A Steering Environment for Online Parallel Visualization of Legacy Parallel Simulations

Aurélien Esnard, Nicolas Richart and Olivier Coulaud

***Abstract*—In the context of scientific computing, the computational steering consists in the coupling of numerical simulations with 3D visualization systems through the network. This allows scientists to monitor online the intermediate results of their computations in a more interactive way than the batch mode, and allows them to modify the simulation parameters on-the-fly. While most of existing computational steering environments support parallel simulations, they are often limited to sequential visualization systems. This may lead to an important bottleneck and increased rendering time. To achieve the required performance for online visualization, we have designed the EPSN framework, a computational steering environment that enables to interconnect legacy parallel simulations with parallel visualization systems. For this, we have introduced a redistribution algorithm for unstructured data, that is well adapted to the context of $M \times N$ computational steering. Then, we focus on the design of our parallel viewer and present some experimental results obtained with a particle-based simulation in astrophysics.**

***Index Terms*—Computational Steering, Numerical Simulation, Data Redistribution, Parallel Visualization.**

I. Introduction

In most research fields, 3D scientific visualization plays a central role in the analysis of data generated by numerical simulations. Nowadays, simulations are typically running in batch mode on supercomputers, and the analysis of the results is then performed on a local workstation as a post-processing step, which implies to preliminary collect all the simulation output files. In this approach, the lack of control over the in-progress computations might drastically decrease the profitability of the computational resources (repeated tests with different input files separated by excessively long waiting periods). Computational steering is an alternative approach to the typical simulation work-flow of performing computation and visualization sequentially. It mainly consists in coupling a remote simulation with a graphics system through the network in order to provide scientists with online visualization and interactive steering. Online visualization appears very useful to monitor the evolution of the simulation by rendering the current results. It also allows us to validate the simulation codes and to detect conceptual or programming errors before the completion of a long-running application. Interactive steering allows the researcher to change the parameters of the simulation without stopping it. As the online visualization provides an immediate visual feedback on the effect of a parameter change, the scientist gains additional insight in the simulation, regarding to the cause-effect relationship. Such a tool might help the scientist to better grasp the complexity of the underlying models and to drive more rapidly the simulation *into the right direction*.

Authors are members of the ScAlApplix Project at INRIA Futurs and LaBRI (UMR CNRS 5800, University of Bordeaux I). 351, cours de la Libération, 33405 Talence, France. E-mails: {esnard,richart,coulaud}@labri.fr.

Because the datasets produced by simulations can be very large and complex (multivariate, multi-scale, multidimensional), their visualization can be almost as computationally demanding as the simulation itself, typically with iso-surface computation, particle rendering, volume rendering, *etc.*. In such a context, parallel visualization and rendering techniques can provide the necessary level of performance required for the online visualization. In order to accommodate larger datasets and higher image resolutions, the use of a PC cluster equipped with 3D accelerated graphics cards, called *graphics cluster*, is an attractive approach – both in terms of cost and performance. Parallel rendering techniques have already demonstrated their scalability for viewing very large datasets on such graphics clusters. These solutions are typically based on the sort-last algorithm that combines the images produced independently by each node [1]. Compared to other parallel rendering strategies (i.e. sort-first or sort-middle), and regardless of hardware mechanisms, the sort-last algorithm scales better for very large datasets, but is limited by the display resolution. To overcome this issue, Sandia National Laboratories have recently proposed in [2] a solution based on multiple tile displays that scales appropriately.

Even though most of existing computational steering environments, such as CUMULVS [3], DAQV [4] or gViz [5] support parallel simulations, they are limited to sequential visualization systems. This leads to an important bottleneck and increased rendering time. In the gViz project, the IRIS Explorer visualization system has been extended to run the different modules (simulation, visualization, rendering) in a distributed fashion on the Grid, but the visualization and the rendering modules are still sequential. Recent works in the Uintah PSE (Problem Solving Environment) [6] has addressed the problem of massively parallel computation connected to a remote parallel visualization module, but this latter module is only running on a shared-memory machine. Therefore, it would be particularly valuable for the scientist if a steering environment would be able to perform parallel visualization using a PC-based graphics cluster. This is precisely the purpose of the EPSN environment that enables to interconnect parallel simulations with visualization systems, that can be parallel as well [7]. While the EPSN environment is mainly focused on the

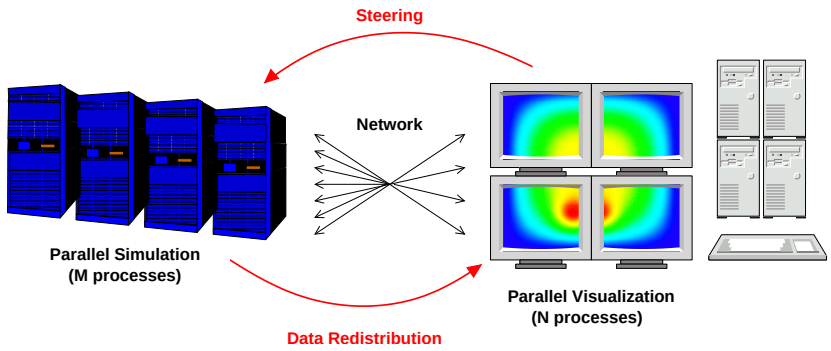

Fig. 1. Overview of the EPSN infrastructure.

