Transparent Adaptation of e-Science Applications for Parallel and Cycle-Sharing Infrastructures

João Morais   João Nuno Silva   Paulo Ferreira   Luís Veiga
INESC-ID / Technical University of Lisbon, Portugal
joao.c.morais@ist.utl.pt, {joao.n.silva, paulo.ferreira, luis.veiga}@inesc-id.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 users' geographical dispersion is a key factor. However, regular home users willing to take advantage of distributed processing cannot regard this 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 this work, distributed computation, and cycle-sharing in particular, are addressed in a different manner. Users share idle resources with other users provided that such resources (namely, CPU cycles) are mostly employed to execute already installed applications (e.g., popular commodity applications targeting video compression/transcoding, image processing, ray tracing). Users need not to modify an application they already use and trust. Instead, they require only access to an available format description of the application input/output, in order to allow transparent and automatic decomposition of a job in smaller tasks that may be distributed and executed in cycle-sharing machines.

1 Introduction

In this day and age, a quick observation that can be made about personal computers is that they are either inactive or active with a very small load; this means that most CPU cycles (and the energy spent in them) are wasted without any relevant operation being done. Along with the Internet growing, in recent years several applications have appeared that try to leverage this wasted cycles in useful processing of CPU-intensive applications (the most known being the SETI@Home). However most of these projects have a limited scope in how the Grid resources can be used. In most cycle sharing projects users can only share their resources and are not allowed to run their applications. In such cases, users are motivated to give their CPU time when there is monetary compensation involved (Plura) or, still more frequently, the project's purpose is the Humanity's greater good (e.g., Folding@Home, Seti@Home).

An increasing number of home users makes use of commodity (or *off-the-shelf*) applications that are CPU-intensive and whose performance could be improved if executed in a parallelized manner (namely, for distributed execution employing other users' idle CPU cycles). More so, they tend to form communities around portals exchanging applications and techniques, and sometimes data (e.g., 3D models). Examples include applications such as those to render photo-realistic images and animations using ray tracing, video transcoding for format conversion, batch picture processing for photo enhancement, face detection and identification, among others (many of them free or shareware, and widely available and deployed). While most home users will not have access to Grid or cluster infrastructures, they may be able to access some P2P cycle-sharing or utility computing infrastructure (e.g., Amazon EC2).

Independently of the access to parallel execution infrastructures, most of the users are unable to parallelize applications due to one or more of the following reasons: i) applications are distributed in binary form designed for local execution only, ii) unwillingness to execute applications parallelized (i.e., modified) by others than the publishers, and iii) even when source code is available and free, most users lack the necessary coding skills. Thus, to most users, the only available option for parallel execution in distributed scenarios, while employing the unmodified applications users are acquainted to, is through transparent application adaptation.

This work acts upon the aforementioned restrictions: the users either i) do not have access to the application source code, or ii) do not have the knowledge or time to study it in order parallelize it, or iii) do not trust third-parties other than the publishers to modify the application. Since the application will remain unmodified, its adaptation can be performed in only two ways: i) binary code rewriting which actually modifies application code during run-time, and is complex to do for every application (in essence, injecting parallelized code designed previously), or ii) transparent partition of the input data, adaptation of application parameters and input, and finally regrouping and adaptation of the results.

The fraction(s) of input to provide each task with, as well as how the partial outputs are aggregated, is determined by instructing the middleware with XML-based format and application descriptors that indicate how to process, partition and aggregate the files. These descriptors can be written by a user, application provider, derived from format syntax; once made available, they can be reused by anyone else. A great advantage of working with smaller inputs occurs when the original file is large and the tasks are deployed to other nodes connected through lower bandwidth links (as is the case with some home users in P2P scenarios).

The rest of this paper is organized as follows. In the next section, we address work related to ours along with a brief comparative analysis. In Section 3 we describe the middleware architecture to perform application adaptation for execution in cycle-sharing infrastructures, and its main implementation details and results in Section 4. In Section 5, we close the paper with some conclusions and future work.

