A Parallel Ray Tracing Application – The Base Engine

M. D. ZAHARIA*

Key words: realistic image synthesis, parallel computing, synchronization

Articolul descrie caracteristicile sistemului Y.A.R.T.E., o aplicaţie paralelã din domeniul sintezei realiste de imagini. Aplicaţia a fost dezvoltatã pe sistemul Sun Enterprise 10000 un sistem HPC achiziţionat în cadrul proiectului CoLaborator la Universitatea Politehnica Bucureşti. Lucrarea prezintã tehnicile de implementare utilizate precum şi evaluarea performanţelor în contextul de execuţie oferit de sistemul de calcul menţionat mai sus.

The paper describes the main features of the Y.A.R.T.E. system. Y.A.R.T.E. is a parallel ray-tracing application that was developed on Sun Enterprise 10000 high performance computing system. This hardware platform was acquired as a part of CoLaborator project, developed at the Computer Science Department of POLITEHNICA University of Bucharest. The main implementation techniques as well as the performance evaluation in the above-mentioned computing environment are also presented.

Introduction

At the Computer Science and Engineering Department at University POLITEHNICA of Bucharest, in the framework of the project CoLaborator it was acquired a Sun Enterprise 10000 parallel computer [7]. The system has a shared memory architecture (SMP); the actual configuration has 32 processors

* Conf. dr. ing. Dept. of Computer Science, University "Politehnica" of Bucharest, ROMANIA

(Ultra SPARC II at 400 MHz) and 16 GB RAM memory. This paper presents one of the possible categories of applications suitable to efficiently use this powerful computing environment; it refers to the realistic image synthesis of 3D scenes.

This type of computer graphics applications tries to compute as exactly as possible the distribution of the light energy in a given environment. The calculus takes into consideration the geometrical shapes of the objects, material and optical properties of object surfaces, the geometry of the camera lenses, the relative positions of the objects and light sources, the characteristics of the radiation emitted by the light sources.

The light distribution in a given environment could be computed by integrating the rendering equation of Kajiya ([1], [2], [3]); that is a Fredholm integral equation of the second kind.

The rendering equation describes quantitatively the energy transfer from a point (P) placed on a surface of a scene object to another point (P'):

$$I(P,P') = g(P,P')(e(P,P') + \int_S r(P,P',P'') * t(P,P',P'')dP'')$$

where:

- I(P,P') is the intensity of light radiation transmitted from P to P'
- g(P,P') is a factor depending on the scene geometry, given by:

$$g(P,P') = \begin{cases} 0, & if\_P,P'\_are\_mutually\_invisible \\ \dfrac{1}{dist(P,P')}, & if\_P\_is\_visible\_from\_P' \end{cases}$$

- e(P,P') is the intensity of the light radiation emitted from P' to P
- the integral is defined over all the surfaces S of the scene objects
- r(P, P', P'') is the intensity of the specularly or diffusely reflected radiation from P'' to P and passing through P'

The rendering equation is deduced from the general light transport equation eliminating the terms for polarisation, phosphorescence and fluorescence.

One of the major approaches for solving the rendering equation is ray tracing. This method tries to solve this equation for locations (placed on object surfaces) and light directions determined by the view specification. Only the surfaces of the objects that are visible to the viewer and only the directions that lead back to the camera position are taken into consideration. This type of

solution is view dependent, works in image space and limits the number of points and directions where the lighting calculus is made.

The other type of solution (named radiosity method) is view independent and computes an approximation of the radiance function in the form of radiance values associated to different sampling locations in the object environment. The camera position does not limit the solution scope.

The system YARTE is a ray tracing based image synthesis system developed at the Computer Science and Engineering Department of University "Politehnica" of Bucharest; the name YARTE. is the acronym for "Yet Another Ray Tracing Engine". The system was initially developed in order to help practicing the principles of the realistic image synthesis algorithms; it was redesigned in order to exploit the inherent image space parallelism of the ray-tracing method and ported on Sun Enterprise 10000 computing system. Finally, it will evolve towards an efficient hybrid (radiosity/ray-tracing) image synthesis system.