M×N code coupling issue, including problems of parallel data redistribution and time-coherent transfers, this paper presents more precisely the design of a parallel visualization system based on the EPSN framework.

In the next section, we describe the architecture of EPSN and the redistribution algorithm we use for unstructured data such as particles or meshes. In section III, we detail our solution to efficiently perform online parallel visualization with EPSN. Finally, in section IV, we validate our approach with some experimental results obtained with a particle-based simulation in astrophysics.

II. The EPSN Framework

EPSN is a distributed computational steering environment which allows the steering of remote parallel simulations with user interfaces, which include sequential or parallel visualization tools [7]. This environment is based on a coupling framework between parallel simulation components (running on M processes) and parallel visualization components (running on N processes). The visualization application is viewed in EPSN as a client program that can dynamically connect and disconnect from the simulation during its execution. Once a client is connected, it interacts with the simulation component through an asynchronous and concurrent request system. We distinguish three kinds of steering request. Firstly, the "control" requests (play, step, stop) allow to steer the execution flow of the simulation. Secondly, the "data access" requests (get, put) allow to read/write parameters and data from the memory of the remote simulation. Finally, the "action" requests enable to invoke user-defined routines in the simulation.

A. General Principles

In order to make a legacy simulation steerable, the end-user annotates its simulation source-code with the EPSN API. These annotations provide the EPSN environment with two kinds of information: the description of the program structure according to a Hierarchical Task Model (HTM) and the description of the distributed data that will be accessible by the remote clients. Thanks to a logical date system associated to the HTM, one can precisely follow the parallel work-flow of the simulation. Moreover, this date system enables EPSN to efficiently coordinate the treatment of steering requests in parallel, and to ensure the time-coherence of parallel treatments (see [8] for more details). Indeed, the data distributed over parallel processes must be accessed carefully to ensure they are presented to the visualization system in a meaningful way. This simply means that each piece of data collected from the simulation and transmitted to the parallel visualization system must refer to the same logical date. In order to ensure such a coherence, our strategy consists in scheduling on-the-fly the next common date where all the simulation processes can start to transfer data in parallel. This strategy is efficient for it does not imply to synchronize the simulation processes.

Once a simulation has been instrumented with the back-end API, it can be launched as usually. The client locates its simulation on the network using a naming service and then connects to it through the proxy. As regards the development of client applications, we also provide a front-end API that enables to integrate EPSN in a high-level visualization system such as *AVS/Express*. However, it is often more convenient to start a steering session with our own user interface, called *Simone*. Thanks to this lightweight program, the end-user can easily connect any simulations and interact with them, by controlling the computational flow, viewing the current parameters or data on a simple data-sheet and modifying them optionally. *Simone* also includes simple visualization plug-ins to online display the intermediate results, as shown in figure 2. Moreover, the EPSN framework offers the ability to exploit parallel visualization techniques, as we will see in the section III.

B. Architecture

As shown in figure 3, each parallel component in EPSN is made up of one *proxy* and several *ports*. A port consists in a thread attached to each application node (simulation or visualization), which runs a server waiting for client requests. The proxy provides a unified access point to the parallel component, that dispatches the client requests to all nodes. These requests are forwarded to all local ports which are in charge to perform the steering treatments in parallel. For instance, when a request to get

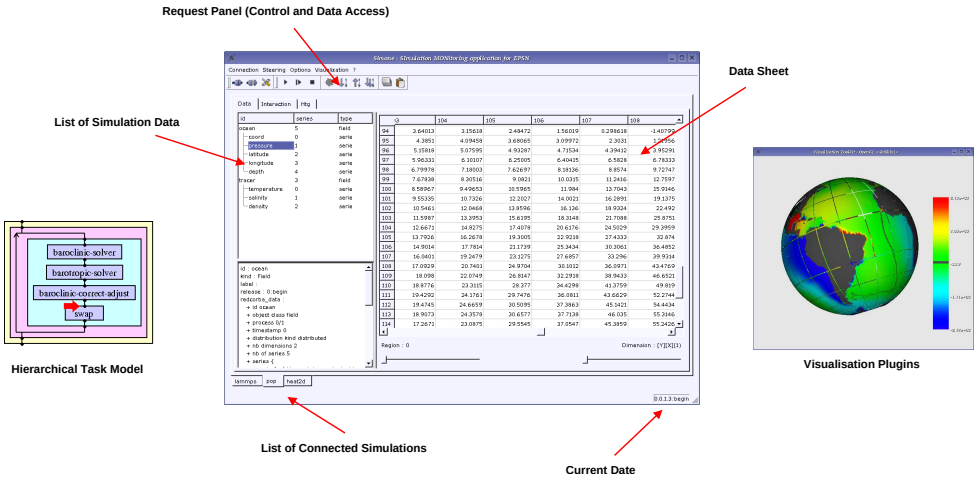

Fig. 2. Simone (Simulation Monitoring Interface for EPSN) connected to the Parallel Ocean Program (POP) of the Los Alamos National Laboratory.

