Transparent usage of grids for data parallelism

Daniel Millot, Christian Parrot, Eric Renault

Transparent Usage of Grids for Data Parallelism

Daniel Millot, Christian Parrot and Éric Renault
GET / INT
Computer Science Department
Évry — France

Abstract

This paper describes a software production chain dedicated to end-users which, given a sequential program to be applied to a set of individual data, enables efficient parallel processing of huge collections of data on a grid. This system requires no involvement of the user and produces fairly good results.

1 Introduction

Frequently the same computation is made independently over the elements of a given collection of data, inducing data parallelism [11]. As grids [9, 4] are getting more and more common, it is therefore important to be able to run applications involving data parallelism on such target platforms. In order to try and do so, one has the choice between either design a parallel application, which may be a hard task reserved to specialists –unless user-friendly tools are used– or simply write a sequential program, relying on automatic tools to produce the corresponding parallel code. However, this technique is mainly dedicated to symmetric multiprocessors or vector processors, and can only be used on clusters through software-based distributed shared memory systems [2, 14]. As so many computing resources appear to be available by means of grids, we focus on easing their utilization rather than on making full use of the available computing power.

One can consider any software component as consuming data from a data stream, and producing results into a results stream. Therefore, in the design of a sequential implementation of such a component, these two streams must be respectively sliced into and generated from data structures (namely *data* and *result*). Thus, the datatypes *data* and *result* have necesseraly to be specified when writing the sequential corresponding code. As we do not need anything else, we do not require any extra effort from the user.

So, let's consider a sequential software component which continuously reads *data*, calls some function *work(in data, out result)* on each item and writes the corresponding *result* as shown in Fig. 1. The

Figure 1: Generic sequential software component.

data stream is fed by means of a data source (file system, database, sensor network, network connection, etc), and results are consumed by a consumer process. Such a software component could be part of a pipeline involving a whole chain of successive processing steps. If one needs to increase the flow through the pipeline, this component is a potential bottleneck; thus, it is a good candidate for parallel execution. Due to data parallelism, one can expect a near linear speed-up; so increasing the number of processors could allow to increase the flow, possibly leading to another bottleneck upstream or downstream. Additionnally, introducing some buffering between pipeline stages can help absorbe flow fluctuations and make communication and computation overlap. But in order to be useful to end-users, the parallel execution should be obtained by means of a seamless replacement of sequential software components by automatically-generated equivalent parallel components.

In this paper, we propose a software production chain : AIPE (**A**utomatic **I**ntegration for **P**arallel **E**xecution) to deal with a large variety of applications. AIPE enables end-users to get parallel execution of an application on grids without writing anything more than its sequential code.

Our method has been successfully used to automate the processing of a huge collection of spectra for identification of semi-nucleotide polymorphisms in a database of DNA samples. It can be used either for mass production of results or for rapid prototyping. Indeed, it allows the developer to concentrate on details of algorithms without dealing with anything else

than sequential programming. Moreover, the developer can get quick feedback when modifying an algorithm, as he can achieve significant results by considering important samples without incuring dissuasive delays. Therefore the development phase of the application can also benefit from parallelism, with no specific programming effort, as we provide a transparent component substitution, not a programming tool.

This paper is organized as follows. The next section gives an overview of environments helping users have their applications execute on a grid. Section 3 presents the software architecture we designed to allow efficient parallel execution. Section 4 focuses on how AIPE makes its usage almost transparent to an end-user. Section 5 presents performance figures. Section 6 compares our new approach to related research works, outlines future work and concludes the paper.

2 Related works

Among the environments which have been developed in order to do part of the job for the end-user who needs to design parallel applications, some are devoted to parameter sweep applications, such as Nimrod-G [1] or ILab [13], whereas others could be used in more general settings (AppLeS [5], TOP-C [7] or Ninf-G [12]). Parameter sweep can indeed be considered a particular form of data parallelism, as it iterates identical computations on all the elements of a set of numerical parameters, and each of the individual computations of a parameter sweep application can be run independently of the others, as a different process. Therefore, a user can create a corresponding set of ad hoc scripts and schedule their execution on one or more processors.

Nimrod-G [1] is a grid version of a tool designed for distributed parametric modeling, which has been enhanced from clusters to grids. Built over the Globus [8] grid middleware, it is dedicated to parametric studies and provides a graphical user interface to help the user specify the parameter spaces of his application and the computation to be done. Nimrod-G can then prepare the corresponding independent jobs, control their execution and manage the results. On user request, Nimrod-G queries the MDS [15] component of Globus to locate resources, then selects them according to some criteria including both tasks deadlines and cost usage depending on some economic model. To our knowledge, no other type of applications can benefit from Nimrod-G.