The present paper describes the main facilities of the YARTE system, the architecture of the system, the implementation techniques of the main modules, the way the parallelism of the application was achieved and the global performance evaluation.

1. System architecture

YARTE implementation uses the visibility tracing approach[1]. One ray starts from the viewing point, passes through the center of an arbitrary pixel (P) of the view plane and reaches the nearest object surface in point (I) (see Fig. 1). The local color of (I) is determined by the Shading() routine, using the Phong local reflection model and taking into consideration every light source of the scene. The algorithm is then recursively applied considering the reflected and the transmitted rays starting from (I). This allows the final determination of point (P) color, taking into consideration the light interaction between the objects of the 3D scene. In the actual implementation the functions IntersectGlobal() (that computes the ray-object intersection placed nearest the ray starting point) and Shading() (that determines the object color in the intersection point) are mutually recursive. The maximum recursivity level is an input parameter of the program.

[1] Other implementation approaches of the ray-tracing method are photon tracing and bidirectional ray-tracing

The YARTE system was written in C++. The main classes and their derivation relation are specified in the figure below.

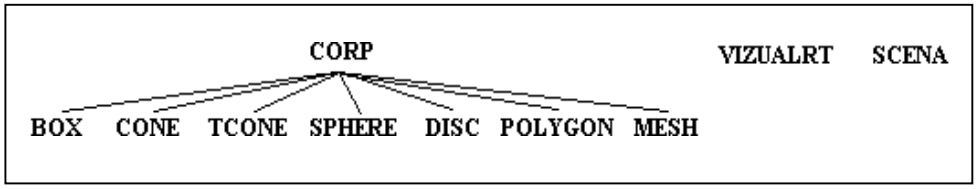

Fig. 1 – Main application classes

Each class associated to a primitive geometrical object is derived from class CORP. The definition of the class CORP is:

```
class CORP {
  private:
    int corpID; // corp ID
  protected:
    int isinstance; // has an assoc. instance transformation
    int isfrontback; // two faces with different materials
  public:
    V3D refx, refy, refz;
    // versors of the definition referential
    MATERIAL supr; // material properties (front face)
    MATERIAL backsupr; // material properties (back face)
    void setCorpID(int cid);
    int getCorpID(void);
    int isFrontBack(void);
    void setFrontBack(int v);
    void setInstanceReferential(V3D& vx, V3D& vy, V3D& vz);
    // set the versors of the associated instance referential
    void setSurfaceMaterial(CULOARE& a, CULOARE& d, CULOARE& s,
                                    double cPh, double cr, double ct,
                    double ind_tr);
    void setBackSurfaceMaterial(CULOARE& a, CULOARE& d,
        CULOARE& s, double cPh, double cr, double ct,
        double ind_tr);
    // a = ambient color d = diffuse color
    // s = specularly reflected color
    // cPh = coefficient of specular reflection (Phong)
    // cr = reflection coefficient of the surface
    // ct = transmission coefficient of the surface
    CULOARE Shading(P3D& p, V3D& r, V3D& n);
```

```
  // p = the point placed on object surface where the color
  // is computed
  // r = versor of the incident ray
  // n = surface normal versor in p
  virtual double Intersectie(P3D& p, V3D& dir) = 0;
  // returns the parametric coordinate of the intersection
  // between the radius starting from p with direction dir
  // and the object surface
  virtual V3D Normala(P3D& p) = 0 ;
  // returns the normal surface versor in point p (placed on
  // the current object surface)
};
```

