Ginger-Video-3D - Adaptação de uma ferramenta de *rendering* gráfico/codificação vídeo para execução paralela em sistema *peer-to-peer* de partilha de ciclos.

João Cruz Morais
Rua Alves Redol 9 1000-029 Lisboa Portugal
joao.c.morais@ist.utl.pt

Abstract

Grid computing is a concept usually associated with institution-driven networks assembled with a clear purpose, namely to address complex calculation problems or when heterogeneity and geographical dispersion of the participants is a key factor. In this context, and given the cost of setting up such an infrastructure, a regular home user willing to take advantage of distributed processing is left without a viable option. Even if Grid access was open to the general public, a home user would not be able to express task decomposition without clearly understanding the program internals. In Ginger, this issue is addressed in a different manner. Users are not required to modify an application they already use, needing instead to access an available format description of the application input/output, in order to decompose a job in smaller tasks that may be distributed and executed in cycle-sharing machines. In the context of this research, an architecture is proposed on which an application can be transparently adapted based solely on input and output data formats. A first phase analyses content and builds sub-tasks that are deployed over the network. In converse fashion, results are analyzed and aggregated according to the output format. The work was tested with a video compression application as well as a ray-tracing renderer, with positive results, especially in the last one, since it is a CPU-intensive application and its I/O has a smaller data load when compared with video compression.

Keywords: grid computing, load-balancing, cycle-sharing, video encoding, video format analysis

1. Introduction

In this day and age, a quick observation that can be made about personal computer usage is that they are either inactive or active with a very small load which means that most CPU cycles are wasted without any relevant operation being made over them. Along with the Internet growing, in recent years several applications have appeared that try to leverage this wasted cycles to perform useful processing of CPU-intensive applications, the most known being the SETI@Home [2] project. However most of these projects have a limited scope on how the Grid resources can be used. In most cycle sharing projects users freely give their resources and are not allowed to run their applications. In such cases the user is motivated to give their CPU time when there's either monetary compensation involved (Plura [12]) or the project's purpose is the humanity's greater good. (Folding@Home, Seti@Home)

In projects like Ourgrid [3] and Integrade [4] work has been done to actually allow any user to enter a cycle-sharing Grid and submit jobs to other peers, but nevertheless there's still a high barrier of entry

since these projects leverage the grid to execute bag-of-tasks applications leaving at the users hands how to create those tasks. If the user has a good understanding of the application internals it can be relatively easy create these tasks, but otherwise the user is restricted on acting on its inputs/outputs.

Ginger-Video-3D acts upon this last restriction: the user either doesn't have access to the application source code or doesn't have the knowledge or time to study it, in order to discover how to distribute the computational flow to other peers. Since the application will remain unmodified it's required to change paradigm and instead of having a modified application receiving the same input at every peer, there will be an unmodified application in every peer each one receiving a small portion of the global input. These smaller inputs are constructed from an XML format description that tells Ginger how to parse and decompose the files. This format can be written by the user or there might be the case that it has already been written and published by someone else. A great advantage of working with smaller inputs is expressed when the original file is large and the work is leveraged to peers connected through low bandwidth links. In the case of video for example, each peer will receive and process a small scene of the whole video. The results are then gathered and merged at the source and finally written to the output file.

2. Related Work

GiGi [1] is the predecessor of this work and as such some ideas have passed along but in this paper we enter in more detail regarding the steps required to fully adapt computational intensive applications. APST-DV [8] has a case study with MPEG4 encoding which has also explores input division load techniques similar to our approach but uses external programs to do the file splitting/merging. BOINC [2] is a middleware and infrastructure for developing distributed computing applications, which became popular for hosting SETI@Home. BOINC projects have a restricted scientific scope and rely heavily on the processing power of voluntary peers around the world and although the unit of work is presented as a concept, every project uses the same unit when there are clearly units from different projects don't have the same computational cost. The Globus Toolkit [10] is an important reference, since it is the lead implementation for OGF (Open Grid Forum) standards and organizes the many components usually found in a Grid such as job scheduling, resource sharing, peer monitoring and user authentication, behind a SOA layer which can be integrated with workflow modelling. Butt et al. [11] purposes a prototype for executing applications over P2P overlays such as Pastry [9] focusing on easy resource discovery and a fair reputation system. This system is restricted to Java applications since it relies on the JVM has a sandbox while our project doesn't have this restriction. Ourgrid [3] aims to allow any user to run bag-of-tasks applications over a Grid of federated clusters. Applications are executed over a sandbox and fair resource usage is handled with a network of favours that rewards users according to their contribution to the Grid. Integrade [4] has similar goals but works on a client-server model where each job must be routed through the cluster server to other peers. It also allows for MPI application execution over a checkpoint system for handling incomplete jobs. Both these projects have failed to reach major acceptance since there is yet to be ported any useful application to work on top of this grid