data is received by a simulation port, the EPSN thread accesses the process memory and directly transfers data to the remote component ports. This transfer occurs concurrently to the simulation execution, during the task of the HTM where the data are said to be accessible for remote clients. The use of an access area rather than a simple access point is an important feature of the EPSN architecture that enables to (partially) overlap the steering communication and simulation computation. As both the simulation and the visualization can be parallel applications, EPSN is based on the M×N redistribution library called RedGRID [9]. This library is firstly in charge to compute all the messages that will be exchanged between the two parallel components, and secondly to perform the data transfer in parallel. Thus, RedGRID is able to aggregate the bandwidth and to achieve high performance. Moreover, it is designed to consider a wide variety of distributed data structures usually found in the numerical simulations, such as structured grids, particle sets or unstructured meshes. Both RedGRID and EPSN use an internal communication infrastructure based on CORBA [10] which provides the interoperability between the components running on different architectures. For further details about the core of the EPSN architecture, the reader can refer to [11] and [8].

C. Redistribution Layer

Basically, the redistribution problem is decomposed into four steps: the description of distributed data, the message generation, the message scheduling and the communication stage. The description of distributed data typically involves to specify how the data are spread through the processes and how the data are stored in the memory of each process. Thanks to these descriptions, the redistribution algorithm computes the *communication matrix* which stores all the messages that are exchanged between processes. At the communication stage, each process sends several messages, either to all other processes, or just to a subset of these. Such parallel communication flows make it possible to aggregate the bandwidth.

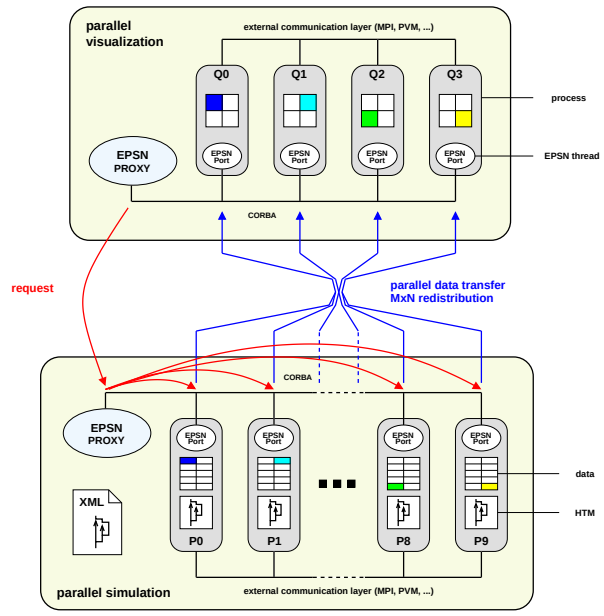

Fig. 3. Architecture of the EPSN environment.

The development of a standard solution to the problem of the redistribution requires to define a model of data description which is standard as well. In this context, we have introduced a data model built upon the notion of *complex object*. A complex object is made up of several variables (e.g. position, velocity, *etc.*) and divided into several element subsets or regions that are distributed over processes.

This section presents a redistribution algorithm that is well adapted to the context of $M \times N$ computational steering. In this case, the data distribution on the visualization code is not initially defined. This offers the opportunity for the redistribution layer to choose it at run-time in "the best way". In this algorithm, we assume that the object elements are equally distributed

over the M simulation processes. Our strategy consists in the placement of the simulation elements to the N visualization processes. In order to equilibrate the number of elements on the visualization code and to minimize the number of messages, we use a *split and merge* strategy. As shown in figure 4, this algorithm simply generates $M + N - GCD(M, N)$ messages, that result from the intersection of the two distribution patterns.

We have implemented this algorithm in the RedGRID library for two kinds of complex objects: particle sets and unstructured meshes. For particles, the split and merge operators are quite trivial, while for unstructured meshes, they are defined thanks to graph partitioning techniques as those provided by Metis [12].

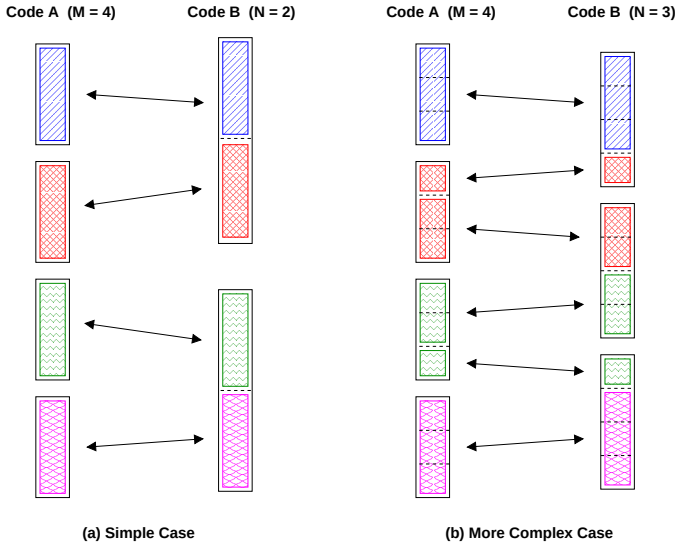

Fig. 4. Redistribution with placement strategy in two cases.

