Implementation and Evaluation of POV-Ray on Desktop Grids: Parallel Rendering of 3D Images and Animations

A. M. Riad[1], A. E. Hassan[2], Q. F. Hassan[1]
[1]Department of Information Systems, Faculty of Computers and Information Systems, Mansoura University, Mansoura, Egypt
[2]Department of Electrical Engineering, Faculty of Engineering, Mansoura University, Mansoura, Egypt

ABSTRACT: This paper discusses the implementation details of a grid-based rendering framework for POV-Ray on desktop grids. Our goal is to present how enterprises can build desktop grids in Windows environment in order to enable semi-real time rendering for 3D models, both images and animations, defined with POV-Ray. Algorithms, code and technical details are given for easy and efficient implementations. We think this work could be useful for both researchers and developers who are interested in the grid computing technology and its applications.
KEYWORDS: Grid Computing, Computer Graphics, Desktop Grids, Ray Tracing and POV-Ray,.NET Framework, Alchemi, parallel Rendering for 3D Models.

Introduction

In our previous paper titled, “On Harnessing Desktop Grids for Semi-Real Time 3D Rendering: A Case Study on POV-Ray” [RHH11], we introduced the main concepts of the ray tracing and grid computing methodologies as well as the installation and configuration details of two enabling technologies, namely POV-Ray and Alchemi, respectively. The paper also discussed our grid-based framework that enables the parallel rendering of the POV-Ray models on desktop grids. This work continues the original work by exploring the implementation details of the grid-based rendering framework of both 3D images and animations. The remainder of this paper is organized as follows: Implementation details of the parallel rendering algorithm of images are discussed in section 1. Section 2 discusses the implementation details of the parallel rendering algorithm of animations. Experimental results and the evaluation of our algorithms are discussed in Section 3. Last section concludes the paper and points out the future work.

1. Parallel Image Rendering

Each image is composed of a set of rows and columns of pixels –this is what is known as the image resolution. The ability to slice an image up into a set of meaningful segments (e.g., of rows, columns or rectangular tiles), in order to concurrently render them on different computers, can drastically decrease the time needed to generate the final output.

In our implementation, we decided to subdivide the rendering task of one image into a number of subtasks where each subtask encompasses a set of rows. That is, our algorithm has three consequent phases: Render, Crop and Combine (RCC).

Fig. 1 illustrates the rendering flowchart of a single image on the grid using our algorithm. As listed, the proposed algorithm is simple to understand and implement:

1. The end user specifies the height and width of the target image along with the passed ".pov" file. In addition, the user can optionally define the number of rows per job and pass it to the client application.
2. The client application creates the grid jobs and submits them to the manager node. Each job has a rendering command for a portion of the target image.
3. The manager node passes the queued jobs to the connected executors.
4. Each executor launches the POV-Ray application by starting an external process in Windows using the/ a "Process" class provided by .NET Framework.
5. POV-Ray generates the (partial) image according to the specified options and then saves it to the specified output path. Implementers should note that all grid nodes should have a shared output path so that the executors can save the rendered outputs. This can be accomplished by using a network drive or a database where all nodes have access to.
6. Each of the completed jobs generates a full-sized image which has an empty (black) portion in addition to the segment that contains the rendered data.

