A Load Balancing Model for 3D HPC Image Reconstruction Applications To Reduce Unnecessary Overheads

J. A. Alvarez-Bermejo and J. Roca-Piera *

Dept of Computer Architecture and Electronics
Universidad de Almería
Ctra. Sacramento S/N. 04120
Spain

Abstract. Scientific applications are high-resource demanding software. HPC constitute a suitable framework to deploy such applications. Porting them to parallel platforms is a tedious task that implies adding a complexity layer to the software. There is still a gap between parallel architectures and want-to-be-parallel software. Abstractions are an useful tool to bridge the gap. This paradigm shift can be used to exploit parallelism and add adaptivity layers easily by means of load balancing strategies. In this paper we present results in porting scientific software to parallel platforms.

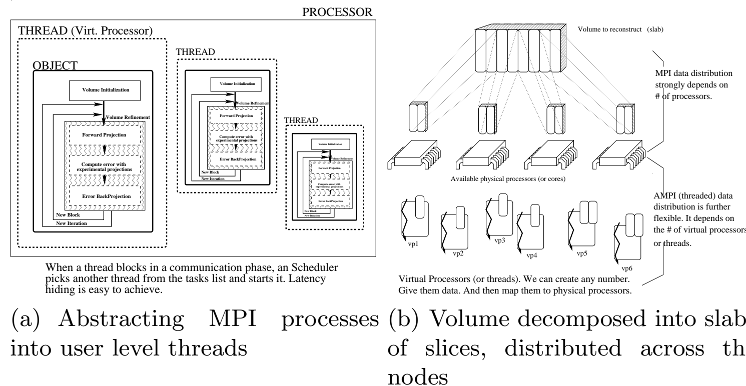
1 Introduction

A high percentage of the scientific code related to image processing is CPU intensive and iterative. HPC and parallelism are almost the unique path for an acceptable solution-to-time ratio. Considerable effort and programming abilities are needed to efficiently parallelize such codes[1]. The application's irregular execution patterns make it difficult to assure optimal runs on a parallel machine, therefore adaptivity to the computational environment is a desired issue. Using load balancers to adapt the application to the environment is a trend but balancers interfere with the code. Our main aim was trying to program efficiently, shifting the programming paradigm and minimizing the gap from algorithm to implementation using more expressive paradigms [2]. We shifted to a particular object oriented paradigm [3] and used the charm framework [4]. This paper gathers the idea of exploiting concurrence and parallelism efficiently using the object orientation pattern. And shows the load balancing techniques we used to achieve adaptivity using our method to avoid interferences from the load balancer at fixed intervals, which may diminish the performance. The paper is structured in Section 2 which introduces the problem of the iterative reconstruction. Section 3 shows improvements achieved when porting our code to the new paradigm. Section 4 exposes how we reached adaptivity, and finally in Section 5 we summarize the conclusions.

* This work has been funded by grant TIN 2008-01117 from the Spanish Ministry of Science and Innovation

2 The Iterative Reconstruction Problem

Series expansion reconstruction methods assume that a 3D object, or function f, can be approximated by a linear combination of a finite set of known and fixed basis functions, with density x_j. The aim is to estimate the unknowns, x_j. These methods are based on an image formation model where the measurements depend linearly on the object in such a way that $y_i = \sum_{j=1}^{J} l_{i,j} \cdot x_j$, where y_i denotes the ith measurement of f and $l_{i,j}$ the value of the ith projection of the jth basis function. Under those assumptions, the image reconstruction problem can be modeled as the inverse problem of estimating the x_j's from the y_i's by solving the system of linear equations aforementioned. Assuming that the whole set of equations in the linear system may be subdivided into B blocks, a generalized version of component averaging methods, *BICAV* [1] can be described. The processing of all the equations in one of the blocks produces a new estimate, see Figure 1(a). All blocks are processed in one iteration of the algorithm. These techniques produce iterations which converge to a weighted least squares solution of the system. A volume can be considered made up of 2D slices. The use of the spherically symmetric volume elements (blobs) [1], makes slices interdependent because of blob's overlapping nature. The amount of communications is proportional to the number of blocks and iterations (as sketched in Figure 1(a)). Reconstruction yields better results as the number of blocks is increased. The main drawback of iterative methods is their high computational requirements. These demands can be faced by means of parallel computing and efficient reconstruction methods with fast convergence. The parallel iterative reconstruction method has been implemented following the Single Program Multiple Data (SPMD) approach. The iterative pattern is certainly appropriated for this problem but imbalanced scenarios may occur if convergence criteria are applied or if the parallel computer is being shared between several users. In this paper we studied how a load balancer could be integrated without interfering excessively with our scientific code. To achive this, concurrence might be of great help so a programming turn was required.



