A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model

Eduardo R. Rodrigues
Philippe O. A. Navaux
Institute of Informatics
Federal University of
Rio Grande do Sul
Porto Alegre - Brazil
{*errodrigues, navaux*}*@inf.ufrgs.br*

Jairo Panetta
Center for Weather Forecast and
Climate Studies - INPE
Cachoeira Paulista - Brazil
panetta@cptec.inpe.br

Alvaro Fazenda
Science and Technology Department
Federal University of Sao Paulo
Sao Jose dos Campos - Brazil
alvaro.fazenda@unifesp.br

Celso L. Mendes
Laxmikant V. Kale
Parallel Programming Laboratory
University of Illinois at
Urbana-Champaign
Urbana - USA
{*cmendes, kale*}*@illinois.edu*

Abstract

Among the many reasons for load imbalance in weather forecasting models, the dynamic imbalance caused by localized variations on the state of the atmosphere is the hardest one to handle. As an example, active thunderstorms may substantially increase load at a certain timestep with respect to previous timesteps in an unpredictable manner – after all, tracking storms is one of the reasons for running a weather forecasting model. In this paper, we present a comparative analysis of different load balancing algorithms to deal with this kind of load imbalance. We analyze the impact of these strategies on computation and communication and the effects caused by the frequency at which the load balancer is invoked on execution time. This is done without any code modification, employing the concept of processor virtualization, which basically means that the domain is over-decomposed and the unit of rebalance is a sub-domain. With this approach, we were able to reduce the execution time of a full, real-world weather model.

1. Introduction

Currently, there is an increasing demand for higher resolution weather forecasting simulations. Some weather forecast centers are already running models at resolutions of a few kilometers and those are soon expected to increase further. However, increasing resolution is not just a matter of running the same model code with a finer mesh. As resolution increases, the executed code changes to simulate new phenomena that were previously in a sub-grid scale. Higher resolution allows the representation of localized phenomena that cannot be explicitly treated at larger scales. One example is small to medium scale cloud formation, which is treated by statistical methods at scales larger than the cloud itself and by explicit methods at finer scales.

A concrete instance of this fact is cumulus convection. At lower resolution, this phenomenon is usually parameterized [8]. Meanwhile, at resolutions of a few kilometers, it is possible to use cloud microphysics. This component is concerned with the formation, growth and precipitation of raindrops and snowflakes. This atmospheric process does not have horizontal data dependences, but it may suffer from load imbalance. Indeed, it is well known that thunderstorms cause this problem. Other sources of load imbalance are chemical and biological processes, such as those involved with burning of biomass.

Therefore, as a consequence of increasing resolution and complexity, weather forecast models face load imbalance. There has been some research on the usage of load balancing strategies in meteorological models, but virtually no production code has this feature. The reason is that it is hard to implement load balancing in legacy codes. Consequently, comparing load balancing algorithms in the context of weather models is difficult.

In this paper, we take a different approach: we use the concept of processor virtualization. Instead of inserting the load balancer into the application code, we use a virtualized implementation of MPI to decouple the load balancing strategy from the model itself. The domain is over-decomposed in more sub-domains than physical processors and each sub-domain is assigned to a "virtual processor". Each physical processor handles a set of virtual processors. Load imbalance is addressed by moving virtual processors from overloaded physical processor to underloaded ones.

In this virtualized environment, we are able to compare different algorithms to deal with load imbalance of a real-

---

This work was partially supported by grants from the National Council for Scientific and Technological Development (CNPq-Brazil) and from the US Dep.Energy (#DE-SC0001845). Our tests used NSF's TeraGrid machines, under grants TG-ASC050039N and TG-ASC050040N. The author Eduardo R. Rodrigues was supported by the Brazilian Ministry of Education - CAPES, grant 1080-09-1.

world meteorological model. Here, we analyze the impact of several load balancing strategies on computation and communication and the effects of the frequency of the load balancer invocation on execution time. In our study case, we employed the weather forecasting model BRAMS, which is a mesoscale model written in Fortran 90. The source of imbalance is an active thunderstorm moving in the Southeast region of Brazil.