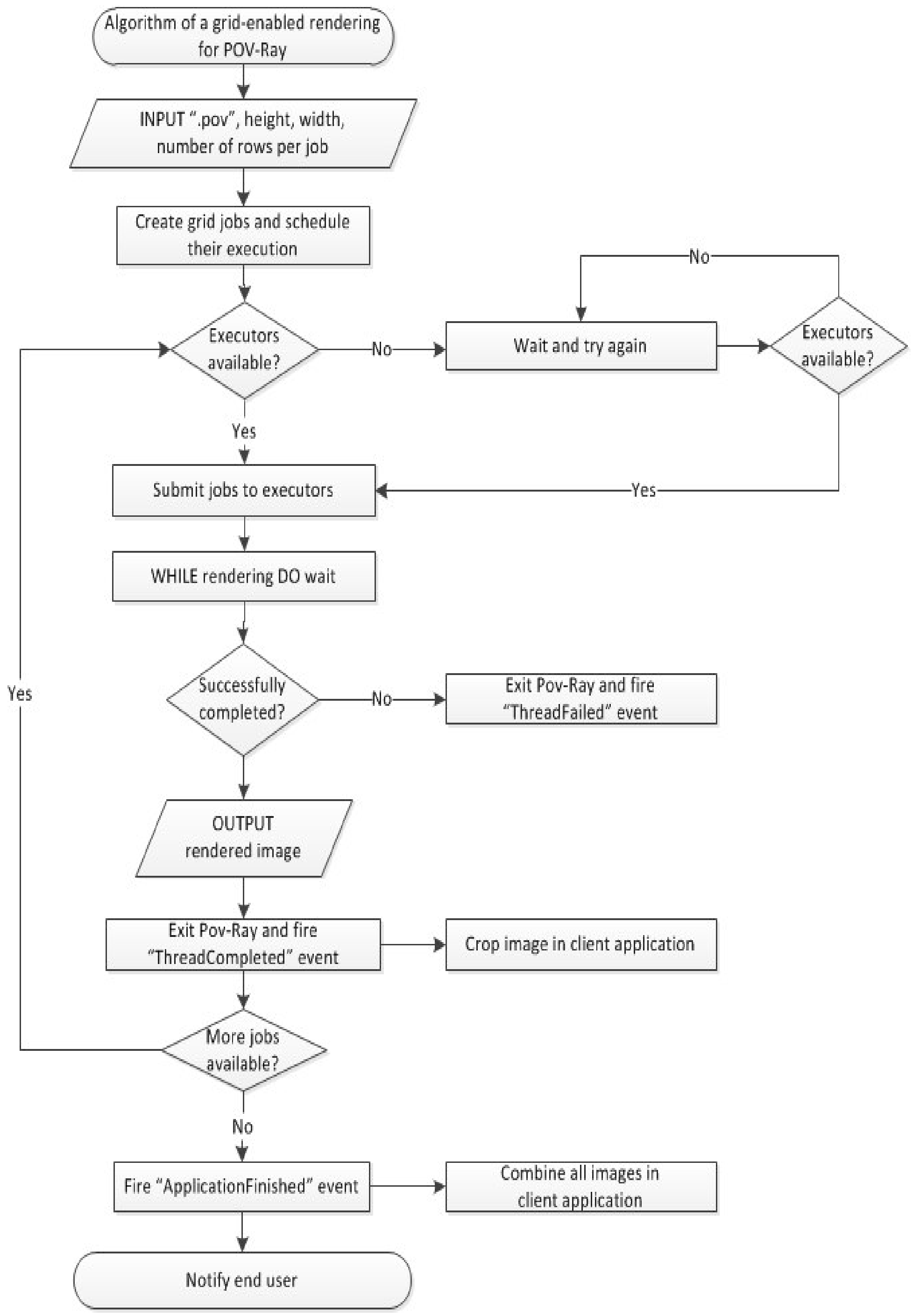
Algorithm of a grid-enabled rendering for POV-Ray
INPUT ".pov", height, width, number of rows per job
Create grid jobs and schedule their execution
Executors available?
No
Wait and try again
Executors available?
No
Yes
Yes
Submit jobs to executors
WHILE rendering DO wait
Successfully completed?
No
Exit Pov-Ray and fire "ThreadFailed" event
OUTPUT rendered image
Exit Pov-Ray and fire "ThreadCompleted" event
Crop image in client application
More jobs available?
Yes
No
Fire "ApplicationFinished" event
Combine all images in client application
Notify end user

Figure 1. Grid-enabled rendering for one POV-Ray Image

7. When one job is finished, the client application crops the rendered image to create a smaller image with only the non-empty segment –after removing the black portion.
8. If any of the jobs fails, the client application will queue it in a temporary list in order to re-process it again either on the grid or on the client machine.
9. When all jobs are successfully completed, the client application merges the cropped images according to their original order to produce the final image.

We have tested our algorithm during its implementation to generate the outputs of each step so that the readers can better understand the idea. We have used four executors to render the "stackerday.pov" scene, which is one of the installation components of the POV-Ray application. As illustrated in Fig. 2, the generation process of our test scenario took place on the four executors by splitting the rendering phase into four simpler rendering subtasks. The generated outputs of each of these subtasks are also presented including the final image.

As illustrated in listing 1, we need to pass the connection string of our grid in order to make use of it. The connection string includes the address and port of the Manager node as well as the users' credentials. We instruct the deployed grid that it is going to deal with our application by passing our class which implements the GThread type (i.e., PovRayThread).

Listing 1. Snippet from the client application showing how we could connect to the grid

```
//Get settings from user and connect to the grid
GConnection gc = GConnection.FromConsole("localhost", "9000", "user",
user");

//Create a new grid application
app = new GApplication(gc);
app.ApplicationName = "POV-Ray";

//Add the module containing the PovRayThread to the application manifest
app.Manifest.Add(new ModuleDependency(typeof(PovRayThread).
Module));
```

As illustrated in listing 2, we create a set of grid jobs according to the width of the target image and the number of rows included in each job. The created jobs will include the rendering arguments that will be passed to POV-Ray via a command-line.

Listing 2. Snippet showing the creation of the rendering jobs

```
for (int i = 0; i < numberOfThreads; i++)
{
   PovRayThread job = new PovRayThread();
   job.PovRayPath = pvenginePath;
   job.PovFileName = povFileName;
   job.OutputPath = outputPath;
   job.ImageFileName = imageFileName;
   job.Height = height;
   job.Width = width;
   job.StartRow = startRow;
   job.EndRow = endRow;
   if ((i + 1) == numberOfThreads)
      job.NumberOfRows = remainingRows;
  else
     povrayThread.NumberOfRows = numberOfRowsPerThread;
   app.Threads.Add(povrayThread);
}
```

As shown, the application creates a number of grid jobs, each of which has a set of properties that defines the rendering settings:

- **PovRayPath:** Sets the path of the POV-Ray application ("pvengine.exe"). The POV-Ray executable must exist in the same location on all executors; otherwise some jobs may fail. Although we have hard coded the value of this property in our sample code, implementers should place it in the configuration file (e.g., App.config or Web.config) so that they can change it whenever needed. The purpose of hard coding such values is simplicity; however, configuration values offer higher flexibility.
- **PovFileName:** Sets the path of the input ".pov" file.
- **OutputPath:** Sets the output path where the rendered image will be saved. This value combined with the value of ImageFileName is mapped to the +O option in the rendering arguments.
- **ImageFileName:** Sets the name of the rendered image.
- **Height:** Sets the height of the rendered image. This value is mapped to the +H option in the rendering arguments.
- **Width:** Sets the width of the rendered image. This value is mapped to the +W option in the rendering arguments.
- **StartRow:** Sets the row where the rendering task begins. This value is mapped to the +SR in the rendering arguments.

