, pixel and row subdivision are here the same, because the image area is rendered by one process. Differences can be seen on Block and Dynamic block approaches. Block subdivision uses image fields 64×64 pixels, dynamic blocks are even smaller – 32×32 pixels.

method	time [s]	*Balls*	*Tetra*	*Shell*	*Room*	*Fluid*
Strip	real	345.766	28.815	403.032	365.654	97.499
	user	345.531	28.775	402.795	365.364	97.418
Pixel	real	345.668	29.006	414.852	364.516	98.087
	user	345.338	28.967	414.555	364.222	97.999
Row	real	345.114	28.830	419.856	366.137	97.759
	user	344.862	28.793	419.557	365.772	97.676
Block	real	334.902	28.524	371.218	360.178	96.951
	user	334.661	28.474	370.949	359.875	96.853
Dynamic	real	345.487	28.789	370.725	364.824	97.839
block	user	345.274	28.753	370.464	364.522	97.756

As expected, the corresponding **S**, **P** and **R** subdivision computational times for one scene were almost equal. The best sequential times were achieved for the method **B**, when data coherence within one small image block helps to decrease frequent access to main memory. **DB** subdivision was a little bit slower due to the communication between main process and one rendering process (no "take ahead" technique was used).

The times from the table above were taken as references values for expression of appropriate speed up of parallel implementation. It means that using method **S** for load balancing, results were compared with the **S real** and **S user** times from this table; using method **P** in parallel computation, the results were compared with the **P real** and **P user** times from the table, etc.

4.2 Results

The following graphs (fig. 2 and 3) show the times depending on the number of processes performed in parallel. Although the computing system has been equipped with eight processors, we have tried to measure more than eight processes in parallel implementation. That is why the graphs express values up to 16 processes.

Let us remind, that in the case of **dynamic block** subdivision, the basic image subarea distributed among processes was 32×32 pixels. In case of **block** subdivision, the squared field was 64×64 pixels and it was further subdivided depending on the number of processes. For the latter we had to developed a special incremental algorithm, which is able to cover a field with n areas for n processes. Each area fulfilled two criteria – its shape was as much squared as possible and the number of pixels was almost equal for all areas within a field.

4.3 Discussion

Real time **speed up** depends on the overall load of the computing system. We have done measurements during many sessions, i.e. the overall load was not constant. The number of users has been alternated, therefore only the shapes of graphs are important, not absolute values.

Notice, that the real speed up continuously grows with the number of processes, although the number of processors are constant - eight. The reason is that the operating system gives various priorities to processes and doesn't take into account any kind of a "global" priority of a user. A newer process has always a higher priority, which is decreased with the living time of a process. Relatively short processes with ray tracing algorithm thus have a higher priority comparing with special applications (chemistry, biology, etc.) computed tens of hours or even longer.

In practice, a user should use the number of processes up to the number of processors, but not more. Higher number of processes takes time, which should be given to other users. Of course, this is also a question of operating system, i.e. rights of users in a multiuser and multitasking computational systems at all.

The graphs with **user** time **speed up** (fig. 3) allow to compare all five implemented load balancing techniques regardless of the number of users. Methods **R** and **P** give the best speed up comparing with **B** and **DB** approaches. In these experiments, the load balancing had the main influence to the overall efficiency. The data coherence has not played so important role. The memory architecture of AlphaServer 8400 with three levels of caches successfully hides differences between "near" and "far" data location.

The results obtained for more than 8 processes are interested mostly from the theoretical point of view. Since the granularity of image subdivision is fine enough (pixels, rows), we can expect almost linear user time speed up in case of the higher number of processors.

An interesting phenomenon can be seen on fig. 3 for the *Shell* scene. When 16 processes were started, the **user** times went down more than 16 times. Of course, the global **real** time stayed below real time for 8 processes. The "exciting" user time speed up better than linear has simple explanation – two processes were mapped onto one processor. Both processes rendered couples of adjacent rows, so data coherence caused better results. This special case cannot occur when real 16 processors are used. This is also an example how the results of the simulation can be imprecise due to the cache system.

5 Conclusion

There has been set of measurements done on shared memory machine with eight processors. The results obtained from five different scenes are comparable.

The load balancing has had bigger influence to the computational time than utilization of data coherence. Pixel (**P**) and Row (**R**) image subdivision approaches are quite satisfactory from the point of view of efficiency and overall speed up. However, the other two methods – block (**B**) and dynamic block (**DB**) subdivision

have given also comparable results. The worst approach was to subdivide an image into strips.

Another observation was done in terms of number of processes versus number of processors. Although the speed up cannot overcome the limit done by the number of processors, a real time can be still decreased using more processes than processors. The extendibility of this solution is not high, but the curve of efficiency continuously grows at least up to twice processes more than processors. Of course, this portion of speed up is paid by other users working on the system.