In the same way as Nimrod-G, ILab [13] is dedicated to the execution of parameter studies on grids. It also provides a graphical user interface, to make the creation of input files easier, and builds automatically a script for each run. Different methods allow the execution of these scripts, either via "rcp" and "rsh" commands or through a job scheduler.

The AppLeS [5] (**App**lication **L**evel **S**cheduling) environment, as its name suggests, focuses mainly on scheduling. When dealing with an application, an AppLeS agent goes through the following steps: resource discovery, resource selection, schedule generation, schedule selection, application execution with iterative schedule adaptation in the case of long-running applications. The main drawback is that the original application has to be a parallel or a distributed one, and it must be modified in order to be scheduled by an AppLeS agent. Therefore, templates have been proposed in order to make things easier for the end-user, at least for the two classical paradigms of parameter sweep and Master-Worker; using such templates, the user merely has to provide an XML description of his tasks and AppLeS automatically generates an application it can schedule. While AppLeS was extended from networks of workstations to grids, its scheduling techniques have been progressively adapted to deal with the heterogeneity of grids.

TOP-C [7] (**T**ask **O**riented **P**arallel **C**/C++) has been designed to help the end-user parallelize a sequential application. It uses a Master-Worker programming model and provides a high level API which has been implemented on various architectures, from shared memory model to grids based on the Globus protocols. The intent is to keep the number of lines that have to be modified in the source code as small as possible. Although the available primitives are higher level than MPI ones, the user is required to write a parallel application.

Ninf-G [12], provides a running environment for GridRPC, a grid implementation of the RPC protocol. Among the applications for which GridRPC has proven to be effective, one is parameter sweep which is well-suited for execution by multiple servers on a grid. However, it is obvious that in order to use Ninf-G, an application has to be re-written using the GridRPC API.

On the one hand, these environments make the use of grids easier. They sometimes allow non trivial parallelism, whereas the tool presented in this paper does not, and their main focus is on scheduling, so as to achieve the best usage of resources.

On the other hand, they either require a significant expertise in parallel and distributed programming (or at least some involvement of the user to describe the conditions for parallel execution), or they are ex-

plicitely limited to parameter sweep applications and cannot deal with more sophisticated datatypes. Furthermore, the different runs entail the repeated creation of processes, together with repeated transfer of executable code files –either through NFS sharing [5] or through the use of some grid component [12] such as the GASS [6] component of Globus– in order to get a copy of the executable code prior execution of each individual task. These transfers can result in costly software interactions which might be an overhead to be considered, especially for short-running individual experiments. On the contrary, as is shown in the next section, we set up a software architecture for the whole duration of the execution and do not consider repeated file copy or transfer. As this software architecture uses neither NFS or NIS, it is well suited for the grid.

3 Software architecture

We based our software architecture on the Master-Worker paradigm, which makes scheduling easier thanks to central control. The Worker is application specific, whereas the Master is kept as generic as possible. As shown in Fig. 2, the Master is split into

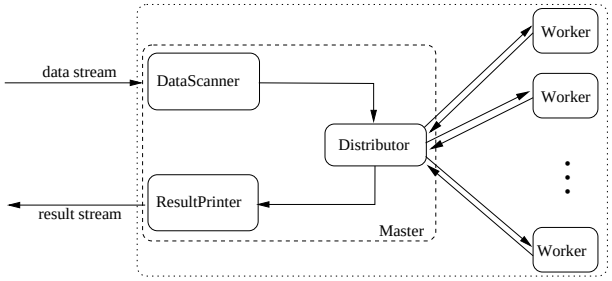

Figure 2: Software architecture.

three processes : processes DataScanner and ResultPrinter ensure the interface with both streams, using respectively *data* and *result* formats corresponding to user-defined datatypes, whereas the third process, Distributor, interacts with the pool of Workers executing the function *work(data, result)*. Precisely, the Distributor process maintains a list of free Workers. When a data item is available, one of the free Workers is chosen according to the current scheduling policy. Then, a message containing the data to be processed is sent to the chosen Worker. Different scheduling policies are possible, each aiming at a particular criterion about resource usage or execution time. When a result message is available, Distributor communicates the result to ResultPrinter and puts the corresponding Worker back into the list of free Workers. If results are not available within some delay, it is possible to re-schedule the corresponding work on a different Worker, making the solution more fault-tolerant.