2 Related work

In the cluster and grid arena, schedulers (such as Condor [1]) are employed to handle the deployment of jobs on the available hosts with different quality of service by minimizing total execution time or taking advantage of processors' idle time. Ourgrid [2] 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 favors that rewards users according to their contribution to the Grid. InteGrade [3] 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. To ease the interaction with grid scheduler there have been specialized developments targeting user interfaces for tasks creation (e.g., Ganga [4]). These tools allow the easy creation of parameter sweep jobs, where each task processes part of a input domain. The partition of parameters intervals, as well as the assignment of different files to different tasks are possible, but the splitting of large data files into smaller data units is impossible (let alone ensuring the semantically coherency of the partitions) . Another approach, APST-DV [5] describes a case study regarding MPEG4 encoding on clusters which has also explores input division techniques, resorting to external programs to do the file splitting/merging. Developing such applications for other data formats and applications would require more programming effort, albeit somewhat redundant, that most users will not be able to develop.

BOINC is a middleware and infrastructure for developing distributed computing applications allowing the execution of lengthy Bag-of-Tasks problems over the internet on donated spare computing cycles. However, users are not allowed to execute their own applications, being restricted solely as cycle donors. In distributed computing environments, the data partition is usually a separate and custom built step. In BOINC, when tasks are being created, the input data must be already partitioned. The project owner is responsible for the development of the code to create the tasks, and for the development of the data splitting code. In nuBOINC [6], users are allowed to submit new tasks that are executed with commodity applications. Users are also able to interactively define the parameters passed to each task. There have been a number of proposed P2P cycle-sharing infrastructures. Most of them address the execution of complete jobs at a single node and do not attempt to perform or otherwise automatic input data splitting. In CCOF [7], 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. The earlier work described in [8] introduces the notion of gridlet as a semantics-aware unit of workload division and computational offload. In this paper, the authors describe a complete architecture able to parallelize lengthy jobs that execute commodity applications, along with its implementation and performance evaluation.

3 Middleware Architecture

In this section we describe the main aspects of the proposed middleware architecture, named Ginger. We address: i) the general network and system architecture, ii) application adaptation with file partition and aggregation, and iii) task creation.

Network and System Architecture: Figure 1 depicts a global view of the network and system architecture of Ginger, using a layered approach, featuring applications, gridlets (i.e., tasks, inputs and results), and the networked cycle-sharing infrastructure providing computational resources. At the top, there are unmodified commodity CPU-intensive applications executed (usually locally) by users. Pictured examples portray image and video rendering and processing, as well as applications usually employed in Grid scenarios to perform calculus, simulation and modeling related to chemistry,

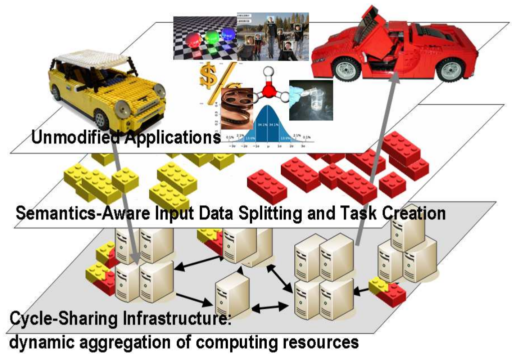
Unmodified Applications
Semantics-Aware Input Data Splitting and Task Creation
Cycle-Sharing Infrastructure:
dynamic aggregation of computing resources

Fig. 1. Ginger network and system architecture

biology, economics and statistics. The two LEGO cars represent the input and output of an hypothetical CPU-intensive application that processes data in a complex format (e.g., MPEG4 transcoding).