- **EndRow:** Sets the row where the rendering task stops. This value is mapped to the +ER in the rendering arguments.

As illustrated in listing 3, we get notified about the status of our rendering jobs by handling Alchemi events in our client application. It also illustrates how we can start jobs processing on the grid.

Listing 3. Snippet from client application showing subscription for Alchemi events

```
//Subcribe to events
app.ThreadFinish += new GThreadFinish(ThreadFinished);
app.ApplicationFinish += new GApplicationFinish(ApplicationFinished);
app.ThreadFailed += new GThreadFailed(ThreadFailed);

//Start the grid processing
app.Start();
```

As illustrated in listing 4, the implementation of our algorithm is simple. The code basically launches a Windows process to start the POV-Ray application, and executes the rendering job by passing a command-line with a set of arguments that defines the rendering options.

Listing 4. Snippet from PovRayThread class showing "Start()" method which contains the rendering logic

```
public override void Start()
{
   string arguments = String.Format("+W{0} +H{1} +A +SR{2} +ER{3}
+O\"{4}\" /EXIT /RENDER \"{5}\"", width, height, startRow, endRow,
outputPath + "\\" + imageFileName, povFileName);

   Process process = new Process();
   process.StartInfo.FileName = povrayPath;
   process.StartInfo.Arguments = arguments;
   process.StartInfo.WorkingDirectory =
Path.GetDirectoryName(povFileName);
   process.StartInfo.UseShellExecute = false;
   process.StartInfo.CreateNoWindow = true;
   process.StartInfo.RedirectStandardError = false;
   process.Start();
   process.WaitForExit();
}
```

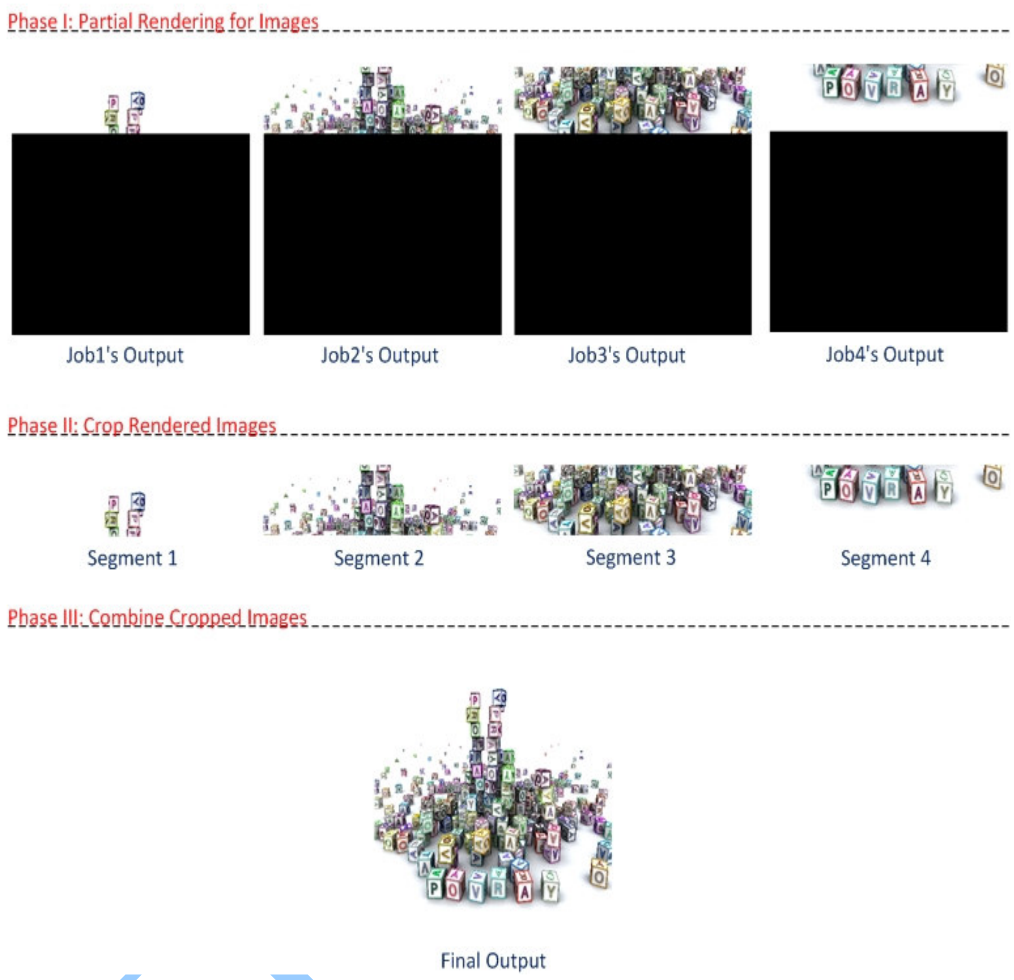
Phase I: Partial Rendering for Images
Job1's Output
Job2's Output
Job3's Output
Job4's Output
Phase II: Crop Rendered Images
Segment 1
Segment 2
Segment 3
Segment 4
Phase III: Combine Cropped Images
Final Output

Figure 2. Image rendering phases using the RCC algorithm

In addition to the aforementioned rendering arguments, we have used the following three options:

- **+A:** Enables or disables anti-aliasing for the generated image. Passing this option will instruct POV-Ray to generate a high-quality output, but this could considerably increase the amount of time taken to render the image.
- **/EXIT:** Instructs POV-Ray to exit after completing the rendering task. Although this option is not required to complete the rendering tasks, we decided to use it to save the computing resources available on the connected executors. The user is able to launch as many instances of POV-Ray as he needs by unmarking the “Keep Single Instance” option in the “Options” menu. However, this action will cause our client

application to launce numerous instances of POV-Ray on each executor which may lead to a memory leak.