infrastructures. CCOF [11] is a cycle-sharing P2P architecture that purposes solutions for computational intensive problems like the ones faced by the SETI@Home project. The application load is distributed by the Wave Scheduler which organizes the resources by geographic regions in order to take advantage of night periods which usually have less activity and although a generic P2P infrastructure like ours is suggested, no programming model is available yet. Regarding multimedia processing over a distributed environment, Ewert [6] suggests applying grid technologies for analyzing multimedia content for detecting transitions, faces, text, etc. in video files. In this work a video is submitted to a job manager service which splits the video in smaller scenes according with its bit-rate and picture dimensions and schedules the jobs for execution in a cluster of nodes that work together to extract all the information from that video. This application has a similar approach to ours, but splitting video for transition detection might require more work than simply looking for key frames. Concerning the adaptation of existing applications, de Lara et al. [7] details transparent adaptation mechanisms through APIs, such as controlling the fidelity of components (low-res images when displaying content over low-bandwidth links) and applying a range of policies to a group of programs. It is also presents a new system for interfacing with production applications from different vendors with the objective of simplifying the design of new policies and features.

3. Architecture

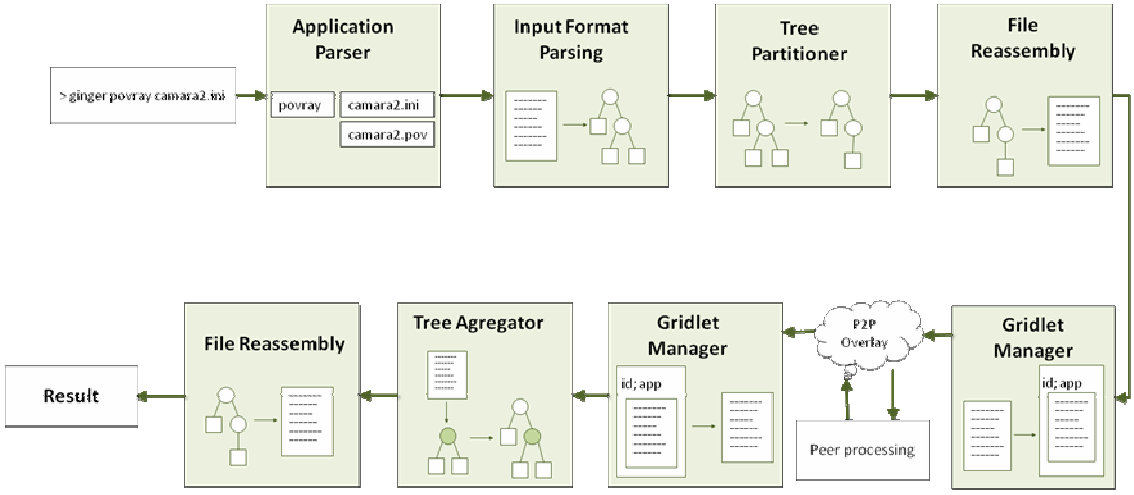

Figure 1. Ginger pipeline

At the moment, Ginger must be explicitly invoked through the command line before the application that is being adapted. Next the application is analyzed in order to discover which are its inputs and outputs what are its arguments and if other requirements are met. All these properties are read from an application descriptor. Knowing what the input file and its format is, the next step consists in parsing the file and constructing an auxiliary tree for subsequent transformation. The parsing is done according to a format descriptor which includes a grammar that accepts any file of this format as well as the operations required to split and merge files of this format.