The remainder of this paper is organized as follows: Section 2 reviews related work; Section 3 describes the load balancing methodology that we use; Section 4 presents the results from our experiments, and finally Section 5 draws conclusions.

2. Related Work

Conventional load-balancing techniques for parallel systems are divided in two classes: centralized and fully distributed. In centralized schemes [2, 15], the balancing decision is made at a single point and communicated to the entire system. These schemes have as their major advantage the full knowledge of global information about system status, which enables better load-balancing decisions. However, centralized schemes are inherently difficult to scale: the single decision point may become a bottleneck, both to receive information from the other processors and to store all that information in a single processor's memory. Meanwhile, fully distributed load-balancing methods [4, 12] are designed to be scalable, but they tend to yield poor balance quality on extremely large machines or tend to take much longer time to yield good solutions in a rapidly changing environment. With the recent growth in the size of parallel machines, a new class of hybrid balancers is emerging [19, 20]. These new balancers divide the processors in groups, applying local balancing methods inside each group and global methods across representatives of each group. Hence, these hierarchical balancers tend to retain the advantages of both centralized and distributed schemes.

Load imbalance has been widely recognized as a problem in weather and climate models. Xue *et al* [18] reported that sub-domains assigned to some processors may incur 20%-30% additional computation due to active thunderstorms. They also claimed that the complexity of the associated algorithm and the overhead imposed by the movement of load prevent using load balancing techniques. Due to these difficulties, some models either ignore the imbalance problem or adopt very simple, static techniques to address it – a typical example is the assignment of "low" and "high" latitudes to the same processor, such that the combined number of grid points per processor remains uniform.

Foster and Toonen [6] conducted one of the few attempts to dynamically balance the load in a climate code. They proposed a scheme based on a carefully planned exchange of data across processors at each timestep. When applying their scheme to the PCCM2 climate model [5], they achieved an overall improvement of 10% on 128 processors, but that improvement degraded with more processors. This technique requires a significant amount of data exchange between processors at each timestep. As the model is scaled, this overhead may dominate execution and offset any potential gains provided by load balancing. Also, implementing this scheme requires intimate knowledge of the application's code, to determine which variables must be exchanged between processors. To the best of our knowledge, there is no existing dynamic load-balancing scheme for weather and climate models that is independent of the model's code.

3. Balancing Methodology

In this section, we describe our methodology to balance the load in BRAMS. We rely on AMPI [10], an MPI implementation based on Charm++. By using AMPI's processor virtualization, we can migrate work across processors dynamically, aiming to balance the load among them. After presenting the major features of Charm++ and AMPI, we discuss a formal solution to the balancing problem and the various heuristics that we explored to address that problem.

3.1 Processor Virtualization

Charm++ is an object-oriented parallel programming system aimed at improving productivity in parallel programming while enhancing scalable parallel performance [11]. A guiding principle behind the design of Charm++ is to automate and optimize the division of labor between the "system" and the programmer. In particular, the programmer can specify what to do in parallel relatively easily, while the system can best decide what processors own which data units and what work units they execute.

Charm++ employs the idea of *overdecomposition* or *processor virtualization* based on migratable objects. In this approach, the programmer decomposes the application's data into a number of objects that is larger than the number of available processors. Those objects are automatically mapped to the underlying processors by the Charm++ runtime system. In addition, the objects can migrate across processors during execution. Each of the objects residing on a certain processor is implemented as a user-level thread, which ensures fast context-switch, and is given control of the CPU by a local scheduler provided by Charm++, in a non-preemptive fashion.

From a programmer's perspective, execution of a Charm++ program proceeds as if each object had its own processor, which we denote as a "virtual processor" (VP).

---

http://brams.cptec.inpe.br/

The set of objects, or *virtual processors*, selected by the programmer is partitioned by the runtime system across the available physical processors. The ratio between the total number of virtual processors and the number of physical processors is called the *virtualization ratio*. By employing multiple virtual processors on each processor, when a virtual processor blocks waiting for a message another virtual processor may execute. This scheme greatly improves the overlap between computation and communication.