III. Online Parallel Visualization with EPSN

This section presents the design of the parallel viewer prototype that we have developed to efficiently produce online visualization on a PC-based graphics cluster. As shown in figure 3, this parallel viewer is a standard EPSN client that can connect to any kind of simulations previously annotated with our back-end API. Basically, the EPSN infrastructure continuously extracts new data generated by the simulation and sends them to the parallel viewer. Then the viewer produces an image that represents the current state of the remote simulation, and so on. Even if the visualization engine is designed according to the SPMD programming paradigm, the parallel viewer has a master node, where the EPSN proxy is located. This node manages both the visualization interactive loop (mouse, timer, *etc.*) and the EPSN request system. This viewer has been designed to be modular and generic: it dynamically retrieves the remote simulation description, including the data distribution, and creates a visualization pipeline adapted to the type of data that is considered (i.e. structured grids, particle sets or unstructured meshes). This task is realized by the *EPSN source*, a particular source that receives data from the network rather than to read them from a file as it is usually done by classical visualization programs. In the following section, we introduce some useful notions for the understanding of the parallel viewer design.

A. Preliminaries

As described in the literature [13], the pipeline of a visualization system is classically based on a data-flow model. This pipeline is based on four kinds of modules – the source, the filter(s), the mapper and the renderer – that are connected as shown on figure 5. The source module typically reads data from a file and produces an output which is then used by the filters. The filter modules can be interconnected in a complex network which processes the input data to generate the desired result. This result is then converted into graphics primitives by the mapper. Finally, the renderer module transforms those graphics primitives into visual images that are interpretable by the end-user.

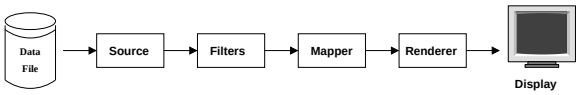

Fig. 5. The different modules in the classical visualization pipeline.

In a data-parallel visualization system, the visualization pipeline is fully replicated on each node of the graphics cluster and the data are typically distributed according to a spatial constraint. The use of a parallel rendering algorithm makes it possible to combine the capabilities of several graphics nodes to produce an image, even if it is displayed on a single screen. In the sort-last algorithm, each node produces a full-resolution image, although processing only a piece of the distributed data. The composite engine is in charge of gathering all these images on a single node and to compare the depth-values to correctly combine the pixel color information (Fig. 6). More recently, K. Moreland *et al.* have proposed a sort-last based parallel algorithm that scales appropriately for high-resolution tiled display wall [2].

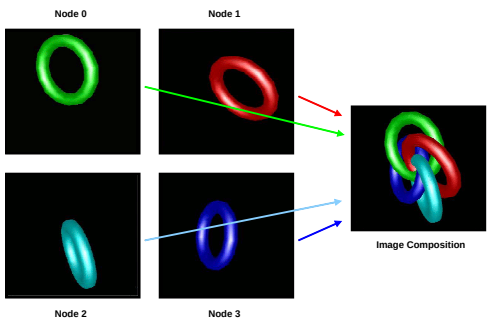

Fig. 6. Principle of the sort-last strategy for parallel rendering on a single screen.

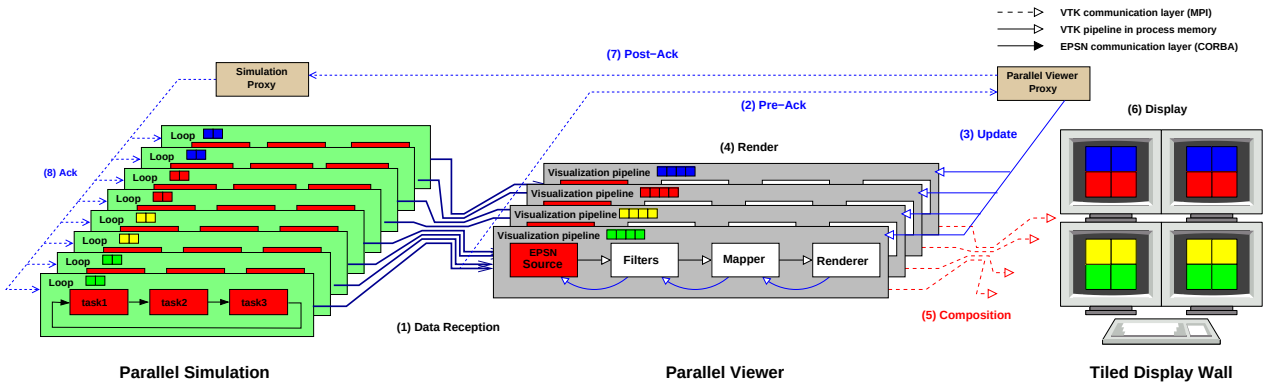

Fig. 7. The different steps required to perform online parallel visualization with the EPSN framework: from the data reception to the image composition on a 2×2 tiled display wall.

B. From the Data Reception to the Image Update

We detail now the different steps that are required to perform online parallel visualization: from the reception of data by the EPSN source to the update of the image displayed on multiple screens. These steps are represented on figure 7.

B.1 Data Reception

Before to start the visualization pipeline, one must first extract the data from the remote simulation. This consists in requesting the EPSN infrastructure to send the data to the parallel viewer whenever a new data release is produced by the simulation. Thus, the parallel viewer periodically receives new data through the EPSN source (step 1). The data distribution on the visualization side is automatically chosen by the viewer program according to the redistribution strategy described in section C. This choice depends both on the simulation data distribution, and on the number of nodes used in the graphics cluster.