The virtual methods Intersectie() and Normala() are effectively implemented in the derived classes (BOX, CONE etc.) and accomplish geometry dependent operations. As an example, the intersection between a cone boundary and a ray specified by its origin and its direction versor is computed by the following sequence:

```
double CONE::Intersectie(P3D& p1, V3D& diraux)
{ double a, b, c, delta, xint, yint, zint;
 double aux, s, smin=INFINIT;
 P3D p; V3D dir=diraux;
 if(isinstance) {
  // the object has an associated instance transformation
  P3D p2;
  p2.x=p1.x-centru.x; p2.y=p1.y-centru.y; p2.z=p1.z-centru.z;
  p=SchimbTriedru(refx, refy, refz, p2);
  dir=SchimbTriedru(refx, refy, refz, diraux);
 }
 else {
  p.x=p1.x-centru.x; p.y=p1.y-centru.y; p.z=p1.z-centru.z;
 }
 aux=h*h/(raza*raza);
 a=dir.x*dir.x*aux+dir.z*dir.z*aux-dir.y*dir.y;
 b=(dir.x*p.x+dir.z*p.z)*aux+dir.y*(h-p.y);
 c=(p.x*p.x+p.z*p.z)*aux-(h-p.y)*(h-p.y);
 if(fabs(a)<=TOLERANTA) {
  if(fabs(b)>TOLERANTA) {
   s=-c/b;
   if(s>0.0 && s<smin) smin=EValid(s, p, dir) ? s:smin;
  }
 }
 else {
```

```
   delta=b*b-a*c;
   if(delta>=0.0) {
     delta=sqrt(delta);
     s=(-b-delta)/a;
     if(s>0.0 && s<smin) smin=EValid(s, p, dir) ? s:smin;
     s=(-b+delta)/a;
     if(s>0.0 && s<smin) smin=EValid(s, p, dir) ? s:smin;
   }
  }
// intersection with the base disc of the cone
  if(fabs(dir.y)>=TOLERANTA) {
    s=-p.y/dir.y;
    xint=p.x+s*dir.x;
    zint=p.z+s*dir.z;
    if((xint*xint+zint*zint)<=raza*raza) {
      if(s>0.0 && s<smin) smin=s;
    }
  }
  return (smin==INFINIT) ? 0.0 : smin;
}
```

The normal at the cone surface in point p1 is returned by the function:

```
V3D CONE::Normala(P3D& p1)
{ V3D v1, v2, v3;
  P3D p=p1;
  double r;
  if(isinstance) {p=SchimbTriedru(refx, refy, refz, p1);}
  if(fabs(p.y)<=TOLERANTA) {
    if(fabs(p.x*p.x+p.z*p.z-raza*raza)<=TOLERANTA) {
      v2.x=0.0; v2.y=h; v2.z=0.0;
      v3=Versor(v2, p);
    }
    else {v3.x=0.0; v3.y=-1.0; v3.z=0.0; }
  }
  else {
    if(fabs(p.x)<=TOLERANTA && fabs(p.y)<=TOLERANTA &&
       fabs(h-p.y)<=TOLERANTA && (h-p.y)>=0.0) {
      v3.x=0.0; v3.y=1.0; v3.z=0.0;
    }
    else {
      v2.x=0.0; v2.y=h; v2.z=0.0;
      v1=Versor(p, v2);
      v2.x=p.z; v2.y=0.0; v2.z=-p.x;
      Normalizare(v2);
      v3=ProdusVectorial(v1, v2);
```

```
    Normalizare(v3);
   }
  }
  if(isinstance) {
   v3=SchimbTriedruInvers(refx, refy, refz, v3);
   Normalizare(v3);
  }
  return v3;
}
```

The other methods of the class CORP establish a behavior independent of the corresponding object geometry. For example some objects could support an instance transformation. A typical example is the BOX object. It corresponds to a rectangular orthogonal prism whose sizes are parallel to the referential axes. The BOX center could have an arbitrary position and the sizes could have arbitrary lengths. By default the box is considered placed in the origin (the definition referential) and it is possible to subsequently apply a translation that moves the center of the box in an arbitrary spatial location. The user could associate an instance referential to the BOX object. This referential is specified by the three versors of its axes (if the third versor is not specified it is considered equal to the vector product of the other two versors). If an instance referential is associated to a geometrical object then the sequence of rotations that transforms the definition referential in the instance referential will be applied to the object followed by the translation of the object center in an arbitrary spatial location.

The specification of an instance referential increases considerably the costs of ray-object intersection computations but allows rendering arbitrarily oriented objects.

The geometric primitives supported by the YARTE system are: BOX, CONE, TCONE, SPHERE, POLYGON, MESH, DISC.