(a) Abstracting MPI processes into user level threads

(b) Volume decomposed into slabs of slices, distributed across the nodes

Fig. 1. Using User Level Threads to Embed MPI Processes

3 The need of a programming shift

Clusters are parallel processing platforms where $BICAV$ can be efficiently solved [1] but load balancing still harms performance due to network communications for exchanging performance profiles. In [5] it is shown how latency can be hidden based upon the programmer abilities. Abstractions provide means to achieve this regardless of the programmer. Multicores are an interesting alternative to clusters, that invite to think in a threaded programming model but adapting parallel HPC applications is still an issue. An interesting abstraction to exploit this new kind of parallelism needed (for cluster and multicores) is by virtualizing processes, see Figure 1(a).

Load balancing

Developing a parallelizable application means adding an extra layer of complexity to the software development process. This complexity refers not only to determine when a certain operation will be processed but where. Making an application adaptive is a two steps procedure, the first one is the heuristic, the second step has to do with how to migrate the computation. Load migration is a scheme reached by consensus for facing imbalance. Conventional strategies indicate the amount of computation units to be moved but say nothing about which of them should be moved to preserve locality and performance. We need to consider locality. Also we needed to optimize the communication between the user code and the load balancer strategies. We used a Runtime System where the LB strategies can be callable from the user code. In order to implement an efficient communication between code an load balancer we used abstractions like objects. Iterative applications must statically specify points in its time-line where the balancer must act, otherwise there is no way to let the runtime collect performance data, evaluate it and propose migrations. So what about creating an object that acts like a coach? An object that is able to detect performance decrements and then invoke the load balancer? This method works as a *swimming team*, the proposed object controls the expected time lapse, as a coach does. If a *slow swimmer enters your lane* you will experience worst time lapses. In our case the *solution* is already implemented in the load balancing strategy, so the coach just need to invoke it. Probably the slower *swimmer* will not be removed away because it does not belong to our team (computing set) but my objects can be moved to better lanes. See Figure 2 in it several cases are shown. Each lane can be considered as a node (cluster) or a core (multicore), and even the three *pools* may be the case of a cluster of multicore nodes. We need to reimplement our code to use abstractions to exploit concurrence and to optimize the calls to load balancers.

Our First Approach

We reimplemented $BICAV$ [6] using AMPI [4], which is a *framework* that allows the virtualization of MPI processes. Concurrence is then a consequence of having