B.2 Pipeline Update Request

In a demand-driven pipeline, the request for updating the visualization pipeline occurs from the end of the pipeline. This raises some difficulties, because the renderer module is not aware when new data has been received by the EPSN source. To overcome this, we introduce a timer in the interactive loop of the master process. It enables the EPSN framework to periodically ask if new data are received. In this case, the EPSN proxy will explicitly request to update the parallel visualization pipeline (step 3). In order to indicate when the data transfer is finished, a pre-acknowledgment must be preliminary sent by each viewer node to the EPSN proxy on the master node (step 2). When the proxy has received all these acknowledgments, we assume the parallel data transfer is globally achieved. Thus, at the next timer event, the proxy will be able to request the full pipeline update.

B.3 Parallel Rendering and Image Composition

Once the image update has been requested on step 3, the visualization pipeline is fully updated in parallel from all the sources to all the renderer modules that produce partial images of the scene (step 4). Afterward, we perform the composition of all the partial rendered images thanks to the sort-last algorithm (step 5). This requires to read back the rendered images from the graphics card memory of each node and to communicate with other nodes in order to generate the final image (step 6). The image composition can be either performed onto a single node and display, or can use a more complex composite engine for tiles, as shown in figure 7.

B.4 Request Acknowledgment

The EPSN infrastructure requires the simulation nodes to be acknowledged. In fact, the data transfer are performed asynchronously within the EPSN threads, and so one must acknowledge the simulation that the parallel data transfer is achieved, otherwise the simulation can remain blocked waiting for this post-acknowledgment (step 7). The purpose of this acknowledgment mechanism is to ensure the data transfer is coherent. Moreover, it is particularly useful to regulate the data transfer and to slow down the simulation that produces new data faster than the viewer can render them. To alleviate both the network load and the viewer load, it is possible to send and render the data at a given period by repetitively skipping some steps. As this acknowledgment mechanism can induce an important time overhead, an optimization consists in acknowledging the simulation earlier, just after that the transfer is done and before the render occurs. With this approach, it is possible to overlap the visualization pipeline update during all the next simulation step.

IV. Results

All the results presented in this section have been performed on two clusters: one computational cluster for the simulation, and one "old" graphics cluster for the visualization. The computational cluster is composed of 50 bi-Opterons (2.2 GHz) connected by both a Myrinet/MX high-speed network and a Giga-bit Ethernet network. The graphics cluster is composed of 4 bi-Xeons (2.8 GHz) equipped with Nvidia GeForce 4 graphics cards (Ti 4800, AGP 4× with 128 MB of memory) and

connected by a Giga-bit Ethernet switch. These two clusters are interconnected by a single Giga-bit Ethernet link.

The EPSN framework is written in C++ and uses *omniORB4* [14], an high performance implementation of CORBA. As regards the parallel viewer, our prototype is built over the VTK library [15], that enables parallel visualization according to the SPMD paradigm and parallel rendering with the sort-last algorithm (on a single screen). In order to perform parallel rendering on tile displays, we use another library, called Ice-T (Image Composition Engine for Tiles [16]), which is developed by the Sandia National Laboratories and is well-integrated in the VTK framework.

A. Performance of the EPSN Framework

In order to evaluate the performance of the EPSN framework, we examine in this section two different experiments.

A.1 Overlapping of the Steering Overhead

The first experiment intends to evaluate the capability of the EPSN platform to overlap the steering communication and simulation computation, and thus to reduce the overhead due to the data transfer for online visualization. In this experiment we consider a simple sequential simulation that is composed of a single computational loop divided in two sub-tasks named A and B. At each simulation step the data we want to extract are updated by the simulation during the first task, A, and so no data transfer can occur at this moment. Then the data become accessible for remote clients during all the second task, B. Let T_A and T_B be respectively the time of the tasks A and B, the time of a simulation step is defined by $T = T_A + T_B$ and set at 100 ms. We measure the mean time T^m of a simulation step when data are transmitted by EPSN to a remote sequential client and we compare this time to the time T of a simulation step without EPSN. This experiment is realized for different amount of data and different *overlapping ratio* defined by $r = T_B/(T_A + T_B)$. As the measured time T^m of a step includes both the simulation computation, the data transfer to the client, and the acknowledgment of this transfer to the simulation, it must be greater than T. However, EPSN can alleviate this overhead thanks to its overlapping strategy. Figure 8 shows that we obtain a good overlapping of the communication by the simulation computation when we increase the access area T_B and as long as the amount of data to transfer is not to high. As both the simulation and the client program are sequential, no redistribution is performed in this experiment. It simply demonstrates the ability of EPSN to reduce the steering overhead $T^m - T$ by using the largest access area available in the source-code, rather than using a sending point like in classical steering environments.