- BOX designates a rectangular orthogonal prism having the faces placed in planes that are parallel with the coordinate planes of the scene referential. The geometry is specified by the coordinates of the box center and the three size lengths. The box can be instantiated.
- CONE designates a circular orthogonal cone, including the disc from the base plane. The cone geometry is given by the coordinates of the base center and the height. By default, the cone is considered to have the base in a plane parallel to xOz and the height direction of versor **j**. The cone can be instantiated.

- TCONE designates a truncated cone surface. The two circular discs (lids) are not part of the truncated cone. It is possible that the user specifies different material properties for the external respectively internal faces of the truncated cone surface. The geometry is specified through: the radius of the large base disc, the truncated cone height, the coordinates of the center of the disc base and the height of the cone that includes the truncated cone surface. The TCONE primitive could be instantiated.
- SPHERE the sphere geometry is given by the center coordinates and the radius.
- DISC designates a circular disc with a given center and radius. The disc is considered placed in a plane with normal versor **j**. The disc can be instantiated and its faces could have different material properties.
- POLYGON designates a quasi-planar polygon specified by the list of its vertices. The user must be sure that the vertices are (quasi) coplanar. The normal versor is computed using the Newell rule ([1]) and its orientation depends on the enumeration order of the vertices. The polygon can not be instantiated.
- MESH this class corresponds to a polygonal mesh surface. The geometry is specified in an ASCII text file with the following syntax:

  number_of_vertices number_of_faces

  coordinates of the vertices (one vertex – 3 coordinates – per line)

  indexes of the vertices associated to each polygonal face of the mesh. Each line corresponds to one face and contains the indices of the vertices adjacent to the face. The enumeration order of the vertices corresponds to the external normal rule.

  The class constructor interprets the .pmf (polygonal mesh) file and generates a number of POLYGON objects. Some polygonal mesh data are available at [5]. An object instance of the class mesh could have associated one scaling, three rotations and one translation transformation. They are applied to all vertices before the generation of the POLYGON objects.

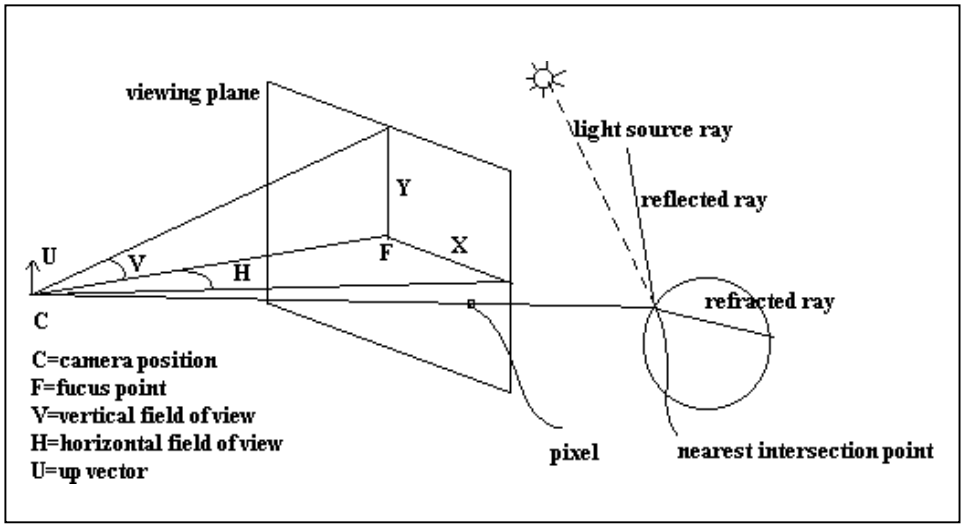

Fig. 2 – Ray tracing viewing geometry

The class VIZUALRT computes the parameters of the viewing transform. These parameters are: camera position, focus point (the origin of the viewing plane referential), the up versor, the image resolution (i.e. the number of sampling points on each axe of the viewing plane referential) and the camera horizontal and vertical fields of view.