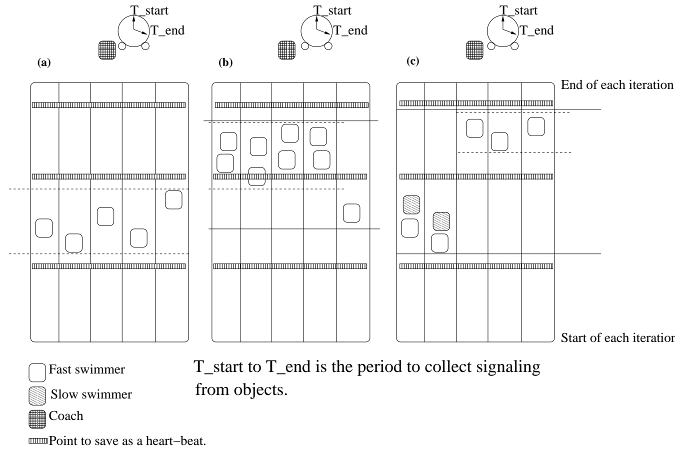


Fig. 2. Swimmers based model

more virtual processors than physical processors. In our application the data is distributed as depicted in Figure 1(b). Communications between neighbor nodes to compute the forward-projection or the error back-projection for a given slab and to keep redundant slices updated are mandatory. The communication rate arises with the number of blocks and iterations. There will be almost no penalty for those virtual processors containing non boundaries slices because the communication will be carried out within the node. Table 1 underlines the gains in the coarsegrain implementation versus MPI. The test reconstructed two volumes, a 256x256x256 volume and a 512x512x512 volume, the number of blocks (K) was set to the maximum (K=256 and K=512, respectively). The efficiency was defined, for these tests, in terms of the idle time computed per processor. Table 1 presents the relative difference (columns *Idle%*) among cputime and walltime for both problem sizes. For the new version, the computed walltime and cputime are almost the same, so cpu was not idling. MPI version behaved worst as the number of processors grew up. Our version (used 128 virtual processors) seized concurrence at maximum. Experiences were performed on a cluster with (*32* computing nodes with two Pentium IV xeon 3.06 Ghz with 512KB L2 Cache and a 2GB sdram).

Table 1. % Relative differences between CPU and WALL times

	K 256 (volume 256)		K 512 (volume 512)	
	MPI	AMPI	MPI	AMPI
Procs	Idle%	Idle%	Idle%	Idle%
2	2.9	0.1	2.8	0.0
4	3.5	0	3.2	0.0
8	5.6	0	4.7	0.3
16	17.1	0.8	9.7	0.7
32	62.4	1.5	32.3	0.2

Implementation from scratch

Objects are good for multicores: they protect their internal data and protect their internal state and do not share it. Objects properties define a scenario based in local and separated environments so objects can be executed in parallel. Objects are structural units and concurrent units. *Parallel Objects* (PO)[7] is an object model where parallelism as well as non determinism can be expressed easily. With this finer-grain implementation, intra-object and inter-object parallelism were exploited. The former with concurrent methods provided by the language, the latter was difficult to achieve in the AMPI version, we implemented a High Level Parallel Composition (HLPC, compositions where internal schedulers are provided together with concurrence control mechanisms) Farm Pattern [8], where there exists a group of concurrent objects that work in parallel under the guidance of a master object (figure 3). To control the concurrence we used MAXPAR [3]. The implementation used a concurrent object framework [4], which is on top of a runtime that *abstracts the beneath architecture*. This runtime offers, separately, the scheduling capabilities to whatever application is run above, avoiding local non determinism because the scheduler is part of the runtime itself. The

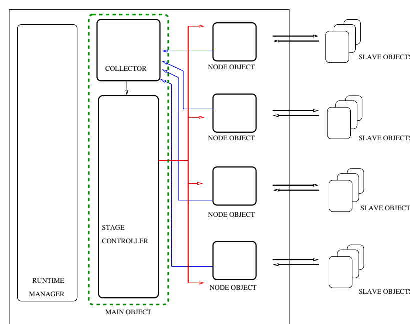


Fig. 3. Object Oriented 3D Reconstruction

Figure 3 shows a *main object* that controls how the program advances. The node objects are placed one per processor, they serve as middleman between slaves and master. Slaves simply do their task concurrently, independently from the processor they are located in. A mechanism for providing *automatic adaptativity* was designed using a special automated object that drive the load balancing (see Section 4). The table 2 shows the behaviour of the reconstruction in MPI and in Charm++ on a cluster and multicore system. The column titled *Objects / Procs* refers to the number of processes started in the multicore for the MPI version; and objects created in the Object Oriented version. When running on the cluster, the column *Procs* means physical processors. Charm++ tests were always launched with four worker objects per processor in the cluster platform. The reconstruction in this version is message-driven, where a message activates an object's service. As Table 2 shows, the object-oriented version behaves better than its MPI counterpart. Using a cluster, the main harm that this scenario suffers from is the network latencies. Communicating two MPI processes is costly.