In order to absorb speed variations in both DataScanner and ResultPrinter, buffers are used to store data and result items; this helps keep Workers busy. So as to avoid extra copies inside the Master, all three parts of the Master process are run on the same host (note that this also allows communication through shared memory, as shown in Fig. 3). Therefore, when a data item is read on the data

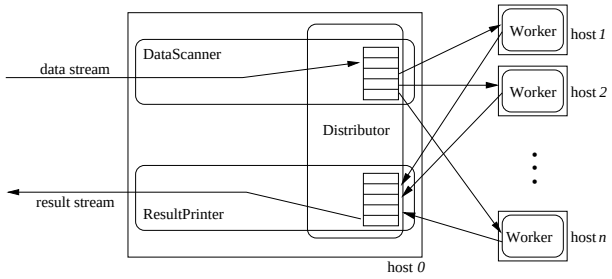

Figure 3: Parallel implementation.

stream, it is written only once in user space before it is sent to a Worker; similarly reasoning applies to results. Besides avoiding useless copies, this makes synchronization between Distributor and the interface processes slightly looser as they just compete shortly to update indexes to shared memory slots. A greater number of slots gives more laxity which is useful in case of large fluctuations in the data stream or in the processing time of items.

With no information about fluctuations magnitude, the number of slots is assumed to be larger enough so as to avoid Workers from waiting. Due to possible variations in their processing time, the order of incoming data might be different from the order of the corresponding results delivered by the parallel component. Therefore, process DataScanner attaches identifiers to data items throughout the Master; if the order is significant for the rest of the application, these identifiers could possibly be delivered together with the results by slightly adapting the result stream. In case the Master becomes a bottleneck when increasing the number of Workers, several Masters could be used, as suggested by [3].

As the software architecture of AIPE is based on the Master-Worker model, it is not sensitive to machine heterogeneity. Workers are launched only once, giving a chance to absorb the start up overhead. On the contrary, most existing environments consider the different tasks as independent jobs, which means that

each of them has to be launched independently. Therefore the penalty incured for finer grain should be less important when using our system.

Communication between process Distributor and Workers rely on the MPI recognized standard which allows *immediate sends* and *receives*; this avoids useless waitings, especially for Distributor. For implementation over a grid, we chose to use Globus which offers all the required services, such as authentication, information service giving criteria about resources for scheduling policies and so on, together with the mpich-G2 implementation of MPI. Moreover, MPI allows the definition of dynamic datatypes, which helps keep the Master generic. Indeed, generic messages between Distributor and Workers are possible without any user assistance, as long as the software production chain is able to generate automatically the appropriate MPI functions, defining MPI datatypes from both *data* and *result* datatypes provided by the user. As MPI programming style requires knowledge and expertise, we relieve the end-user from the burden of writing and debugging any MPI code, as shown in the next section.

4 Transparent integration

The Distributor and Workers have to exchange MPI messages for both data items read from the data stream and results to be written to the result stream. Thus, the generic prototypes for MPI messages have to be adapted to include user-defined data and results which depend upon the application. Regarding the organization of AIPE, three pieces of information must be provided by the end-user: datatypes for both the data stream and the result stream, together with the function to execute on the Worker. This function takes two arguments which types are those of the data stream and the result stream respectively.

As we intend our piece of software to be of transparent use for the end-user, it is important that any kind of data could be transferred from the Distributor to Workers. Moreover, we do not want the end-user to be concerned with our different structures. In order to do so, a specific compiler has been developed to convert C datatypes to MPI datatypes automatically.

Works have been done [10] already to produce MPI datatypes from C datatypes. However, these solutions expect the developer to introduce specific comments in program sources so as to indicate which type to generate. In the present case, we expect the user to give a specific name to structures used to transfer data from the distributor to Workers and vice-versa. Thus, in our solution, no specific comment is required. This is important as it allows existing applications to be executed using our solution. In this case, the user only needs to provide an alias to datatypes used for both data and result streams.

Almost every datatype can be translated from C to MPI automatically. At present, two cases (pointers and unions) lead to problems as they have no equivalence in MPI. For the first case, regarding the fact that a pointer is the address of another variable inside the virtual address space of the process in which it has been declared, pointers shall not be transferred from one process to another one as their meaning is limited to the process they belong to. Thus, the use of pointers in data structures to be translated results in an error. If, for any reason, this would be interesting for an application, pointers could be translated either to an `MPI_INT` or an array of `MPI_BYTE`. For the second case, one has to remind that a `union` in C represents various ways to apprehend the same memory area. Considering that the generated MPI program may run on an heterogeneous platform, it is not possible to determin at compilation which representation should be used for communication. Thus, we left the transformation of `union` structures for the future and their compilation results in an error.