Finally, the class SCENA contains the ray tracing engine, The class constructor interprets the scene description language, initializes the data structures corresponding to the scene objects, light sources and camera parameters and starts running the ray tracing engine. The description of a 3D scene in the form recognized by YARTE is presented below:

```
image_resolution 256 256
camera_position 0.0 0.0 120.0
camera_apertures 45.0 45.0 ; specified in degrees
focus_point 0.0 0.0 0.0
up_versor 0.0 1.0 0.0
light_source 120.0 120.0 120.0 0.8 ; position and brightness
light_source -120.0 150.0 -60.0 0.8
background_colour 0.0 0.0 0.0
max_recurse_level 2
gamma_correction_factor 2.20

material 0.2 0.2 0.2  0.8 0.0 0.0  0.2 0.2 0.2  100.0 0.8 0.0 1.0
```

```
; material properties are: the ambient, diffuse and specular color,
; Phong coefficient, reflection and transmission coefficients
; and the specular refraction index
sphere -60.0 -30.0 20.0 40.0

material 0.0 0.2 0.0  0.0 0.0 0.8  0.2 0.2 0.2  100.0 0.8 0.0 0.0
polygon 4  -200. -100. 200.  200. -100. 200.  200. -100. -500.  -200. -100. -500.

polygon 4  -200. 200. 200.  200. 200. 200.  200. 200. -500.  -200. 200. -500.
backface_colour 0.0 0.2 0.0  0.0 0.8 0.0  0.2 0.2 0.2

material 0.2 0.0 0.0  0.8 0.8 0.0  0.2 0.2 0.2  100.0 0.3 0.0 0.0
truncated_cone -65.0 50.0 20.0 40.0 80.0 30.0
backface_colour 0.0 0.0 0.2 0.0 0.0 0.8 0.2 0.2 0.2
instance_referential 1.0 0.0 0.0  0.0 0.0 1.0  0.0 -1.0 0.0

material 0.0 0.0 0.1 0.0 0.0 0.9 0.3 0.3 0.3 100.0 0.8 0.0 0.0
cone 0.0 60.0 0.0 30.0 60.0
instance_referential 1.0 0.0 0.0  0.0 -1.0 0.0  0.0 0.0 -1.0

material 0.4 0.0 0.0 0.8 0.0 0.0 0.2 0.2 0.2 100.0 0.8 0.0 0.0
disc -65.0 50.0 20.0 40.0
backface_colour 0.0 0.0 0.2  0.0 0.0 1.0  0.0 0.0 0.0
instance_referential 1.0 0.0 0.0  0.0 0.0 1.0  0.0 -1.0 0.0

material 0.5 0.2 0.0 0.0 0.0 0.9 0.3 0.3 0.3 100.0 0.7 0.6 1.2
mesh buck.pmf 30.0  1.0 0.0 0.0  0.0 1.0 0.0  0.0 0.0 1.0  65.0 -20.0 -20.0

material 0.2 0.2 0.2 0.0 0.9 0.0 0.2 0.2 0.2 80.0 0.8 0.3 1.2
box 55.0 -60.0 40.0  40.0 10.0 25.0
instance_referential 0.707 0.707 0.0  -0.707 0.707 0.0  0.0 0.0 0.0
```

2. Achieving parallelism

An important method to improve efficiency considering the huge resources request of the realistic image synthesis applications consists in designing parallel or distributed algorithms that should benefit from the intrinsic parallelism of ray tracing and/or radiosity algorithms.

Many research groups have developed hardware architectures particularly suitable to accelerate image synthesis. Representative examples are the WARP architecture (developed at Carnegie Melon University) and the Pixel-Plane architecture of H. Fuchs ([7], [8]).