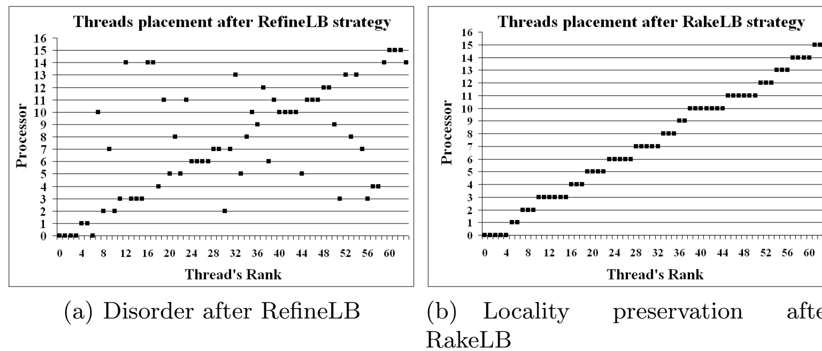
Table 2. Walltime computed in a cluster and in a multicore processor.

	Cluster		Multicore	
Objects/ Procs	MPI WallTime	Objects WallTime	MPI WallTime	Objects WallTime
2	366	42	111,449	4,1
4	86	26	75,651	2,891
8	46	16	45,496	2,697
16	26	14	34,136	2,4
32	n/p	n/p	33,25	2,121

Our new version has better granularity and concurrence, this helps for hiding latencies. Nevertheless when running the application in the multicore (Intel Core 2 Quad Q6600), the MPI version reaches a point (8 processes) where the cache contention affects dramatically the performance. Also one may note that MPI is based on passing messages and although the network remains untouched, when sharing memory this message passing is translated into a copy from private to shared memory where conflicts also exist.

4 Load balancing a mean for adaptive applications

Once the applications are suitable for being splitted into objects we tested them with our strategies to preserve locality [6]. Its behaviour was compared with standard centralized strategies like Refine and Greedy [9] where Greedy strategy does not consider any previous thread-processor assignment, simply builds two queues, processors and threads, and reassigns load to reach average. Refine, in contrast, only migrates objects from overloaded processors until the processor load is pushed under average. Preserving data locality and minimizing latencies (see Section 3) are two issues exploited by RakeLB. We tested *BICAV* to evaluate the dynamic load balancing algorithms. After migration, RakeLB



(a) Disorder after RefineLB

(b) Locality preservation after RakeLB

Fig. 4. Background load and resulting data redistribution after load balancing.

and RefineLB reacted alike, with a fine load distribution as shown by the σ

value. The initial imbalance value (workload placed statically in each node) reflected by σ (standard deviation of the whole system load, normalized to the average load) was over 0.5. After applying the load balancer a similar σ value was achieved (0.051 for GreedyLB and 0.045 for both RakeLB and RefineLB). GreedyLB reached a good load balancing but the distribution of the objects was remarkably messy (see Figure 4) and negative for performance. Figure 4, shows an ordered distribution for RakeLB, aspect that turns out to be an issue for *BICAV*'s performance, as can be seen in Table 3 first row. Locality seemed to be favouring our code. Following we show how we boosted performance replacing fixed calls to the load balancer with asynchronous calls. In Figure 2, (a) represents a computation performing *normally*, no need for load balancing. Each time that an object passes through the stripped line, a *beat* is sent to the coach. (b) shows a case where an object is running on a slower node or core, or may be processing a dense data-set so it progresses slower, the coach should invoke the balancer. (c) identifies a case where our the computation is harmed by a third party, it might be useful to invoke the LB. If we were not using this method, in (a) the load balancer would have been invoked once per a number of iterations, unnecessarily. In (b) and (c) the situation might appear at some point in time, so there is no need to implement *periodic* checks. This method is more flexible in contrast to stopping at load balancing barriers. Results from section 4 were obtained with just one load balancer call, we will compare against that. Table 3