- **/RENDER:** Instructs POV-Ray to render the passed ".pov" file.

2. Parallel Animation Rendering

Any animation movie is composed of a sequence of still pictures known as frames. The quality and smoothness of the rendered animation depends on both the quality of the used images and the number of frames per second. In other words, the higher the quality of the used images and the more frames per second, the smoother and more photorealistic the rendered animation. The average number of frames in an animation usually varies from 24 to 30 frames per second. Thus, with a simple calculation, the rendering of a professional 20-second scene, at the rate of 30 frames per second, would require an artist to render 600 images (20x30). Rendering 600 images on a single computer may take hours or even days to complete –even while using a modern and powerful hardware.

With desktop grids, the task of rendering different frames can concurrently take place on the available network of computers. Furthermore, finer jobs, where each frame is broken into several sub-frames (slices), or coarser jobs, where several images are grouped together, can be created to render the whole scene.

In POV-Ray animation, settings are defined in the ".ini" file which is placed along with the actual ".pov" file. This file normally includes information about the initial frame, the final frame, the initial clock value, and the final clock value. Additional information such as anti-alias toggle, anti-alias threshold and depth, and cyclic animation could also be defined in the ".ini" file. With ".ini" files, users are freed from manually creating the animation sequence. In other words, ".ini" files are responsible for automatically creating the animation frames out of the assigned ".pov" according to the defined settings.

Animation in POV-Ray is managed by means of a clock variable where each frame is assigned with a float value between 0.0 and 1.0 that defines its ordering in the whole scene. Grid implementers can manually obtain clock values between 0.0 and 1.0 using this simple equation: frame number / total number of frames. For example, calculating the clock indicators for 30-frames would generate a sequence of values of 1/30 (0.333), 2/30 (0.666), 3/30 (0.1), 4/30 (0.133) and so forth.

Michael Head introduced the use of an unofficial release of POV-Ray namely, MegaPOV to render a set of grid jobs created with .NET and Alchemi where each job contains only one frame [Hea07]. As will be discussed later, creating fine-grained jobs that contain single frames may incur a performance overhead especially when created for very small and simple frames, due to excessive network traffics, as well as starting and ending the rendering processes.

In this section, we introduce the creation of jobs that encompass groups of frames which are rendered on the grid using the official releases of POV-Ray. To accomplish this goal, the rendering command should include the initial and final frames of the scene being processed. These values will change with the animation and rotation of the scene's objects.

Fig. 3 illustrates the flowchart of the video rendering process which is composed of the following simple steps:

1. The end user specifies the height and width of the frames, the total number of frames, and the number of frames in each grid job along with the passed ".pov" file.
2. The client application creates the grid jobs and submits them to the manager node. Each job has a rendering command for the targeted image.
3. The manager node passes the queued jobs to the connected executors.
4. Each executor launches the POV-Ray application by starting an external process in Windows using the "Process" class provided by .NET Framework.
5. POV-Ray generates the frames according to the specified options and then saves them to the specified output path. Again, implementers should note that all the grid nodes should have a shared output path for the executors to save the rendered outputs in. This can be accomplished by using a network drive or a database that all nodes have access to.
6. If any of the jobs fail, the client application will queue it in a temporary list in order to re-process it again, either on the grid or on the clients' machine.
7. When all jobs are successfully completed, implementers should make sure that all rendered jobs are in the correct order. Rendered tasks do not usually return to the client application in order. This is subject to the availability of grid resources and the processing speed of each node. This means that implementers are required to write a small module or a set of functions to re-order the rendered frames. The re-ordering module should uniquely identify each rendering task before sending it to the grid in order to be able to properly reorganize them upon completion. To make the organizing process easier, this module creates a temporary subdirectory for each rendering task in the root output directory to separate the generated frames from each other.