Adaptive MPI (AMPI) is a full MPI implementation built on top of Charm++ [10]. In AMPI, each MPI "rank" is implemented as a user-level thread embedded in a Charm++ virtual processor. This approach ensures that the benefits of Charm++ are available to the broad class of applications written using MPI. Thus, an MPI program designed to be run on K processors is typically executed by AMPI with K virtual processors on P physical processors, where $K \geq P$.

3.2 Model for Ideal Balancing

The problem of balancing N communicating threads among M processors can be modeled by Mixed Integer Quadratic Programming (MIQP). There are two objectives: (1) minimize the imbalance among processors and (2) minimize communication between any two processors. The second objective is necessary in order to guarantee that threads that communicate frequently are mapped close to each other and therefore the communication cost is reduced. We have previously shown that, in a cluster of multi-core machines, mapping sub-domains of a meteorological model appropriately can reduce by up to 9% the total execution time [16].

The MIQP model is the following:

$$minimize \qquad f : \sum_{i=0}^{M-1} \left[\left(\sum_{j=0}^{N-1} w_j x_{ij} \right) - W_{mean} \right]^2 \tag{1}$$

in which w_j is the weight of thread j and W_{mean} is the average load. The variables x_{ij} are binary and represent the placement of thread j on processor i. This objective penalizes processors that have load above and bellow average. The second objective function is:

$$minimize \quad g : \sum_{k=0}^{M-1} \sum_{l=k+1}^{M-1} D x_{ka} x_{lb} + \sum_{k=0}^{M-1} S x_{ka} x_{kb}, \qquad \forall\, a, b \;/\; a \text{ communicates with } b \tag{2}$$

where D represents the communication cost when two threads that communicate with each other are placed in different processors, while S represents the cost when these threads are placed in the same processor. Again, x is a binary variable. This function penalizes communicating threads that are placed in separate processors. The constraint of this model is:

$$\sum_{i=0}^{M-1} x_{ij} = 1, \forall\, j \tag{3}$$

The problem expressed by the first objective function is a generalization of the multiprocessor scheduling problem [7] and is known to be NP-complete.

Solving this model to optimality can take a very long time on current machines, even for small cases. Indeed, solving the case $\{M = 4, N = 16\}$ takes several minutes with the state of art solver (CPLEX). Consequently, heuristics must be used to deal with realistic cases and obtain a result in a feasible amount of time. Describing some of those heuristics is the subject of the next subsection.

3.3 Balancing Algorithms Employed

By leveraging its thread-migration capability, Charm++ provides a powerful infrastructure for measurement-based load balancing [19]. Automatic instrumentation in the runtime system allows the capture of computational loads and communication patterns from each thread. This information is stored in a database that can be dynamically updated and used to decide where to map each thread. AMPI programs can benefit from this infrastructure via the AMPI function *MPI_Migrate()*. This is a collective call that invokes the load balancer, marking a point in the execution where migrations can occur if such migrations would lead to better balance of load across processors. The particular balancing policy to be used is chosen as a command-line argument.

In general, minimal changes to an original MPI code are required to use this load balancing infrastructure with AMPI. For an iterative code, all that is needed is the insertion of calls to *MPI_Migrate()* at certain iterations, according to some pre-determined criteria. In the experiments that we report in Section 4, we explored calling the load balancer both after a fixed number of iterations or just at iterations where the imbalance across processors was higher than a certain threshold. The bottom-line is that the implementation of a particular balancing policy is totally isolated from the application's source code, thus no changes to the application are required for using a given load balancer.

We investigated the use of various load balancers available in Charm++: *GreedyLB*, *RefineCommLB*, *RecBisectBfLB* and *MetisLB*. *GreedyLB* is a load balancer that has simplicity as its major feature; the thread with the heaviest computational load is assigned to the least loaded processor, and this continues until a balance is reached. Hence, no communication information is considered, which can lead to a situation where two threads that communicate intensely are placed in distinct processors. However, given the simplicity of this policy, the balancing process is often very fast.

RefineCommLB is a balancer that takes both computational load and communication traffic into account. It attempts to move objects away from the most overloaded processors to reach average, but also considers how that movement would affect locality of communication. In addition, it limits the number of migrations, regardless of the observed loads. In general, this balancer is used for cases when moving just a few threads is sufficient to achieve balance.