In the middle layer, the LEGO blocks colored yellow represent parallel tasks, each with its associated partition of the input data. These tasks are automatically created by the middleware after determining and analyzing the application being invoked (e.g., by its command line), its parameters, and the input file(s) provided. Each task is carried out by executing the intended application at one of the nodes of the cycle-sharing infrastructure. This way, parallelism is extracted not by analyzing application code (assumed to be binary and opaque) but rather by identifying sections or blocks of the input file(s) that could be processed independently, and therefore, in parallel manner. Nonetheless, since the applications are not modified, the partition of the input data fed to each task must also be adapted by the middleware in order to appear as a properly formatted input file (this involves header analysis and structure manipulation).

The lower layer illustrates an example cycle-sharing infrastructure where nodes communicate among themselves to discover available resources and installed applications. Each node receiving a gridlet, executes the intended unmodified application locally (possibly over a virtual machine or embedded in a virtual appliance) with the transformed input partition assigned to its task, and also with possibly adapted parameters. The middle layer also depicts LEGO blocks colored red that represent the results of the execution of parallel tasks. These results need to be aggregated by the middleware into a complete output file according to format description and application semantics (this involves header reconstruction and structure manipulation) and provided to the user. The result file should have no relevant differences (w.r.t. application semantics) from one that would have been produced by a local, serial execution. LEGO blocks representing gridlets have different sizes in order to depict tasks with different costs (CPU, bandwidth, memory) associated to their execution (both estimated and determined after execution). Such costs estimates and measurements can be used to drive resource discovery and a possible reputation/accountancy mechanism.

Application Adaptation: At the moment, Ginger must be explicitly invoked through the command line before the application (e.g., `ffmpeg`) that is being adapted (in practice, this can easily be circumvented by using a customized command shell). Next the application (its name) is analyzed in order to discover which are its inputs and outputs, and what are its arguments and if other requirements are met. These properties are read from a XML application descriptor (described in [9] due to space limitations). This avoids rewriting custom tools for each application, improving middleware extensibility.

The next step consists in parsing the file and constructing an auxiliary tree for subsequent transformation. The parsing is done according to a XML format descriptor which includes a grammar that accepts any file of this format, as well as the operations required to properly split (partition) and merge (aggregate) files of this format. This includes all the necessary header analysis and reconstruction, patching and structural modifications (e.g., moving blocks across the file).

Tree Manipulation: During *input partitioning*, the tree being manipulated is generated from the input file, while at *result aggregation*, the tree is generated starting 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 (create, insert, update and delete tree nodes before, after, or between specified elements or tokens). After being transformed, the tree is serialized to 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 task. After receiving all the replies the process is reverted. The output files are each one converted to a tree representation and sent to a *transformer*, to be aggregated in the final output file.

Again, the merging (aggregation) operations are taken from the format descriptor which also tells the transformer how the final document should look like before the merging starts. Available options include: i) start from an empty output file, ii) start with the result created from the response to the first request, or iii) start with the result created from the response that arrived first. For example, in the case of an AVI with an MPEG movie, since the important headers should be the same on every response, we choose to start the final result tree from the first response that arrives. With the final result tree completely built, it is now simply a matter of serializing/reassembling the tree back into a file form and the file manipulation process is complete. This is done incrementally on disk by the middleware. There is no need to hold all the output files on memory simultaneously in order to aggregate them. Only the result tree is preferably maintained in memory for better performance.

Task Creation: 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) including in this request, if required, the maximum number of tasks that can be created from this particular job. The GM first consults with the application descriptor in order to rate the total cost of each task. The cost is a vector:
$< CPU, I/O, UpBW, DownBW >$ with $CPU + I/O = 1$; 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. The lower-level resource discovery

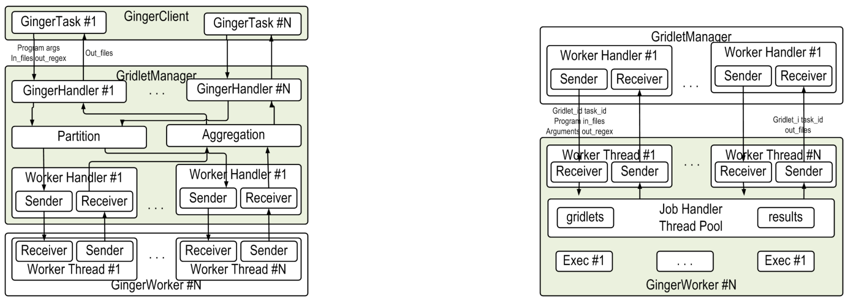

Fig. 2. Gridlet Manager

mechanism informing the GM of suitable peers with available resources and the desired applications/virtual appliances set up is out of the scope of this paper.