3.1. Format Analysis

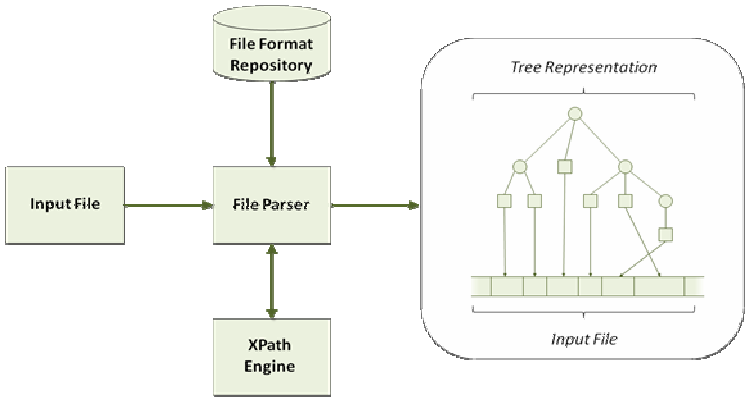

Figure 2. File parsing process

If the grammar recognizes the file we get as result a tree representation in which the leaf nodes are pointers to parts of the file. As an example, suppose we which to recognize a text file which contains a list of people names and locations, each property being stored in a contiguous chunk `<ID, Size, Value>`.

```
FN6JohnnyLN5BravoCY6LisbonCT8Portugal
```

Is an example that respects such a format for a single person. You can see the grammar that recognizes this input string on the left, and the result tree on the right.

```
<grammar>
  <start>
    <zeroOrMore>
      <interleave>
        <element name="name">
          <ref name="chunk" chunk-id="FN"/>
          <ref name="chunk" chunk-id="LN"/>
        </element>
        <element name="location">
          <ref name="chunk" chunk-id="CY"/>
          <ref name="chunk" chunk-id="CT"/>
        </element>
      </interleave>
    </zeroOrMore>
  </start>
  <define name="chunk">
    <block>
      <data name="id" type="string" size="2" equals="${chunk-id}"/>
      <data name="size" type="char"/>
      <data name="value" type="string" size="xpath:preceding-sibling::size"/>
    </block>
  </define>
</grammar>
```

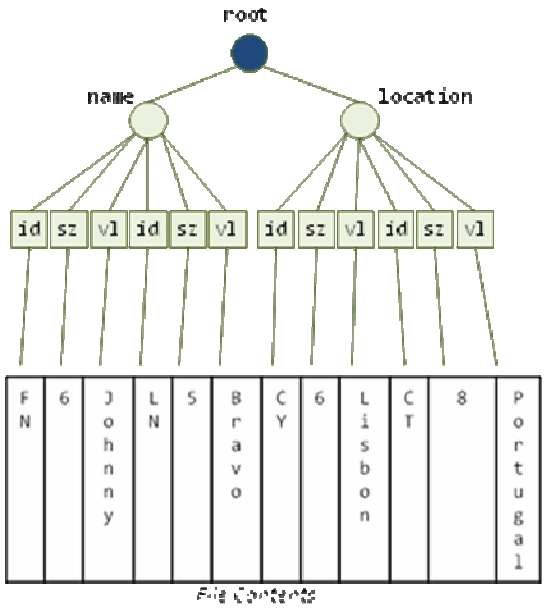

Figure 3. Sample grammar and the tree output for recognized input "`FN6JohnnyLN5BravoCY6LisbonCT8Portugal`"

The interleave pattern allows that the enclosed patterns match in any order and as such we can either have any combination of the 4 properties with a restriction - both ‘last-name’ and ‘country’ must appear after ‘first-name’ and ‘city’ respectively, since they are grouped. Note that the data patterns ‘id’, ’size’, ’value’ are enclosed inside a block pattern. This pattern blocks the interleaving, forcing the enclosed pattern to completely match, or otherwise the block fails. In practice when an ‘id’ is matched against the input, a new processing loop starts running only on that block. This loop ends when either there are no patterns left or there are none that match input. Also note that the tree is being dynamically created and as such we can use XPath at any time to discover the value of a previous token. In the example we use it to find out the size of the ‘value’ token.