6 Future work

We are going to pay further attention to the processing of textures. In case of large 2D images used as mapped textures, data coherence plays important role. The adjacent points on a 3D surface are often mapped onto points in texture space, which are in different memory locations. In such cases, the data coherence is achieved for ray traced objects only, but not for the texture data. Our aim is to test the hierarchical image description and its influence to a speed up of rendering in a parallel environment.

Acknowledgment

We would like to thank to Jan Buriánek who has been the programmer of the sequential ray tracer and who has helped us to adapt the software into parallel environment, and to Ale Holeek who spent his time with discussions on load balancing techniques.

References

[1] Green, S.: *Parallel Processing for Computer Graphics.* Research Monographs in Parallel and Distributed Computing, Pitman Publishing, London 1991.

[2] Haines, E.: *A Proposal for Standard Graphics Environments.* IEEE Comp. Graph. & Apps., Vol. 7 (11), pp. 3–5, 1987.

[3] Havran, V.:*Simulation of Real Camera for Rendering.* Master Thesis, Faculty of Electroengineering, CTU Prague, 1996.

[4] Horiguchi, S., Masayuki, K.:*Parallel Processing of Incremental Ray Tracing on a Shared Memory Multiprocessor.* The Visual Computer, Volume 9, pp. 371–380, Springer-Verlag, 1993.

[5] Jansen, F., Chalmers, A.:*Realism in Real Time?* Invited presentation on 4th EG Workshop on Rendering, Paris, pp. 27–46, June 1993.

[6] Keates M.J, Hubbold R.J.:*Interactive Ray Tracing on a Virtual Shared-Memory Parallel Computer.* Computer Graphics Forum, Volume 14, number 4, pp. 189–202, 1995.

[7] Menzel K. et all.: *Distributed Rendering Techniques using Virtual Walls.* Proceedings of First European PVM Users Group Meeting, Roma, Italy, October 9-11, 1994.

[8] Voorhies, D.: *Space-Filling Curves and a Measure of Coherence.* in Graphics Gems II, Academic Press, UK, pp. 26–30, 1991.