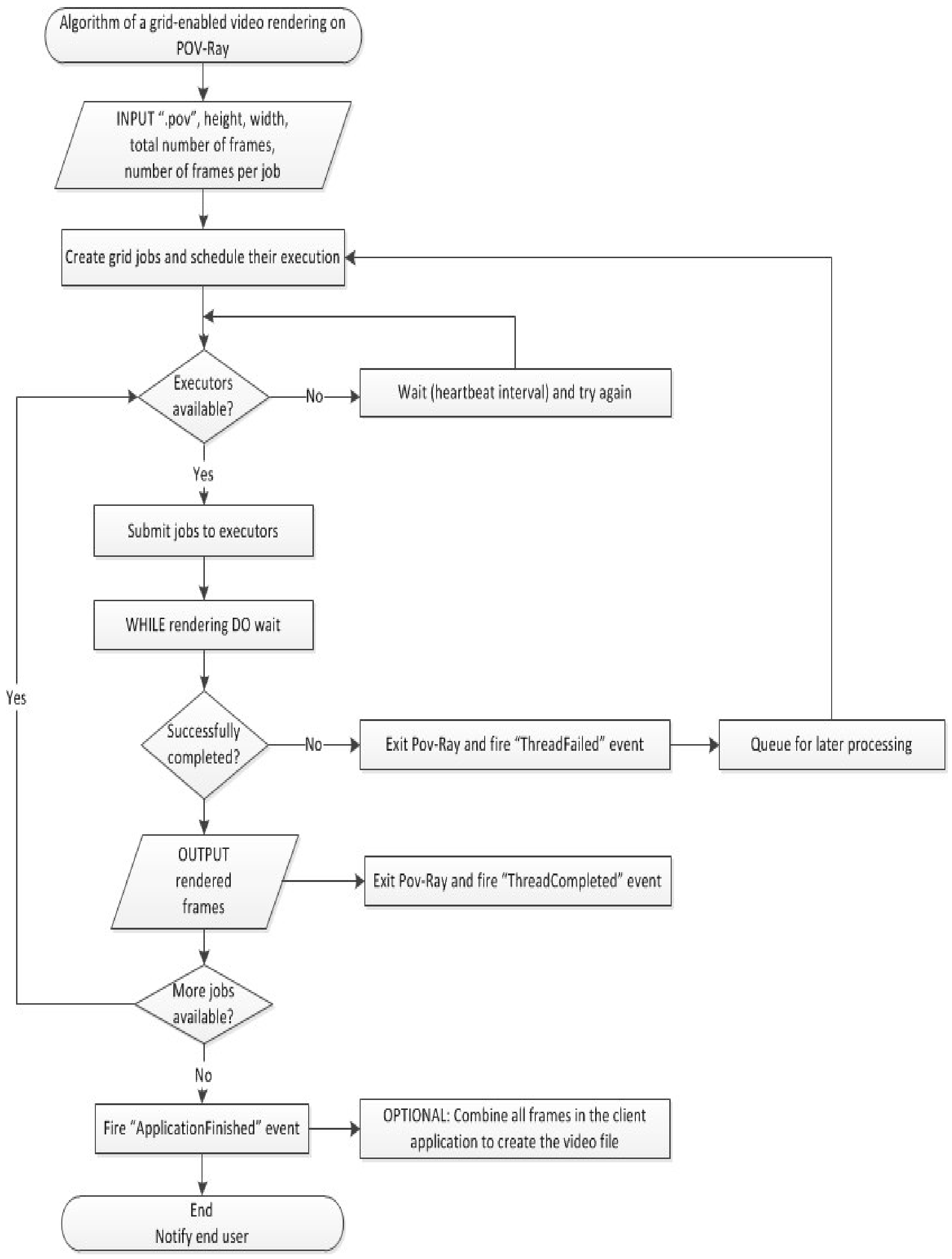
Algorithm of a grid-enabled video rendering on POV-Ray
INPUT ".pov", height, width, total number of frames, number of frames per job
Create grid jobs and schedule their execution
Executors available?
No
Wait (heartbeat interval) and try again
Yes
Submit jobs to executors
WHILE rendering DO wait
Yes
Successfully completed?
No
Exit Pov-Ray and fire "ThreadFailed" event
Queue for later processing
OUTPUT rendered frames
Exit Pov-Ray and fire "ThreadCompleted" event
More jobs available?
No
Fire "ApplicationFinished" event
OPTIONAL: Combine all frames in the client application to create the video file
End
Notify end user

Figure 3. Grid-enabled rendering for one POV-Ray animation

After the frames rearranging process takes place, implementers are able pass the generated frames to a tool that combines them creating a video file. This can either be done manually by the artist or defined automatically by a simple code in the handler of the "ApplicationFinished" event. A number of tools that can combine ".bmp" files into ".avi" or ".mpeg" videos are available. EasyBMPtoAVI is an example of free ".avi" tools (http://easybmp.sourceforge.net/download.html).

As illustrated in listing 5, our video rendering algorithm is mainly concerned with the creation of grid jobs that contain several frames. We calculate the values of the initial and final clock values as well as the initial and final frames while iterating through the total number of frames specified by the end user. Since we use the same equation ("i / numberOfFrames" where "i" is the current frame being processed) to calculate the initial and final clock values, we have defined the "getInitialValues" as a flag variable to differentiate between both values. Our differentiation condition is: "(i + 1) % numberOfFramesPerJob = 0" which computes the remainder after dividing (i + 1) by numberOfFramesPerJob using the modulus operator (%). If the remainder equals zero then a new job will be created to encapsulate a new set of frames; otherwise the current frame will be added to the previously created grid job. The same concept applies to the calculation of the initial and final frames in each iteration. In other words, the value of the initial frame will only increment with the creation of each new grid job, whereas the value of the final frame will only increment while the previously created job is being used.

The new properties we defined here are:

- **InitialClock:** Sets the initial clock value. This value is mapped to the +KI option in the rendering arguments.
- **FinalClock:** Sets the final clock value. This value is mapped to the +KF option in the rendering arguments.
- **InitialFrame:** Sets the initial frame to be rendered in the animation sequence. This value is mapped to the +KFI option in the rendering arguments.
- **FinalFrame:** Sets the final frame to be rendered in the animation sequence. This value is mapped to the +KFF in the rendering arguments.