A.2 Redistribution and Parallel Data Flow

When both the simulation and the visualization are parallel programs, EPSN uses a redistribution algorithm

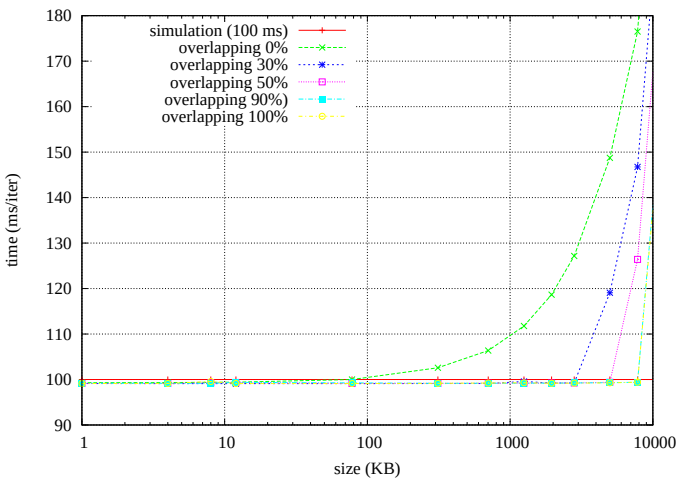

Fig. 8. Overlapping of communication by simulation computation in EPSN.

to generate all the messages that must be exchanged between the coupled codes. In the following experiment, we evaluate the performance of the data transfer from a parallel simulation of M processes to a parallel client of N processes, both running on the same cluster. As EPSN performs parallel data transfer between codes, it makes it possible to aggregate the bandwidth and to achieve high performance. We consider a test case with particle data equally distributed between the M simulation processes. We use the placement strategy called *split and merge*, presented in section C. This experiment is performed for different amount of data, and different $M \times N$ configurations.

Figure 9 shows that the bandwith is nearly 97 MB/s in the 1×1 configuration. Indeed, we almost reach the best performance that OmniORB4 allows on giga-ethernet network, except that EPSN performs a copy on the receiving node. For more complicated $M \times N$ cases, the bandwidth is well aggregated, but is obviously limited by the number of receipting nodes (N). Indeed, the aggregate bandwidth for the case 16×4 is almost as good as for the case 4×4, except that there are more shorter messages in the first case. In our experiment, the case 8×7 is the one which requires to split particles on sending. So we have a more complex communication pattern in this case. However, the performance are as good as in the case 8×8 because each process almost reaches its maximum throughput rate, i.e. ~ 97 MB/s per process.

B. Case Study: The Gadget2 Cosmological Simulation

This section presents some results obtained with a simulation in astrophysics, called Gadget2 [17]. It is a parallel simulation written in C and using the Message Passing Interface (MPI). Gadget2 is a legacy code publicly available on [18], that can be used to address a wide variety of astrophysically interesting problems, ranging from colliding and merging galaxies, to the formation of large-scale structure in the Universe. It follows the evolution of a self-

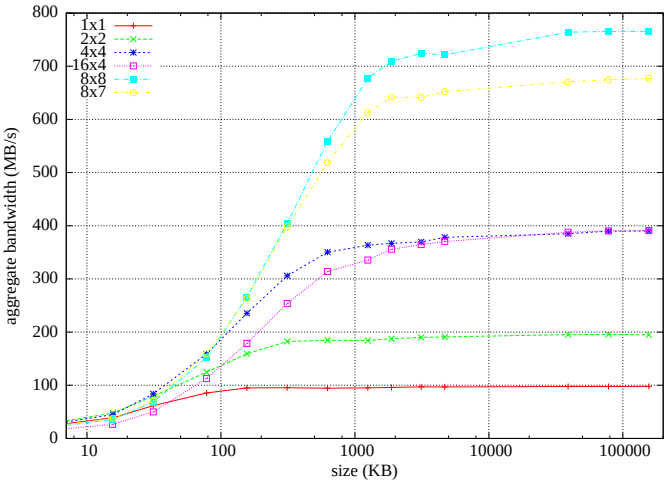

Fig. 9. Parallel data redistribution in EPSN for different $M \times N$ configurations.

gravitating collision-less N-body system. In our case study, Gadget2 simulates the birth of a galaxy, which is represented by a gas cloud that collapses gravitationally until a central shock. The gas cloud is modeled by 1,000,000 particles distributed on 60 processes (30 nodes of the computational cluster) according to a domain decomposition method based on the Peano-Hilbert curve (See [17] for more details).

The simulation is represented in the hierarchical task model of EPSN as a simple computational loop that is composed of 3 sub-tasks: (1) the displacement of particles, (2) the domain decomposition, (3) the computation of the acceleration (Fig. 10). During the last task, we compute the forces to move the particles and the interesting data for visualization, such as positions and velocities, are not modified. The "domain decomposition" task is in charge to dynamically balance the work-load for each domain during the galaxy evolution, that leads to frequent particle migrations between simulation processes.

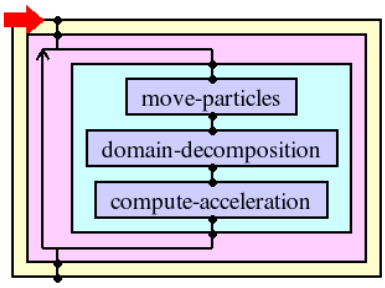

Fig. 10. Representation of the Gadget2 simulation in the Hierarchical Task Model of EPSN.