The ray-tracing algorithm has an inherent – image space - parallelism due to the fact that the computations associated to each ray shot from the camera position could independently be done. In order to emphasize the parallelism of the ray-tracing algorithm, the image space can be structured as a set of rectangular regions with vertices placed on a rectangular grid. The regions should be separately treated, by processes that do not intercommunicate. In a UNIX environment the processes that separately render image space regions could be children of the same father. In UNIX such processes have reentrant code and each of them inherits from the father the scene description that will allow building one sub-region of the image. The father process could compile the scene description language, assign regions to its sons and route the resulted pixels to a final visualization server. This type of decomposition is implemented in YARTE system. The actual implementation offers two versions differing by the fact that the independent rendering activities are accomplished by separate threads (of the same process) or by separate processes (having the same father). Below is given a code sample of the parallel ray-tracing engine. Function Rtpartial() is executed by every son rendering process; it accomplishes the gamma correction and writes the corresponding part of the current image line in the output file.

```
struct sembuf P={0, -1, 0}, V={0, 1, 0};
int* GLgammatable;

void SCENA::Raytrace(void)
// RayTracing engine core
{int x, y;
 double t;
 unsigned short usx, usy;
 CULOARE cul;
 rinit=vrt.getRazaInit();
 Normalizare(rinit);
 if(GLgamma!=1.0) consGammaTable();
 unsigned short usxrtp, usyrtp;
 int k, it;
 if((frtrtp=open("/tmp/imag.dmp", O_CREAT|O_SYNC|O_TRUNC|O_WRONLY, 0777))<0)
 {
  fprintf(stderr, "RTpartial -- open file error\n");
  exit(1);
 }
 // in current configuration the /tmp directory is placed on a RAM disk
 sxrtp=sx; syrtp=sy; usxrtp=sx; usyrtp=sy;
 //sx and sy designate the x and y image resolution
```

```
  write(frtrtp, &usxrtp, sizeof(unsigned short));
  write(frtrtp, &usyrtp, sizeof(unsigned short));
  ecxrtp=ecx; ecyrtp=ecy; rinitrtp=rinit;
  if((linie_imagine=(unsigned char *)malloc((sxrtp)*3))==NULL) { // 3 octeti RGB
    fprintf(stderr, "Raytrace -- linie_imagine allocation error\n");
    exit(1);
  }
// synchronization with semaphores
  if((semid=semget(IPC_PRIVATE, 1, IPC_CREAT | 0666))<0) {
    fprintf(stderr, " scena2 -- semget error\n");
    exit(1);
  }
  if(semop(semid, &V, 1)<0) {
    fprintf(stderr, "scena2 -- V1 error\n");
    exit(1);
  }
  for(it=0; it<NTHREADS; it++)
    if(fork()==0) { // every son process treats a different image region
      RTpartial((void *)&it);
      exit(0);
    }
  while(wait(&k)!=-1);
  close(frtrtp);
  semctl(semid, 0, IPC_RMID);
}
#if ((NTHREADS>=2) && (NTHREADS<=NMAXTHREADS))
// Rendering of an image space strip. The image is decomposed
// in NTHREADS rectangular strips that will be separately treated
void* RTpartial(void *arg)
{ int DX, it, tid[NTHREADS];
  int i, x, y;
  double t;
  CULOARE cul;
  V3D razacrt;
  int indice_thr, *sarg;
  sarg=(int *)arg;
  indice_thr=*sarg;
  DX=sxrtp/NTHREADS;
  i=indice_thr;
  for(y=-syrtp/2; y<syrtp/2; y++) {
    for(x=-sxrtp/2+i*DX; x<((i==NTHREADS-1)? (sxrtp/2) : (-sxrtp/2+(i+1)*DX)); x++) {
      razacrt.x=rinitrtp.x+x*ecxrtp.x-y*ecyrtp.x;
      razacrt.y=rinitrtp.y+x*ecxrtp.y-y*ecyrtp.y;
      razacrt.z=rinitrtp.z+x*ecxrtp.z-y*ecyrtp.z;
      Normalizare(razacrt);
      t=Intersectglobal(-1, GLpobs, razacrt, cul);
```

```
    unsigned char color[3];
    if(t>0.0) {
          color[0]=(unsigned char)(cul.R*255);
          color[1]=(unsigned char)(cul.G*255);
          color[2]=(unsigned char)(cul.B*255);
          if(GLgamma==1.0) {
            linie_imagine[(x+sxrtp/2)*3]=color[0];
            linie_imagine[(x+sxrtp/2)*3+1]=color[1];
            linie_imagine[(x+sxrtp/2)*3+2]=color[2];
          }
          else {
            linie_imagine[(x+sxrtp/2)*3]=(unsigned char)GLgammatable[(int)color[0]];
            linie_imagine[(x+sxrtp/2)*3+1]=(unsigned char)GLgammatable[(int)color[1]];
            linie_imagine[(x+sxrtp/2)*3+2]=(unsigned char)GLgammatable[(int)color[2]];
          }
    }
    else {
          color[0]=(unsigned char)(GLbackground.R*255);
          color[1]=(unsigned char)(GLbackground.G*255);
          color[2]=(unsigned char)(GLbackground.B*255);
          if(GLgamma==1.0) {
            linie_imagine[(x+sxrtp/2)*3]=color[0];
            linie_imagine[(x+sxrtp/2)*3+1]=color[1];
            linie_imagine[(x+sxrtp/2)*3+2]=color[2];
          }
          else {
            linie_imagine[(x+sxrtp/2)*3]=(unsigned char)GLgammatable[(int)color[0]];
            linie_imagine[(x+sxrtp/2)*3+1]=(unsigned char)GLgammatable[(int)color[1]];
            linie_imagine[(x+sxrtp/2)*3+2]=(unsigned char)GLgammatable[(int)color[2]];
          }
    }
  } // end of for(y= …
// write the current line of the image in the output file
  if(semop(semid, &P, 1)<0) {
     fprintf(stderr, "scena2 -- P error\n");
     exit(1);
  }
  if(lseek(frtrtp, (long)((y+syrtp/2)*(sxrtp)*3+3*i*DX+2*sizeof(short)), SEEK_SET)<0){
    perror("Seek");
  }
  if(write(frtrtp, linie_imagine+(3*i*DX), (i==(NTHREADS-1))?
            ((sxrtp)*3-3*i*DX):(3*DX))<0) {
    perror("Scriu");
  }
  if(semop(semid, &V, 1)<0) {
     fprintf(stderr, "scena2 -- V error\n");
```

```
      exit(1);
    }
} // end of for(x ...
  return NULL;
} // RTpartial
#endif
```