The automatic generation of MPI datatypes from C datatypes has been made possible by introducing extra steps in the usual compilation chain as shown in Fig. 4 on the next page. Typically, in order to make sure the definition of a given type is complete, the first step in the compilation chain (`cpp` for the preprocessor) is executed. As the generated file is to be compiled by the effective C compiler (`cc`), it must contain all the information required by any type in the file (if not, this file could not be compiled properly by `cc`). Thus, instead of feeding `cc` with the intermediate file generated by `cpp`, this intermediate file is transformed by `mpipp` (a specific parser we wrote) to produce a new C file in which the definition of MPI datatypes have been automatically appended to the original file. This new source file is then provided to `cpp`. The intermediate file which results is then processed by the usual compilation chain.

The list of C datatypes from the original file, to be transformed to MPI datatypes, is provided on the command line when invoking `mpipp`. At present, this list is hard coded and matches the requirements of our software suite. However, options are scheduled to allow more flexibility and reuse by other pieces of software.

The end-user may not be aware of all this as it is hidden by the compiler suite provided for AIPE. In fact, in order to compile his application, the devel-

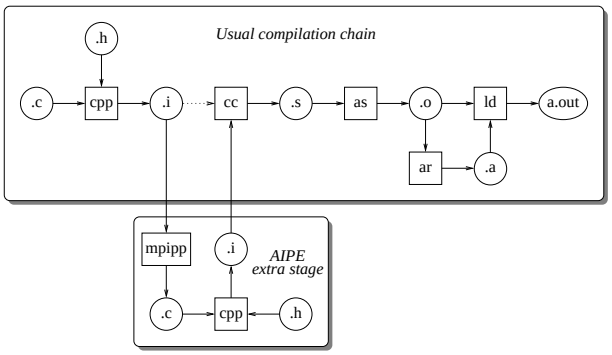

Figure 4: The compilation chain with AIPE.

oper simply uses the `aipecc` compiler which is merely a wrapper around the GNU C compiler in which two extra stages have been added for the automatic transformation of C datatypes to MPI datatypes.

5 Performance measurements

The sequential component chosen for the evaluation of the software production chain iterates the exponentiation of square matrices of integers read on a stream. So the initial component includes three functions: the function *work(in data, out result)* which makes the computation, the function that reads a matrix on the standard input and stores it in a *data* structure, and the function that writes the contents of a *result* structure on the standard output. The *data* datatype is

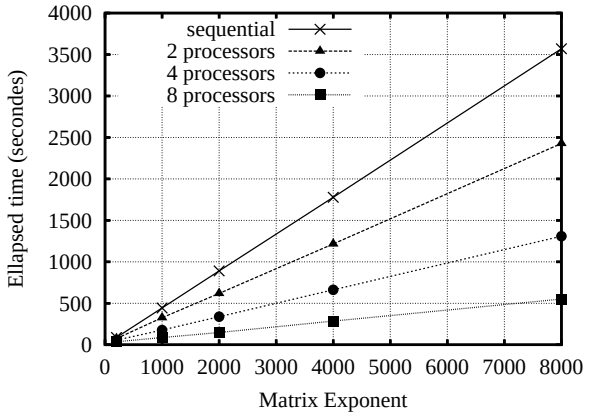

Figure 5: Ellapsed time.

a structure including the contents of the matrix and the exponent value, whereas the *result* datatype is a structure including only the contents of the result matrix. From these elements, the parallel code was generated automatically in one step, without further effort. The technique has been successfully applied to various example applications involving more sophisticated datatypes.

The platform used for performance measurements is composed of eight Pentium IV running at 2.8 GHz with 512 MB of RAM memory. There is no NFS file sharing among these computers, and no NIS users database. Globus 2.4.3 is used to map the user on a local account on each computer.

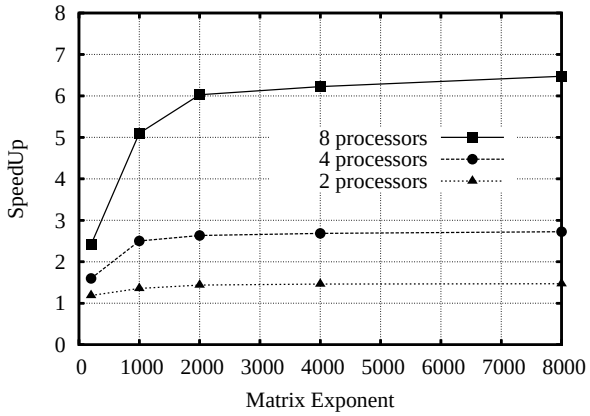

Figure 6: Speed-up.