After having the allocation done, the GM creates N empty gridlets where N is the number of tasks created to handle this job, specifying in each one its offset and size relative to the overall job. Then, for each gridlet, the input tree is copied and the transformer runs the splitting operations over this tree, referring the current and next block variables so that the splitter knows what nodes to add/remove/change. 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 and, whenever there is an update on the node, the value of the listening node is automatically updated according with an XPath expression.

4 Implementation Issues and Evaluation

Figure 2 illustrates the implementation of the main components of Ginger (implemented in Java). The Saxon XPath engine was used to read and process XML configuration files. Communication between remote components uses both Java RMI (for gridlet exchange over LAN or to virtual appliances running in utility computing infrastructures) and FreePastry (for gridlet exchange over peer-to-peer cycle-sharing overlay).

The **Ginger Client** interacts with the user, allowing him to specify the file to be processed, the application to be used, and its parameters. **Ginger Workers** execute on the remote computers and, after receiving all the information about the tasks to be executed (file to be processed and task parameters), execute the intended application. The **Receiver** and **Sender** modules are responsible for communication with the **Gridlet Manager**. These modules interact with their corresponding **Worker Handler**. The **Job Handler Thread Pool** guarantees that, depending on the number of available CPU/cores available, the optimal number of concurrent applications is executed and that all pending gridlets are executed.

The **Gridlet Manager** receives the information about the jobs to be executed, and creates the necessary gridlets, storing the information about each job on the corresponding GingerHandler. The partitioning is performed by the **Partition** module taking into account the input files and the descriptors in a XML file. This module creates a tree representing the file, and taking into account the corresponding XML file partition descriptor, splits the original tree in several coherent branches. For each produced tree, a gridlet input file is created. After receiving the result, the **Worker Handler** feeds the resulting file to the **Aggregation** module to create the complete result file.

Evaluation: As a preliminary evaluation (further results in [9]), we address video transcoding job (using the `ffmpeg` application) which is not only a CPU-bound process but also relatively I/O intensive, accepting and producing usually large inputs/outputs. We use the AVI format (a complex format with an intricate internal structure) to contain our video and the partitioning is done on keyframes which are independent frames (i.e., not represented as differences from previous or successor frames), so we can safely assume that they are good splitting points. We performed the following 2 tests: i) Transcoding of a 44.4MB xvid video to the h263+ codec, and ii) Transcoding of a 7MB h264 video to the h263+ codec. To exemplify the computers that would serve as peers providing resources to execute jobs, both in a cycle-sharing infrastructure as well as on a Grid scenario, we executed the applications on machines with the following characteristics: I) Laptop: Intel Core 2 Duo 2.0GHz, 2GB RAM, 7200RPM disk (main user machine), II) Desktop: AMD 2.2 GHz single-processor, 1GB RAM (Local Network), III) VM allocated in Department cluster (Sigma): Dual Opteron 2.4GHz, AFS (Internet).

Regarding the qualitative evaluation of the Ginger middleware architecture, the main result is that the validity and generality of the approach used have been confirmed. In fact, users are able to execute unmodified applications as they are used to, and they are transparently adapted by the middleware that is able to decompose the submitted job in several gridlets to be executed in remote nodes. The applications executed are oblivious of the fact that they are being adapted in order to process only a fraction of the work to carry out the submitted job (see [9]).