Another type of spatial decomposition technique, suitable to be used by a ray tracing application running in a parallel environment was proposed by Dippé and Swensen [8]. They divide the object space in non-intersecting adjacent cells and assign each spatial region (or more such regions) to one processing element (PE). The spatial regions could have parallelepiped or tetrahedral shapes.

The rendering process begins when the spatial region that contains the viewing point start firing rays at a specified resolution. Each ray has an associated pixel (home pixel) that will be colored at the end of the ray tracing process. When a ray enters a spatial sub-region (SSR), the associated PE computes the intersections with the objects contained in the spatial region. When the ray exits (SSR) it will eventually enter one region adjacent to (SSR) and the corresponding PE will receive an adequate message. The algorithm is more accurately described in [8] and [9]. This type of object space based decomposition will be implemented in a future version of YARTE.

3. Performance evaluation

The test version of the YARTE system has the following maximal parameters:

- NTHREADS – number of processes that are independently processing disjoint regions of the 3D scene. This constant could have a value of maximum 32.
- NRMAXCORPURI – maximum number of geometrical objects in a scene (actual value 1500)
- NRMAXSURSE – maximum number of light sources (actual value 10)
- NMAX_VARF_POLIGON – maximum number of vertices of a polygon (actual value 10)
- NMAX_VARF_MESH – maximum number of vertices of a polygonal mesh (actual value 250)

The following table gives the running times necessary to render a scene with N objects, using 1, 2, 4, 8, 16 and respectively 32 rendering processes. The times are expressed in seconds. The rendering times for the same scene on a Pentium II processor (working at 400MHz) are also presented.

N	$T_{Pentium}$	T_{SUN}					
		1	2	4	8	16	32
39	79.35	32.78	17.73	11.93	6.12	3.75	2.32
21	62.45	24.79	13.24	7.24	3.95	2.42	2.12
189	134.25	72.03	37.56	19.83	10.26	5.33	5.02

The system overhead was under 0.1 s for every test case and the rendering times decreased almost linearly with the number of processors. Figure 3 gives the dependency between the application real running time and the number of processors.