In order to avoid hardware optimization effects in the computation, odd integers are used for matrix elements and the matrix size, as the use of even integers would lead to zeroes through overflows for large numbers of iterations, when increasing the value of the exponent. The size chosen for the matrices is 101, and values for exponents are taken in the range from 200 to 8000. Fig. 5 shows that the ellapsed time varies linearly when the exponent value increases. In fact, in the program we have chosen as an example, the higher the exponent of the matrix, the larger the amount of computation; however, the amount of communication remains the same as the size of the result matrix is constant. Thus, this curve shows that our method can distribute computations efficiently over the network. Fig. 6 presents the speed up versus the exponent used to evaluate the result matrices. It highlights that very good speed-up can be provided (an efficiency of up to 80% is achieved for eight processors) even when the amount of computation is rather low (e.g. an efficiency of 75% is achieved on eight processors for an exponent of 2000 in which case the computation time for a single matrix is 8.9 seconds).

6 Conclusion

In this paper, we have presented the software production chain AIPE. Given any sequential sofware component, it enables the end-user to obtain a parallel execution on a grid and exploit the data parallelism involved in his application. Using AIPE, the end-user does not have to make any parallel programming effort to adapt the application in order to benefit from the system. Experiments have confirmed that the parallel execution can be efficient.

Yet, up to now, the choice of the free Worker by the Distributor process has been kept very simple, more sophisticated scheduling must be considered in the future. The centralized control of the Master-Worker paradigm makes the integration of any scheduling strategy in Distributor quite straightforward. The user might even be given the opportunity to decide the trade off between performance and resource usage that best suits his needs.

Apart from scheduling, other works are in progress to improve the scalability of the software architecture. For instance on the performance side, in case the Master becomes a bottleneck when increasing the number of Workers, several Masters could be used. Fault-tolerance is a direction for future work, together with portability through the use of a virtual machine.

Acknowledgements

A collaboration between the Centre National de Génotypage (CNG) and the Institut National des Télécommunications (INT) stands at the inception of the AIPE project. Therefore, we would like to acknowledge the staff of the CNG bioinformatics group. Especially, we wish to thank Nino Margetic, Dominique Van Den Broeck and Fernando Dos Santos for their support.

References

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid ? pages 520–528. IPDPS'2000, Mexico, 2000.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, R. R. H. Lu, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] C. Banino. Scalability limitations of the master-worker paradigm for grid computing. PARA'04 Workshop on state-of-the-art in scientific computing, Copenhagen, Denmark, June 20-23 2004.

[4] F. Berman, G. Fox, and T. Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, March 2003.

[5] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Computing*, 14(4):369–382, Apr. 2003.

[6] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. Gass: A data movement and access service for wide area computing systems. Sixth Workshop on I/O in Parallel and Distributed Systems, May 5 1999.

[7] G. Cooperman, H. Casanova, J. Hayes, and T. Witzel. Using TOP-C and AMPIC to port large parallel applications to the computational grid. pages 109–116. Cluster Computing and the Grid 2nd IEEE/ACM International Symposium (CCGRID'02), Berlin, Germany, May 21 - 24 2002.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[10] D. Goujon, M. Michel, J. Peeters, and J. Devaney. AutoMap and AutoLink: Tools for communicating complex and dynamic data-structures using MPI. In D. Panda and C. Stunkel, editors, *Network-based Parallel Computing. Communication, Architecture, and Applications*, volume 1362 of *Lecture Notes in Computer Science*, pages 98–109, Las Vegas, NV, January 1998. Springer-Verlag.

[11] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, Inc. New York, NY, USA, 1998.

[12] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing*, 1:41–51, 2003.

[13] M. Yarrow, K. M. McCann, R. Biswas, and R. F. V. der Wijngaart. An advanced user interface approach for complex parameter study process specification on the information power grid. In *GRID*, pages 146–157, 2000.

[14] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice & Experience*, 9(11), 1997.

[15] X. Zhang and J. Schopf. Performance analysis of the globus toolkit monitoring and discovery service, MDS2. Proceedings of the International Workshop on Middleware Performance (MP 2004), part of the 23rd International Performance Computing and Communications Workshop (IPCCC 2004), April 2004.