The grammars described in this format are of the LL(n) class. In most instances only one token is required to figure out which pattern should be chosen but as it is evident on out previous example the block pattern requires 3 successful tokens matched for it to be accepted. If it fails the token(s) need to be retracted and file pointer reset so a new path can be chosen. Regarding the grammar determinism, by default the parser allows for non-deterministic grammars choosing first the left-most pattern before trying others. It can be configured to look at all the possible paths, stopping when an iteration matches more than one token which is useful for debugging purposes.

3.2. Tree Manipulation

In figure 1, there are two blocks that are used for transforming document trees. In the case of the Partitioner the tree being manipulated is generated from the input file, while at the Aggregator it’s a tree generated from one of the, possibly many, output files. The transformation operations can be found along with the format descriptor and consist in sequences of CRUD operations.

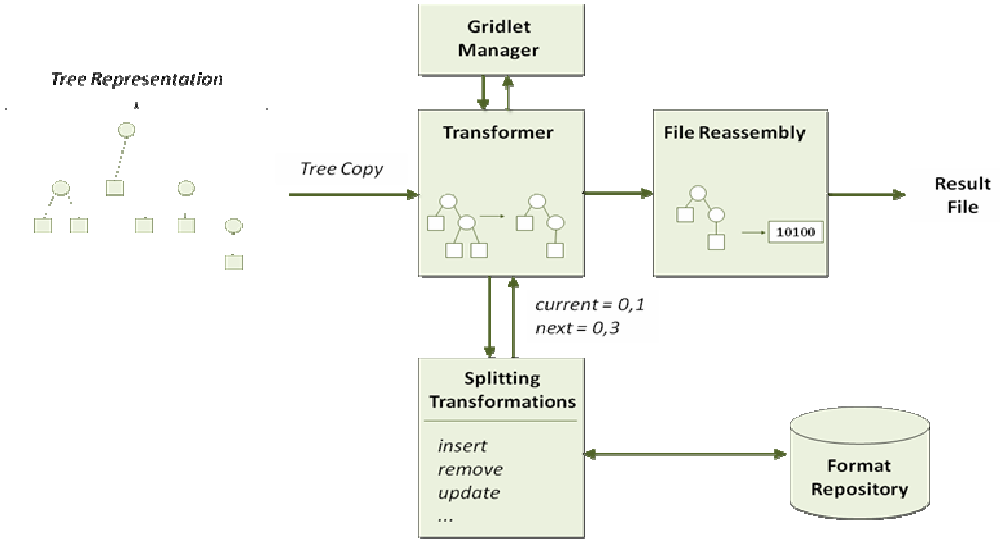

Figure 4. File splitting process

Before executing the splitting operations, the Transformer asks the Gridlet Manager (GM) for a list of empty gridlets that will contain the smaller tasks (computational wise) referring in this request, if required, the maximum jobs that can be created from this particular task. The GM first consults with the application descriptor in order to rate the total cost of the task. The cost is a vector:

<CPU, I/O, UpBW, DownBW>, CPU + I/O = 1; UpBW, DownBW in KB

Knowing this cost, the GM makes a best case scenario choosing the peers which minimize the processing time according to this task cost vector and tries to allocate a time slot for this task on each of them. Whether the peer allows or denies the allocation it returns the maximum time that it is willing to concede to this client. This way if any peer denies the allocation the GM can either partition the whole task again or partition only the job which failed the allocation.

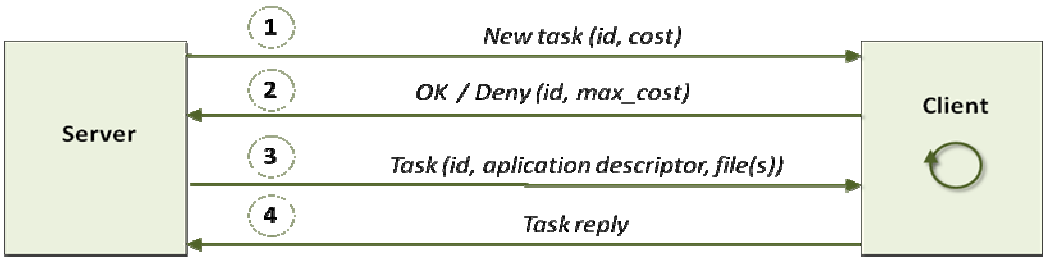

Figure 5. Task allocation/execution protocol

After having the allocation done, the GM creates N empty gridlets where N is the number of jobs created to handle this task, specifying in each one its offset and size relative to the overall task. Then for each gridlet the input tree is copied and transformed according to splitting operations taken from the format descriptor and which are based on the gridlet offset and size. In binary formats there are usually variable-sized blocks, with previous blocks holding the actual size of the next block. To avoid having to write updates every time an add/remove is done on the variable block we allow for a data node to listen for changes on other data nodes or elements in the tree. Whenever there's an update on a listened node the value of the listening node is updated according with an XPath expression.