As shown in the first line of table I, one step of the original simulation, without EPSN annotations, takes an average time of 2150 ms. The "acceleration" task takes 1770 ms, which roughly represents 80% of the whole iteration time; so we have a large access area to extract these data safely and to overlap the communication. The integration of Gadget2 in the EPSN framework requires to annotate its source-code. This instrumentation is entirely negligible when there is no client connected. Indeed, we have about 10 μs for each instrumentation point that are used in the source-code to delimit the tasks. The figure 12 shows that only few lines are required by EPSN and RedGRID to integrate this simulation compared to the 20,000 lines that represents the whole source-code of Gadget2. The description of the HTM and the configuration of data access is given by an auxiliary XML file, called *gadget2.xml* (Fig. 13).

Here, astrophysicists want to visualize the evolution of the galaxy in 3D (Fig.11). It requires to extract both the particle position and the velocity, i.e. 23,438 kB of single-precision floating-point data to transfer at each simulation step. The particles are represented by a point sprite technique. Here, point sprites are simple 2D objects that represent an image of a sphere. This visualization technique considerably increases the efficiency of the rendering, compared to the classical glyph technique, that represents particles by polygonal spheres.

In the sequential case ($N = 1$, $S = 1$), the visualization time takes roughly 1160 ms and the transfer time is about 140 ms. It should lead to a huge overhead of 1300 ms, which is 60% of the time of one simulation step. However, the real overhead is only 21% because EPSN has partially overlapped it. In the parallel case ($N = 4$), the visualization time is roughly divided by 2, that leads to a very small overhead, less than 2% of a simulation step, because EPSN has fully overlapped it. As we use a sort-last algorithm for the image composition, the parallel rendering time is mainly limited by the global resolution of the tiled display wall. At the same global resolution, the number of screens ($S = 1$ versus 4) does not have too much influence on the visualization time. On the other hand, when the global resolution increases, the visualization time increases as well and can induce an important simulation overhead. As we can see on the last line of table I, here we have an overhead of 24% of a simulation step for the global resolution 3200×2400. One track to overcome this issue is to adjust the network bandwidth of the graphics cluster to the maximal resolution we want to reach.

Finally, this experiment validates our approach for computational steering on a parallel legacy code and demonstrates the interest of parallel visualization and rendering techniques to reduce the steering overhead.

V. Conclusion

In this paper, we have presented a modern approach for computational steering that can benefit from parallel rendering techniques and takes advantage of redistribution algorithm. We have detailed the architecture of EPSN and the design of our parallel viewer prototype. This viewer enables efficient online visualization on a PC-based graphics

M	N	S	Global Resolution	Transfert Time	Visualization Time	Simulation Time		
						Accel.	Total	Overhead
60	–	–	–	–	–	1770	2150	–
60	1	1	1600×1200	140	1160	2200	2600	+21%
60	4	1	1600×1200	91	620	1800	2180	+1.4%
60	4	4	1600×1200	91	500	1775	2155	+0.2%
60	4	4	3200×2400	90	1300	2260	2670	+24%

TABLE I

GADGET2 AVERAGE TIME FOR SIMULATION COMPUTATIONS, DATA TRANSFER AND VISUALIZATION (IN MS/ITERATION), OBTAINED AFTER 60 STEPS. M = NUMBER OF SIMULATION PROCESSES; N = NUMBER OF VISUALIZATION PROCESSES; S = NUMBER OF SCREENS.

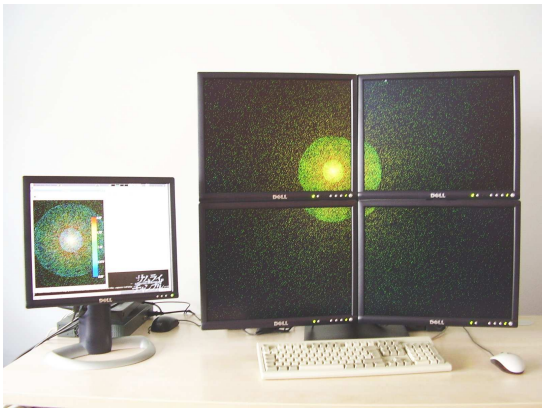

Fig. 11. Online parallel visualization of the Gadget2 simulation on a tiled display wall composed of 4 screens.

cluster. The solution we have retained is based on The Visualization Toolkit (VTK [15]) and uses the Ice-T library developed by the Sandia National Laboratories to perform scalable parallel rendering on tile displays. Finally, we have validated our approach with a particle-based simulation in astrophysics. In future works, we intend to integrate our approach in a high-level visualization application like ParaView [19] that supports parallel rendering as well.

ACKNOWLEDGMENTS

This project has been supported by the ACI-GRID program from the french Ministry of Research (grant number PPL02-03). The authors acknowledge V. Springel from the Max-Plank Institute of Astrophysics for the simulation code Gadget2, publicly available on [18]. In addition, the authors would like to thank J. Biddiscombe and J. Favre from the Swiss National Supercomputing Centre (CSCS) for their code on sprite point.

REFERENCES

[1] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994.

[2] K. Moreland, B. Wylie, and C. Pavlakos. Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays. In *PVG '01: Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 85–92. IEEE Press, 2001.