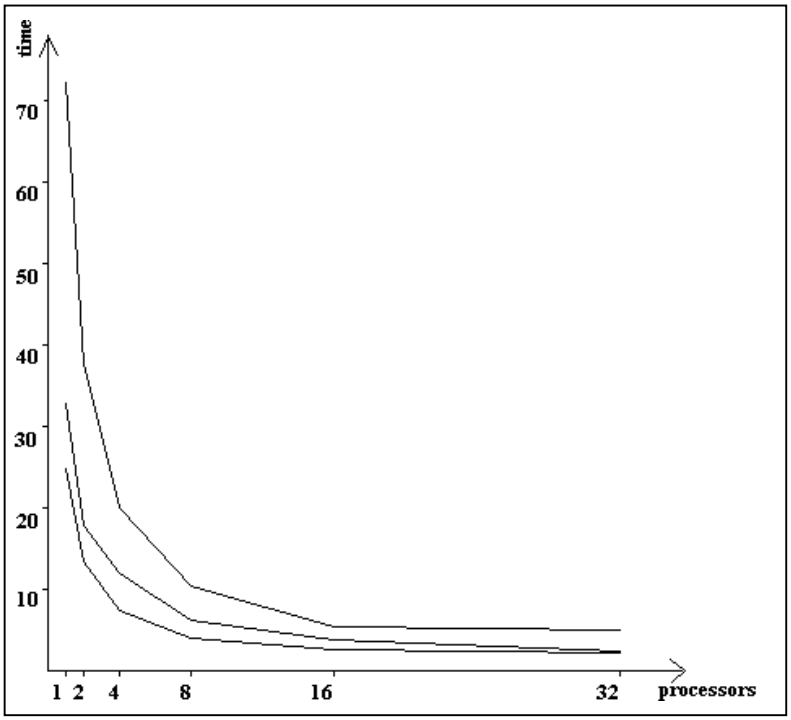

Fig. 3

Conclusions

The base engine of a ray-tracing realistic image synthesis application presented above runs in a parallel environment. The way parallelism was achieved was based on an image space decomposition technique. The paper describes the structure of the ray-tracing application and the implementation of the main modules. The system will be improved through: new geometric primitives, texture mapping, box shaped bounding volumes for better efficiency in intersection computations, an object space decomposition technique for better parallelism, a new WEB interface.

The excellent running times of the application make an idea about the performances of the Sun Enterprise 10000 computing environment.

REFERENCES

1. *J. Foley, A. van Damm, S. Feiner, J. Hughes,* Computer Graphics Principles and Practice. Addison Wesley Publishing Company, 1992
2. *M. Cohen, J. Wallace,* Radiosity and Realistic Image Synthesis, Academic Press Professional, Harcourt Brace & Company Publishers, 1993
3. *A. Glassner,* Principles of Digital Image Synthesis, Morgan Kaufmann Publishers Inc., San Francisco California, 1995
4. *C. Lindley,* Practical Ray Tracing in C, John Wiley and Sons Inc., Wiley Professional Computing Series, New York, 1992
5. *http://www.prenhall.com/hill* Prentice Hall Home Page, Upper Saddle River NJ07458
6. *http://www.hpc.pub.ro* National Center of Information Technology, High performance computing home page, Department of Computer Science and Engineering, Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest
7. *R. Kuchkuda,* An Introduction to Ray Tracing, NATO ASI series, vol **F40**, appeared in Theoretical Foundations of Computer Graphics and C.A.D., Springer Verlag Berlin, Heidelberg, 1988, pp. 1039-1060
8. *M. Zaharia,* Algoritmi paraleli/ distribuiți în domeniul graficii de sintezã şi al prelucrãrilor de imagini, Raport de Cercetare in cadrul proiectului TEMPUS S-JEP 07101/94, apãrut în: Aspecte ale elaborãrii algoritmilor distribuiti, Universitatea POLITEHNICA Bucuresti, Politecnico di Torino, 1996, pag. 370-493.
9. *P. Dew,* Parallel Processing for Computer Vision and Display, Addison Wesley Publishing Co., 1989
10. *M. Zaharia*, Some application architectures for realistic image synthesis in a distributed environment, ANALELE Universitãții "Ştefan cel Mare" din Suceava, Secțiunea electricã, ISSN 1222 – 4316, vol **6**, Nr. 10, 1999, pp. 1-7