After being transformed, the tree is serialized to a file and encapsulated inside the respective gridlet, along with its id the application descriptor and other required files and finally sent to the peer that has been allocated to handle this job. After receiving all the replies the process is reverted. The output files are each one converted to a tree representation and sent for the transformer this time to be merged in the final output file. Again the merging operations are taken from the format descriptor which also tells the transformer how the final document should look like before the merging starts. The current options are to start from an empty document, start with the document created from the response to the first request, or from the response that arrived first. For example in the case of AVI since the important headers should be the same on every response, we choose to start the final document tree from the first response that arrives.

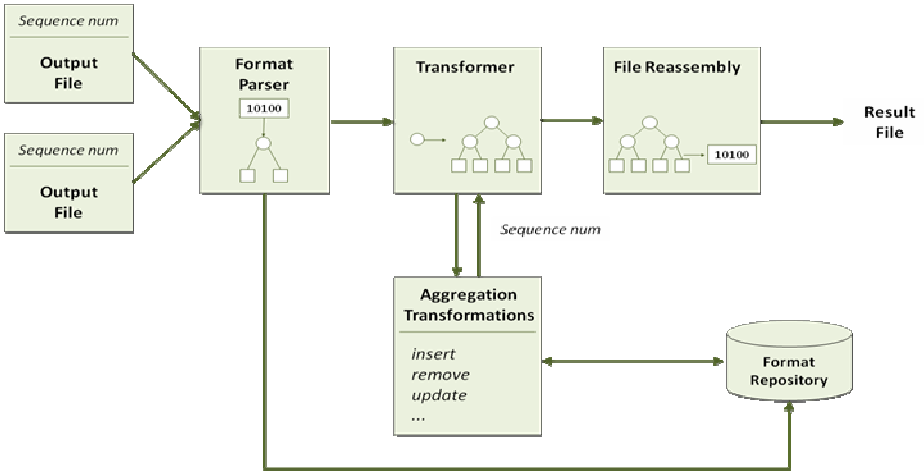

Figure 6. File merging process

With the final document tree completely built it is now simply a matter of serializing/reassembling the tree back into a file form and the process is complete.

4. Results

Two media tasks were chosen to test this work. The first is video compression which is not only a CPU-bound process but also very I/O intensive, accepting and producing usually large inputs/outputs. We will use the AVI format to contain our video and the partitioning is done on keyframes which are independent frames, so we can safely assume that they are good splitting points.

The other is a ray-tracing rendering application (PovRay) which has the characteristic of being mainly a CPU-bound application, taking small scene files as input and producing moderately sized picture files. On PovRay the scene initialization file can define “Subset_Start_Frame” and “Subset_End_Frame” properties to define which frames the ray tracer should render, and the partitioner explores these properties to create the jobs. Next, we show 3 tests that are being done over these applications:

- Compression of a 44.4MB xvid video to the h263+ codec
- Compression of a 7MB h264 video to the h263+ codec
- Ray tracing animation (250 frames, 640x480) following compression in MPEG4 video at 25fps, producing a 10 second movie.

The computers that are being used to execute these applications have the following characteristics:

- Laptop: Intel Core 2 Duo 2.0GHz, 2GB RAM, 7200RPM disk **(main machine**)
- Desktop: AMD 2.2 GHz single-processor, 1GB RAM, Raid-0 configuration **(local network)**
- Sigma server: Dual Opteron 2.4GHz, shared AFS **(internet)**

To get more representative results these teste were done over 5 different configurations with the load distribution varying in each of them, according to the following table:

Configurations	Sigma	Desktop	Laptop
I	1/3	1/3	1/3
II	2/3	1/3	0
III	1/2	1/2	0
IV	1/3	2/3	0
V	1	0	0
Local	0	0	1

Table 1. Load Distribution

Note that in the results for configurations I-IV the request, processing and response times are taken from the gridlet that takes the longest time to produce its result (many of the results would be already available before that moment). Also note that in the configuration V the time for Splitting and Aggregation is 0, since one node takes the whole processing.