Table 3. Walltime ratio for *BICAV*, pool method

	Greedy	Refine	Rake
Master's Walltime 1 call (% Greedy)	100	66.59	50.29
Master's Walltime 1 per 50(% Greedy 1 call)	101.5	67.4	50.36
Master's Walltime 1 per 20(% Greedy 1 call)	101.62	67.59	50.41
Master's Walltime auto obj(% Greedy 1 call)	100.8	66.63	50.3

shows the application *walltimes* taking as a reference the walltime of the application using the Greedy strategy when the load balancer is invoked just once in the traditional way (Greedy 1 call). In a LB call, the whole computation is stopped into a barrier, the control is passed to the runtime that loads the heuristic, retrieves the object's traces and decide if the mapping should stay as it is or it should migrate. It is a good idea not to abuse from LB calls, our method

```
Data: trigger ⟵ NumberOfBeatsToCollect
Result: Invoke load balancer if the beat slows down
if (−−trigger==0) then
    lapse ← gettime();
    if (CollectFirstTime) then
        auto.getFasterTime(lapse);
    if lapse−idealLapse > ϵ then
        objects[].signalObjectsLBCallRecommended();
    trigger ← NumberOfBeatsToCollect ;
```

Algorithm 1: Algorithm for the method beat(), *auto* object

places an object (*auto object*) per core / node. Computing objects send a simple lowpriority message each time that the object passes through the check points in the code. The *auto object* collects a predefined number of beats, computes the time to collect them and it the time intervals are worst each time, then the LB is invoked. Algorithm 1 describes the basic workings of the *auto-object*.

5 Conclusions

Object oriented abstractions can efficiently exploit parallelism. As a consequence latency hiding and adaptivity issues are easier to achieve. Load balancing strategies must be carefully devised to be locality aware, we have shown that this benefits the walltime of the application. Load balacers must at some point in time, interfere the application to collect data and re-balance the problem, this means loosing flexibility by having to specify statically the points in time when the load balancer should enter in action. A flexible method to invoke load balancers was presented and shown to be effective.

References

1. S. Matej and RM Lewitt. Practical considerations for 3-D image reconstruction using spherically symmetric volume elements. *IEEE Transactions on Medical Imaging*, 15(1):68–78, 1996.
2. José A. Álvarez, Javier Roca-Piera, and José J. Fernández. From structured to object oriented programming in parallel algorithms for 3d image reconstruction. In *POOSC '09: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, pages 1–8, New York, NY, USA, 2009. ACM.
3. A. Corradi and L. Leonardi. Concurrency within objects: layered approach. *Inf. Softw. Technol.*, 33(6):403–412, 1991.
4. Laxmikant V. Kalé and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *OOPSLA*, pages 91–108, 1993.
5. J.J. Fernandez, A.F. Lawrence, J. Roca, I. Garcia, M.H. Ellisman, and J.M. Carazo. High performance electron tomography of complex biological specimens. *Journal of Structural Biology.*, 138:6–20, 2002.
6. J.A. Alvarez, J. Roca, and J.J. Fernández. A load balancing framework in multithreaded tomographic reconstruction. In *Proceedings of the International Conference ParCo 2007*, page in press, Aachen-Julich, September 2007.
7. Antonio Corradi and Letizia Leonardi. Po: an object model to express parallelism. *SIGPLAN Notices*, 24(4):152–155, 1989.
8. Mario Rossainz López and Manuel I. Capel Tunón. An approach to structured parallel programming based on a composition. In *CONIELECOMP*, page 42. IEEE Computer Society, 2006.
9. Gagan Aggarwal, Rajeev Motwani, and An Zhu. The load rebalancing problem. *J. Algorithms*, 60(1):42–59, 2006.