To get representative results these tests were done over 5 different configurations with gridlet load distribution (Sigma, Desktop, Laptop) varying in each of them, as follows: 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), and V) (0, 0, 1). The results are shown in Figure 3. Note that in the results for configurations I-IV, the overall processing and response times are taken from the gridlet that takes the longest time to produce its result and that may depend on network latency (many of the results would be already available before that moment and the file could have been previewed in a media player). Our transformer is not yet optimized; partitioning and tree creation have delay. Additional latency is caused by gridlet transfer to remote

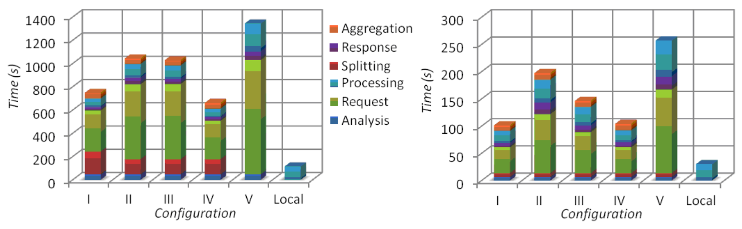

Fig. 3. Execution time of video compression (44.4MB file on the left, 7MB file on the right)

peers, and AFS versus a local disk. Nonetheless, users are able to transparently leverage available remote cycles to execute several jobs in parallel.

5 Conclusions

Open grids are not a new concept, but adapting existing applications without requiring modifying them and, instead, explore the opportunity to perform adaptation on the inputs and outputs, is indeed a novel approach. In this paper we propose a new middleware architecture (Ginger), able to parallelize the execution of unmodified commodity CPU-intensive applications (e.g., video compression/transcoding, image processing, ray tracing) on distributed cycle-sharing scenarios. Thus, users need not to modify an application they already use and trust. Applications are transparently adapted via middleware driven by format descriptions of the application input/output. Adaptation deals solely with input and output data formats, as well as application parameters. Thus, popular commodity applications, such as `ffmpeg` can be transparently adapted to execute several tasks in parallel over a distributed environment, allowing users to execute jobs remotely, in parallel fashion, without the need to modify applications' source or binary code.

References

1. Thain, D., Tannenbaum, T., Livny, M.: Condor and the grid. In Berman, F., Fox, G., Hey, T., eds.: Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons Inc. (December 2002)
2. Andrade, N., Cirne, W., Brasileiro, F., Roisenberg, P.: OurGrid: An approach to easily assemble grids with equitable resource sharing. In: 9th Workshop on Job Scheduling Strategies for Parallel Processing. (2003)
3. De Camargo, R.Y., Kon, F.: Design and implementation of a middleware for data storage in opportunistic grids. In: CC-GRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, IEEE Computer Society (2007) 23–30
4. Egede, U., K.Harrison, Jones, R., Maier, A., Moscicki, J., Patrick, G., Soroko, A., Tan, C.: Ganga user interface for job definition and management. In: Proc. Fourth International Workshop on Frontier Science: New Frontiers in Subnuclear Physics, Italy, Laboratori Nazionali di Frascati (September 2005)
5. van der Raadt, K., Yang, Y., Casanova, H.: Practical Divisible Load Scheduling on Grid Platforms with APST-DV. Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International (2005) 29b–29b
6. Silva, J., Veiga, L., Ferreira, P.: nuboinc: Boinc extensions for community cycle sharing. In: Self-Adaptive and Self-Organizing Systems Workshops, 2008. SASOW 2008. Second IEEE International Conference on. (Oct. 2008) 248–253
7. Zhou, D., Lo, V.: Cluster computing on the fly: Resource discovery in a cycle sharing peer-to-peer system. In: IEEE International Symposium on Cluster Computing and the Grid. (2004)
8. Veiga, L., Rodrigues, R., Ferreira, P.: Gigi: An ocean of gridlets on a "grid-for-the-masses". In: CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, IEEE Computer Society (2007) 783–788
9. Morais, J., Silva, J., Ferreira, P., Veiga, L.: Transparent adaptation of e-science applications for parallel and cycle-sharing infrastructures, inesc-id tech. report 15/2011 (February 2011)