The *RecBisectBfLB* balancer recursively partitions the communication graph of threads with a breadth-first enumeration; the partitioning is done based on the computational loads of the threads, until the number of partitions is equal to the number of processors. Although communication is considered by this scheme, there is no explicit guarantees that the resulting communication volume across partitions is minimized.

Meanwhile, *MetisLB* is a balancer that uses Metis [14] to partition the thread communication graph. Both the computational load and communication pattern are considered. All of these Charm++ balancers employ a centralized approach, which works well for a moderate number of processors. However, none of them directly takes into account the spatial relationship between threads.

In BRAMS, like in many other weather forecasting models, the atmosphere is represented with a three-dimensional grid of points. Those points are distributed across the MPI ranks according to a domain decomposition in the latitude/longitude plane. Each rank receives the full atmospheric columns corresponding to the points in its domain. Because the ranks are implemented by threads in AMPI, there is a high volume of communication between threads associated to ranks from domains that are neighbors. Hence, mapping two threads from neighbor domains to the same physical processor will ensure that their communication is local to that processor, which minimizes the communication overhead.

To preserve this spatial relationship between threads more explicitly, we developed a new load balancer, *HilbertLB* [17], based on a Hilbert space-filling curve [9]. We traverse the threads with a Hilbert curve that covers the entire domain of the forecast (see Figure 1), and recursively bisect that curve according to the observed loads of the threads. This process is repeated until the number of segments is equal to the number of physical processors. The resulting segments will have approximately the same load, and each segment should contain threads that represent neighbor regions.

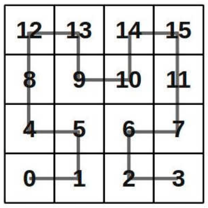


Figure 1. Hilbert curve for the case of 16 threads

4. Experimental Results

Our study case is a forecast of a moving thunderstorm in the Southeast region of Brazil. We configured BRAMS to use a grid of 512×512 horizontal points and 40 vertical levels. The resolution was 1.6 Km and the timestep was 6 seconds. We conducted forecasts of 4 hours, corresponding to executions with 2,400 timesteps. These experiments were run on a Cray XT5 system at Oak Ridge National Lab., whose nodes have two six-core AMD Opteron processors at 2.6 GHz. The network connection is a SeaStart2+. We used 64 physical processors and 1024 virtual processors. To analyze in detail the executions, we used *Projections* [13], a performance analysis tool from the Charm++ infrastructure.

We divided the experiments in three groups. The first group is a basic comparison of all load balancers; subsection 4.1 presents the execution times for each algorithm and an analysis of the reasons for these results. In subsection 4.2, we investigate how the frequency of balancing impacts performance of a selected algorithm. We also establish a threshold beyond which migration will occur. Subsection 4.3 analyzes a generalization of the *HilbertLB* algorithm to deal with domains of arbitrary sizes; we compare those results with the ones from another experiment using the same domain and the *MetisLB* load balancer.

4.1 Basic Comparison of Balancers

In our first group of experiments, we executed BRAMS with a load-balancing invocation at the end of every forecast hour, corresponding to timesteps 600, 1200 and 1800, respectively. Table 1 presents the BRAMS execution time on these experiments and the corresponding execution time reduction in comparison to the case without virtualization. By just over-decomposing the domain, the application already experiences improved performance (no load balancer case). This is due to overlap of computation and communication coupled to better cache usage. In a previous study [17], we conducted an in-depth analysis of the factors that lead to these improvements.

Configuration	**Execution Time (s)**	**Execution Time Reduction**
No virtualization	4987.51	-
No load balancer - 1024 VP	3713.37	25.55%
GreedyLB - 1024 VP	3768.31	24.45%
RefineCommLB - 1024 VP	3714.92	25.52%
RecBisectBfLB - 1024 VP	4527.60	9.23%
MetisLB - 1024 VP	3393.12	31.97%
HilbertLB - 1024 VP	3366.99	32.50%

Table 1. Load balancing effects on BRAMS (all experiments were run on 64 real processors)

Load Balancer	**Balancing Time (s)**
GreedyLB	80.81
RefineCommLB	10.81
RecBisectBfLB	78.33
MetisLB	81.00
HilbertLB	51.45

Table 2. Observed cost of load balancing

The only load balancers that produced performance gains were *HilbertLB* and *MetisLB*. Although the other load balancers produced better performance in comparison to the non-virtualized case, they actually lost part of the gains obtained from over-decomposition, i.e. there was a reduction in performance when compared to the "No load balancer" case. There are three potential reasons for these results: (a) the cost of executing the balancing algorithm and the migration cost were excessive; (b) the load balancer was unable to rebalance load completely; and (c) the cross-processor communication increased after rebalancing.