Listing 5. Snippet from a client application showing the creation of the grid jobs that contain several frames

```
//Flag variable to toogle between the calculation of the initial and final
frames and the initial and final clock values
bool getInitialValues = true;
int initialFrame = 1, finalFrame = 1;
double initialClock = 0, finalClock = 0;

//Iterate through the toal number of frames
for (int i = 1; i <= numberOfFrames; i++)
{
   if (getInitialValues)
   {
      //Calculate initial clock and initial frame
      initialClock = i / numberOfFrames;
      initialFrame = i ++;
      getInitialValues = false;
   }
   if ((i + 1) % numberOfFramesPerJob == 0)
   {
      //Calculate final clock and final frame
      finalClock = i / numberOfFrames;
      finalFrame = i ++;
      getInitialValues = true;

      PovRayVideoThread job = new PovRayVideoThread();
      povrayThread.PovRayPath =
@"C:\Users\Qusay\AppData\Roaming\POV-Ray\v3.6\bin\pvengine.exe";
      job.PovFileName = povFileName;
      job.OutputPath = outputPath;
      job.InitialClock = initialClock;
      job.FinalClock = finalClock;
      job.InitialFrame = initialFrame;
      job.FinalFrame = finalFrame;
      job.Height = height;
      job.Width = width;
      job.Threads.Add(job);
    }
}
```

Listing 6 illustrates the new set of arguments used to render a group of video frames. Since the remaining logic of the “Start()” method is identical to the one used to render POV-Ray images, and for the sake of simplicity, we have excluded it.

Listing 6. Snippet from PovRayVideoThread class showing the command-line arguments used by the “Start()” method

```
string arguments = String.Format("+W{0} +H{1} +O\"{2}\" +KI\"{3}\"
+KF\"{4}\" +KFI\"{5}\" +KFF\"{6}\" /EXIT /RENDER \"{7}\"", width,
height, outputPath, initialClock, finalClock, initialFrame, finalFrame,
povFileName);
```

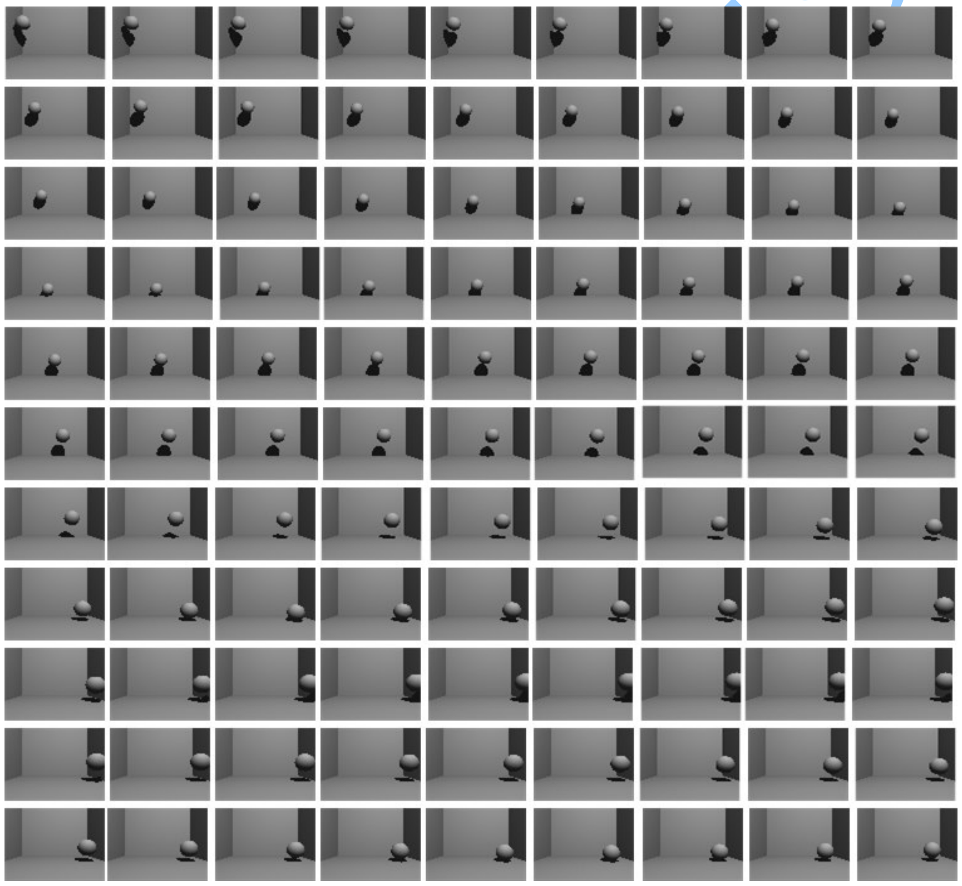
Figure 4. Grid-enabled rendering for one POV-Ray animation

Fig. 4 illustrates 99 frames of the “bounce.pov” scene, which were rendered using our algorithm. The “bounce.pov” scene is one of the installation components of the POV-Ray application and it is, simply, for a ball falling from the top left side and keeps bouncing towards the right side until it hits the wall, and then starts bouncing back up to the left side

3. Experimental Results

We have evaluated our algorithms using a test-bed composed of three laptops with different hardware specifications. We first evaluated the rendering of the "bounce.pov" models at different resolutions (800 x 600, 1024 x 768, 1366 x 768 1600 x 1200 pixels) separately on each node, and then we connected the three nodes together to test the grid-enabled framework.