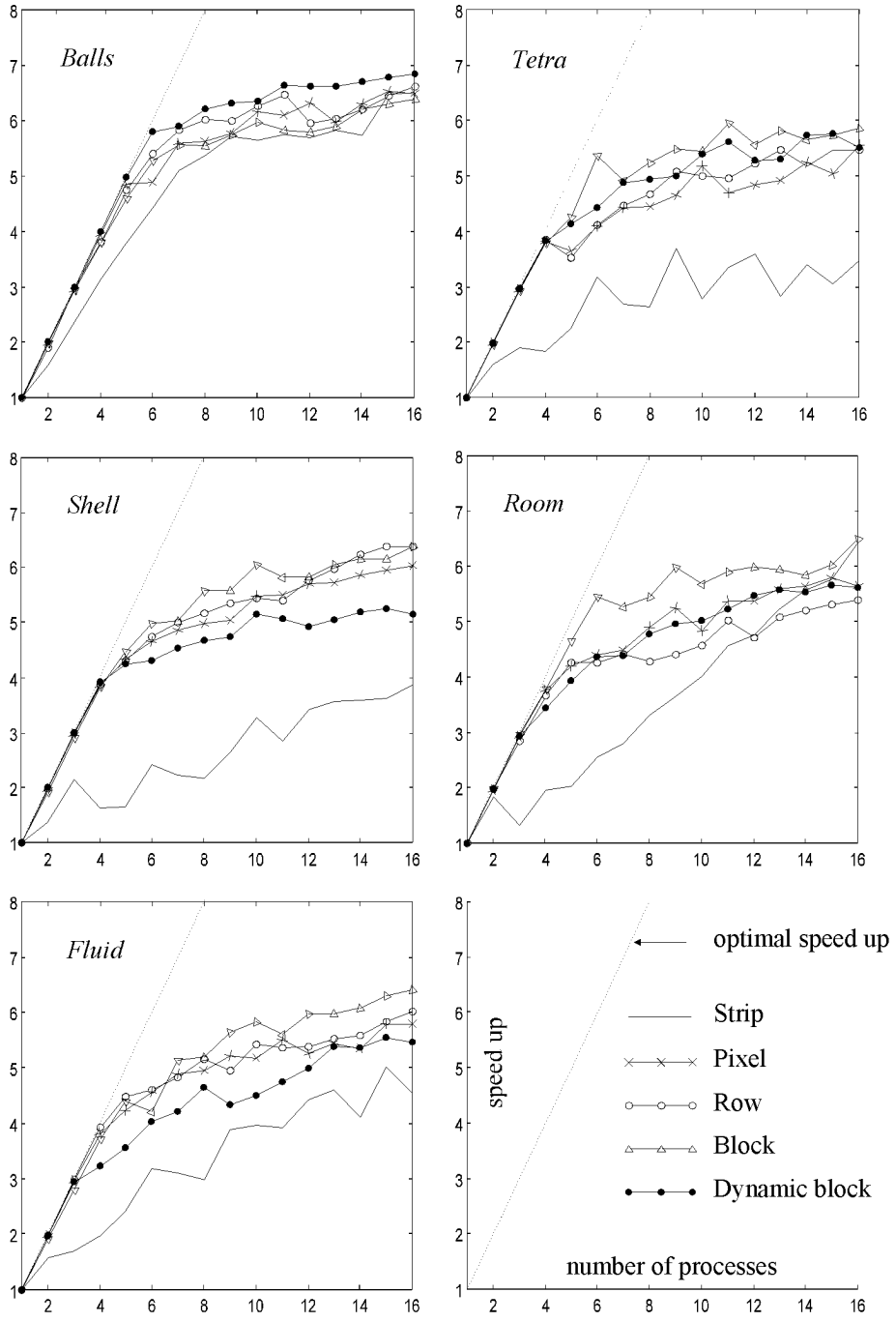

Figure 2: Speed up based on the **real** times

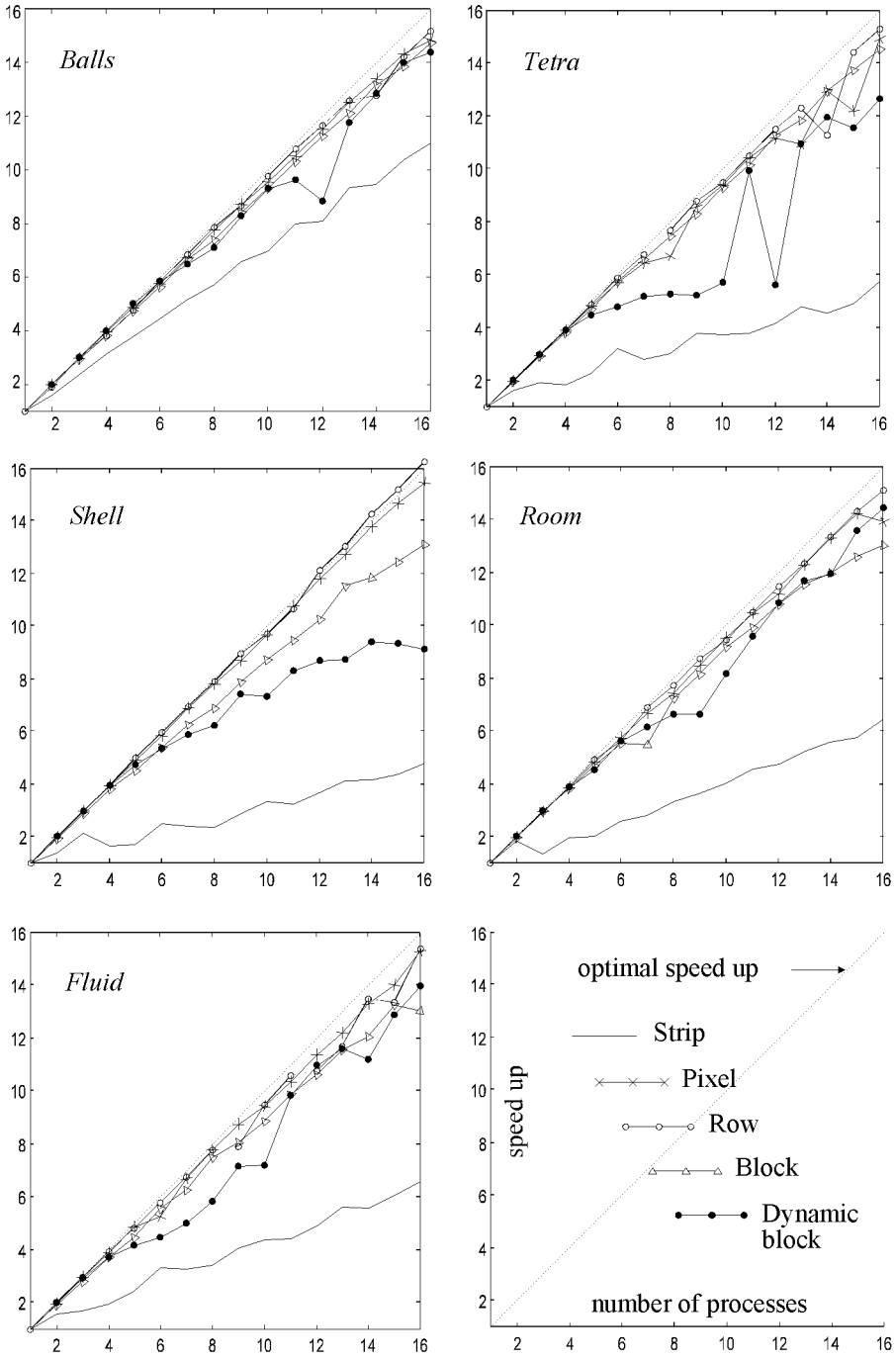

Figure 3: Speed up based on the **user** times