[3] J. A. Kohl and P. M. Papadopoulos. CUMULVS: Providing fault-tolerance, Visualization, and Steering of Parallel Applications. *Int. J. of Supercomputer Applications and High Performance Computing*, pages 224–235, 1997.

[4] S. Hackstadt, C. Harrop, and A. Malony. A Framework for Interacting with Distributed Programs and Data. In *HPDC*, pages 206–214, 1998.

[5] J. Wood, K. Brodlie, and J. Walton. gViz – Visualization and Steering for the Grid. In *Proceedings of e-Science All Hands Meeting*, Nottingham, September 2003.

[6] J. Davison de St. Germain, S. G. Parker, J. McCorquodale, and C. R. Johnson. Uintah: A Massively Parallel Problem Solving Environment. In *Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 33–42, 2000.

[7] EPSN. A Computational Steering Environment for Distributed Numerical Simulations. http://www.labri.fr/epsn.

[8] Aurélien Esnard, Michael Dussère, and Olivier Coulaud. A time-coherent model for the steering of parallel simulations. In *Euro-Par 2004 Parallel Processing*, number 3149 in Lecture Notes in Computer Science, pages 90–97. Springer Verlag, 2004.

[9] RedGRID. Parallel Data Redistribution Library. http://www.labri.fr/epsn/redgrid.html.

[10] OMG: Object Management Group. Common Object Request Broker Architecture Specification. http://www.corba.org.

[11] Olivier Coulaud, Michael Dussère, and Aurélien Esnard. Toward a computational steering environment based on corba. In G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, volume 13 of *Advances in Parallel Computing*, pages 151–158. Elsevier, 2004.

[12] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical Report 95-035, University of Minnesota, june 1995.

[13] R.B. Haber and D.A. McNabb. Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. *IEEE Computer Society Press*, pages 74–93, 1990.

[14] OmniORB. http://omniorb.sourceforge.net.

[15] W. Schroeder, K. Martin, and B. Lorensen. *The Visualisation ToolKit*. Kitware, 2002.

[16] K. Moreland and D. Thompson. From cluster to wall with VTK. *IEEE symposium on Parallel and Large-Data Visualization and Graphics*, pages 25–31, 2003.

[17] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):Page 1105–1134, Dev 2005.

[18] Gadget-2: A Code for Cosmological Simulations of Structure Formation. http://www.mpa-garching.mpg.de/gadget.

[19] ParaView (Parallel Visualization Application). http://www.paraview.org.

```
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &ThisTask);
  MPI_Comm_size(MPI_COMM_WORLD, &NTask);
  load_configuration_file_and_init();

  EPSN_init("gadget2","gadget2.xml",ThisTask,NTask,argc,argv);
  RedGRID_Particles * particles = RedGRID_Particles_create("particles",ThisTask,NTask,nb_dims);
  seriePos = RedGRID_Particles_addSerie(particles,"position",RedGRID_double,nb_dims);
  serieVel = RedGRID_Particles_addSerie(particles,"velocity",RedGRID_double,nb_dims);
  RedGRID_Particles_setCoordinateSerie(particles,seriePos);
  RedGRID_Particles_setNumberOfParticles(particles,NumParticles,MaxParticles);
  RedGRID_Particles_wrap(particles,seriePos,baseAddress,strideStruct,offsetPos);
  RedGRID_Particles_wrap(particles,serieVel,baseAddress,strideStruct,offsetVel);
  EPSN_addData(particles);
  EPSN_ready(); // ready for remote client connections

  EPSN_beginHTM();
  init();
  EPSN_beginLoop("main");
  do {
    EPSN_beginTask("move-particles");
    move_particles();
    EPSN_endTask("move-particles");
    every_timestep_stuff();
    EPSN_beginTask("domain-decomposition");
    domain_decomposition();
    // update the number of particles in case of migration
    RedGRID_Particles_setNumberOfParticles(particles,NumParticles,MaxParticles);
    EPSN_updateData("particles");
    EPSN_endTask("domain-decomposition");
    EPSN_beginTask("compute-acceleration");
    compute_accelerations();
    EPSN_endTask("compute-acceleration");
    advance_and_find_timesteps();
  } while(Time <= TimeMax);
  EPSN_endLoop("main");
  save();
  EPSN_endHTM();

  EPSN_finalize();
  MPI_Finalize();
}
```

Fig. 12. *Gadget2* instrumented pseudo-code for EPSN (*main.c*). The black code represents the Gadget2 main steps; the red code describes the particle data structure used in Gadget2 thanks to the RedGRID API; the blue code is the one required by EPSN to initialize the platform and to delimit the tasks of the HTM.

```
<?xml version="1.0" encoding="UTF-8" ?>
<simulation id="gadget2">

    <data>
      <particles id="particles">
        <serie id="position"/>
        <serie id="velocity"/>
      </particles>
    </data>

    <htm>
      <loop id="main">
        <data-context ref="particles" context="protected"/>
        <task id="move-particles">
          <data-context ref="particles" context="modified"/>
        </task>
        <task id="domain-decomposition">
          <data-context ref="particles" context="protected"/>
        </task>
        <task id="compute-acceleration">
          <data-context ref="particles" context="readable"/>
        </task>
      </loop>
    </htm>

</simulation>
```

Fig. 13. *Gadget2* short XML description for EPSN (*gadget2.xml*).