Table 1 lists the hardware specifications of our test-bed. Windows 7 was installed on all nodes and a 100mbps switch was used to connect them.

Since Node 1 is the most powerful node in our test-bed, we have installed both the Alchemi Manager and the Executor components on it to play both roles. The client application was also installed on Node 1, whereas all outputs were saved on Node 2 in a shared directory assigned with the needed access permissions.

Table 1. Hardware Specifications

Node Number	Processor	RAM
Node 1 *(Manager + Executor + Client Application)*	Intel Core i5 2.3 GHz	4.00 GB
Node 2 *(Executor + Output Storage)*	Intel Core i3 2.53 GHz	3.00 GB
Node 3 *(Executor)*	Intel Core 2 Due 2.26 GHz	4.00 GB

Before delving into the evaluation details, we should point out that while evaluating our solution we have tried different sizes for our grid jobs. We noticed that both very large and very small jobs were the least efficient and slowest of the techniques. Rendering very large jobs that contain many rows in each slice takes a long time as the total number of the grid jobs are reduced resulting in a semi-serial rendering (with the possibility that some of the connected computers are not assigned with some jobs) rather than parallel rendering. Contrarily, creating very small grid jobs incurs a large number of roundtrip traffics between the connected nodes, and extra time is needed to start and end the rendering processes as well.

In order to test the grid-based techniques, a size of 50 - 100 rows per job for rendering images and 50 - 100 frames per job for rendering animations was the most efficient size for our implementation. Certainly this number should increase and decrease accordingly with the total number of the grid nodes and image rows/animation frames. Also, the size of the jobs should be smaller in case of building a grid out of computers with different specifications in order to enable the grid manager to better schedule these jobs. In our test scenario, we have configured each job to include 80 rows for the image rendering case, and 80 frames for the animation rendering case.

As illustrated in Fig. 5 and 6, the total rendering time of the "stackerday.pov" image is drastically reduced when rendered on the grid:

- The rendering at a resolution of 800 x 600 pixels on the grid is 205%, ~247% and 440% faster than the serial rendering on Node1, Node 2 and Node 3, respectively.
- The rendering at a resolution of 1024 x 768 pixels on the grid is 225%, ~329% and ~478% faster than the serial rendering on Node1, Node 2 and Node 3, respectively.
- The rendering at a resolution of 1366 x 768 pixels on the grid is ~217%, ~320% and ~410% faster than the serial rendering on Node1, Node 2 and Node 3, respectively.
- The rendering at a resolution of 1600 x 1200 pixels on the grid is ~226%, ~304% and ~530% faster than the serial rendering on Node1, Node 2 and Node 3, respectively.

As illustrated in Fig.7 and 8, the total rendering time of the "bounce.pov" animation is drastically reduced when rendered on the grid:

- The rendering at a resolution of 800 x 600 pixels on the grid is ~232%, ~339% and ~448% faster than the serial rendering on Node1, Node 2 and Node 3, respectively.
- The rendering at a resolution of 1024 x 768 pixels on the grid is ~253%, ~293%, ~390% faster than the serial execution on Node1, Node 2 and Node 3, respectively.
- The rendering at a resolution of 1366 x 768 pixels on the grid is ~266%, ~336%, ~427% faster than the serial execution on Node1, Node 2 and Node 3, respectively.
- The rendering at a resolution of 1600 x 1200 pixels on the grid is ~165%, ~331%, ~381% faster than the serial execution on Node1, Node 2 and Node 3, respectively.