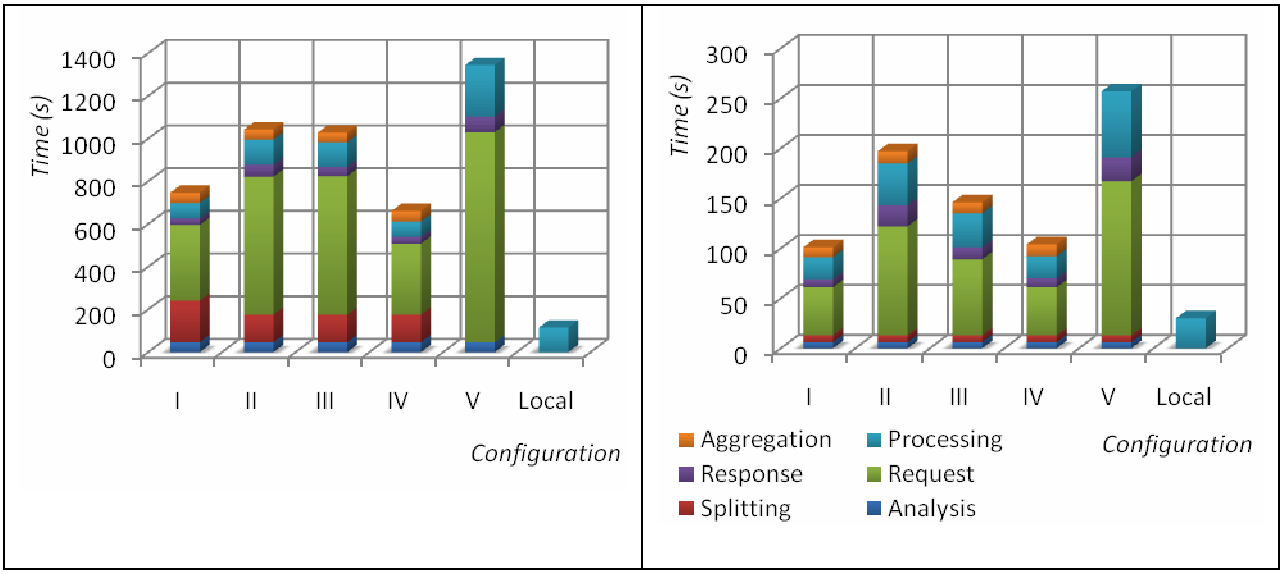

Figure 7. Execution time of video compression (44.4MB file on the left, 7MB file on the right)

The first observation that we can take from this test is that the local execution is always fastest then executing the application remotely. Firstly our transformer is not yet optimized and the every time we wish to make a new partition the source tree needs to be copied so we can apply the transformations over a clean version of this tree. Since the 44.4MB video produces a tree of around 20000 nodes it is a slow process to make a copy, apply the transformations and produce the job file. We can say with some confidence that this is the main problem and not the splitting transformations, since the aggregation step is a lot less noticeable.

The next problem is unavoidable and is related to the low bandwidth upload link from the local machine to the remote peer (Sigma). Topping at around 50KB/s sending the gridlet to this peer is very slow and takes more time than the local execution in every instance. An interesting observation is that Sigma being the most powerful machine shows the slowest times in compressing this video. The reason is

probably related with this server having a networked and shared file server (AFS) which introduces some I/O latency in comparison with a local disk and this reflects especially in I/O intensive applications just like this one. With the 7MB file we can alleviate the strain in the request time and in fact the difference between the best distributed time (configuration 1) and the local execution time narrows by a factor of 2. It's also noticeable the splitting time has a lot less impact since the file has a lot less nodes than in the previous example.

Executing the application in the desktop computer only, appears as a good alternative since both machines are in the same LAN and as such the request/response delays virtually disappear, although this solution being somewhat slower. In fact the fastest machine in the video compressions tests is actually the local machine (laptop).