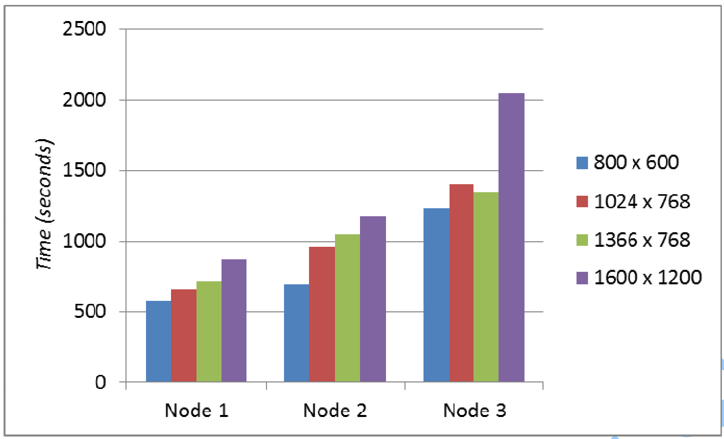

Figure 5. Serial image rendering on single nodes

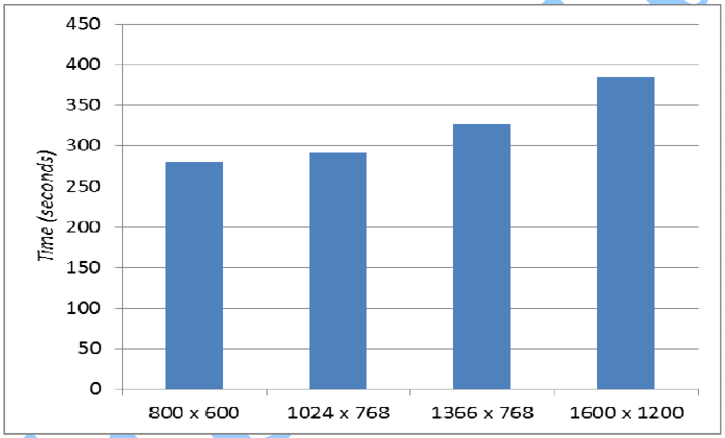

Figure 6. Parallel image rendering on a grid of three nodes

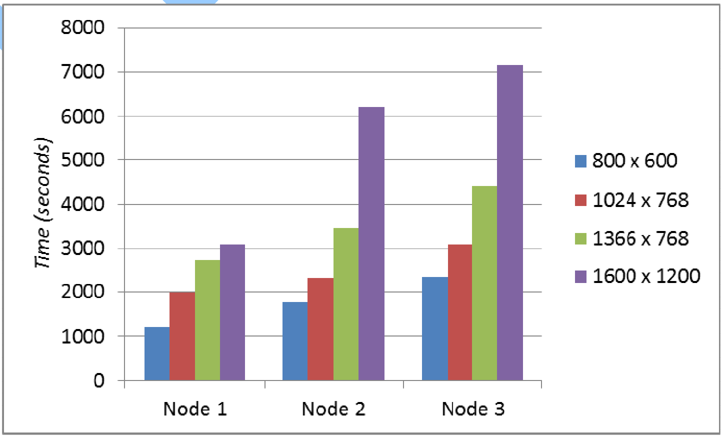

Figure 7. Serial animation rendering on single nodes

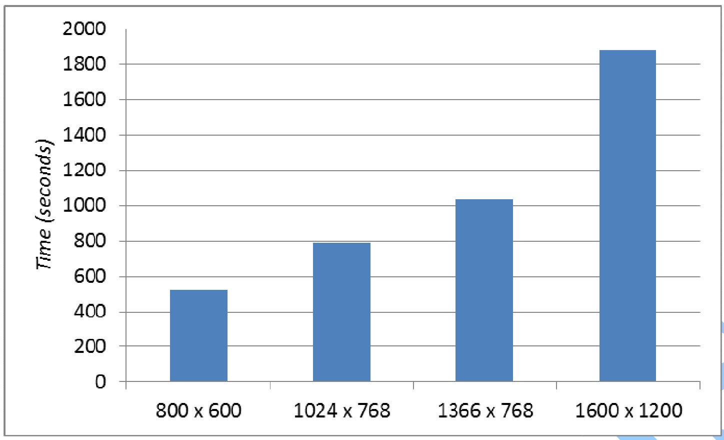

Figure 8. Parallel animation rendering on a grid of three nodes

Conclusion and Future Work

This work presented a grid-enabled implementation of the Ray Tracing technique on a Windows environment using POV-Ray, .NET Framework and Alchemi. The technical details and sample source code were included in order to enable other researchers and practitioners to implement similar solutions. Moreover, a comparison between the serial and the grid-enabled implementations of POV-Ray was presented to show how enterprises can benefit from the proposed technique.

Results show that the speed of the 3D rendering task of both images and video frames increases proportionally with the number of the connected workers. This means that we can render the presented images and animation frames in a few seconds if we are able to build a grid of 50 or 100 computers. Building such a grid is trivial since most enterprises, including small ones, usually have more than 50 computers. By achieving a semi-real time rendering for complex models, enterprises can do more work in much less time without spending any extra cost.

It is worth mentioning that the utilization of desktop grids for 3D rendering is not only limited to ray tracing or POV-Ray. Implementers can harness the power of grid computing with other 3D graphics and modeling software like Autodesk Maya, Blender, etc. This can be easily achieved by creating grid-enabling plug-ins for these products using the provided APIs and scripting languages, for example [CSL06].

Service-oriented Architecture (SOA) is a set of design and development methodologies that are being widely adopted by software implementers to enable an easy and effective integration between different software systems –both modern and legacy [Has09] [RHH10]. A service is a well-defined, self-contained software component that encapsulates business and/or technical functionalities related to a specific domain in an abstract and reusable manner. We plan to service-enable our implementation so that developers from different enterprises can make use of it without worrying about the underlying details and complexities.

Acknowledgment

The authors would like to thank Rana Sherif for her generosity and continues assistance in editing their work.

References

[CSL06] **A. Chong, A. Sourin, K. Levinski -** *Grid-based Computer Animation Rendering*, Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia, 2006.

[Has09] **Q. F. Hassan -** *Aspects of SOA: An Entry Point for Starters*, Anale. Seria Informatica. Vol. VII fasc. 2 – 2009.

[Hea07] **M. Head -** *SOA grid design patterns for computer graphics animation: Using Alchemi to render POV-Ray animations on a grid*, 2007, developerWorks, IBM.

[RHH10] **A. M. Riad, A. E. Hassan, Q. F. Hassan -** *Design of SOA-based Grid Computing with Enterprise Service Bus*, International Journal on Advances in Information Sciences and Service Sciences, 2010.

[RHH11] **A. M. Riad, A. E. Hassan, Q. F. Hassan -** *On Harnessing Desktop Grids for Semi-Real Time 3D Rendering: A Case Study on POV-Ray*, Anale. Seria Informatica. Vol. IX fasc. 2 – 2011.