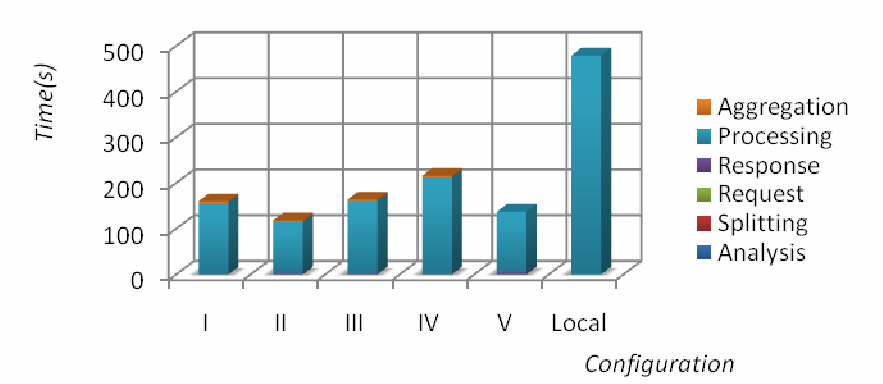

Figure 8. Execution time of ray-tracing rendering

With the raytracing application the results are a lot more satisfying. Since we don't partition the scene but instead only act upon the rendering start/end frames the analysis/splitting takes only some milliseconds. The input (pov, ini) and resulting output files (avi) are very small which reflects in almost negligible request, response and aggregation times. In practice +99% of the execution time is spent rendering the scene frames. Being a CPU-bound application the constraints observed on the Sigma server for the last test disappear, which results in the configuration V being the fastest. The results showed that the local machine is the slowest for this test which means that in the configuration 1, the RTT for the gridlet sent to the remote server and for the remote desktop is lower than the one sent to the local machine. With these results, it's highly suggestible to leverage the power of remote machines to execute ray tracing animations.

5. Conclusions

Open grids are not a new concept, but adapting existing applications without making modifications on them and instead explore the inputs and outputs is indeed a novel approach.

Throughout this paper we described in some detail the several steps through which the application must go through in order to be adapted to the Ginger infrastructure. We also introduced the Gridlet concept as a unit of work and an indicator for selecting the best peers.

The results have shown that distributed video compression is still confined to local area networks due to the I/O cost when transferring work to peers over links with low bandwidth. On the other end of the spectrum ray-tracing rendering has small I/O requirements and so it is highly advantageous to explore distributed computation for this application both in local and remote nodes on the grid.

Future work could include more advanced techniques to integrate applications (APIs, DDE, etc.) on this architecture and integration with existing middleware (like Ourgrid).

6. References

[1] L. Veiga, “GiGi: An Ocean of Gridlets on a "Grid-for-the-Masses",” May. 2007;

[2] D.P. Anderson, BOINC: a system for public-resource computing and storage. *In Proceedings. Fifth IEEE/ACM International Workshop on Grind Computing, 2004.*

[3] N. Andrade et al., “OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing,” *9th Workshop on Job Scheduling Strategies for Parallel Processing*, 2003;

[4] A. Goldchleger et al., “InteGrade: object-oriented Grid middleware leveraging the idle computing power of desktop machines,” *Concurrency and Computation: Practice and Experience*, vol. 16, 2004, pp. 449-459.

[5] A.R. Butt et al., “Java, peer-to-peer, and accountability: building blocks for distributed cycle sharing,” *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, San Jose, California: USENIX Association, 2004, pp. 13-13.

[6] Ralph Ewerth, “Grid Services for Distributed Video Cut Detection,” Dec. 2004;

[7] E. de Lara et al., “Iterative adaptation for mobile clients using existing APIs,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, 2005, pp. 966-981.

[8] K. van der Raadt, “APST-DV: A Practical Framework for Scheduling and Deploying Divisible Loads on Grid Platforms.”

[9] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[10] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. In *Proc. of the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM*, August 1996.

[11] V. Lo et al., Cluster Computing on the ly: P2P Scheduling of Idle Cycles in the *Internet*, 2004;

[12] Plura Project [http://www.pluraprocessing.com]