To investigate the first reason, Table 2 presents the time each algorithm took to rebalance load. These values correspond to the sum of the timestep durations for the three timesteps where load balance occurred (i.e. timesteps 600, 1200 and 1800); they include both the execution of the balancing algorithm itself and the thread migrations. As it can be seen, there is not much difference among these values, except that *RefineCommLB* was much faster, as expected, since it limits the amount of migration. One of the most expensive algorithms, *MetisLB*, had one of the best application execution times. Therefore, another reason must exist to explain the poor application performance caused by the the first three load balancers in Table 2.

Let us consider the *GreedyLB* load balancer. This balancer was able to rebalance load quite well, as shown in Figure 2(a). This figure plots CPU usage of each physical processor and the first bar is the average CPU usage. The load is well balanced across all processors but the CPU usage is low, with an average near 70%. The reason for this fact is communication, as a CPU becomes idle when it waits for data from its neighbors. Since *GreedyLB* does not consider communication in its balancing decisions, the external (i.e. cross-processor) communication can increase as a consequence of rebalancing. We confirmed this by comparing the cross-processor communication in this experiment with the one from the "No load balancer" case. We found that the cross-processor communication volume increases by a factor of nearly five with *GreedyLB* (see Figure 3).

In turn, *RefineCommLB* kept much of the communication similar to the pattern in the original mapping. However, it did not migrate enough threads to fully rebalance load. Consequently, the load was still imbalanced after its use, as confirmed by the utilization plot of Figure 2(b). The problem with this load balancer is that it assumes the load is almost balanced and it will just perform a refinement (as its name implies). The imbalance of our experiment, in contrast, was much larger than what *RefineCommLB* could effectively handle.

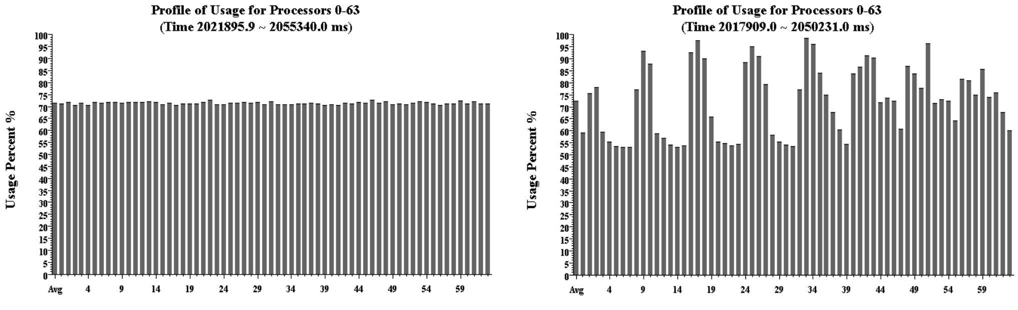
With *RecBisectBfLB*, a good balance was achieved, as shown in Figure 2(c). However, the utilization was quite low, with an average near 55%. We noted that the use of this balancer resulted in some processors having a much larger communication frequency than others. Because the execution in BRAMS proceeds with an implicit synchronization caused by the exchange of boundary data between sub-domains at each timestep, delays in one processor may slow down the entire execution. We are conducting additional checks to confirm that this was indeed the cause for the poor performance of *RecBisectBfLB* observed in Table 1.

Finally, for *MetisLB* and *HilbertLB*, which achieved the best performance in Table 1, the balance was good and the average utilization was quite high; Figure 2(d) shows those details for *MetisLB*, while similar behavior was observed with *HilbertLB*.

4.2 Adaptive Balance Period

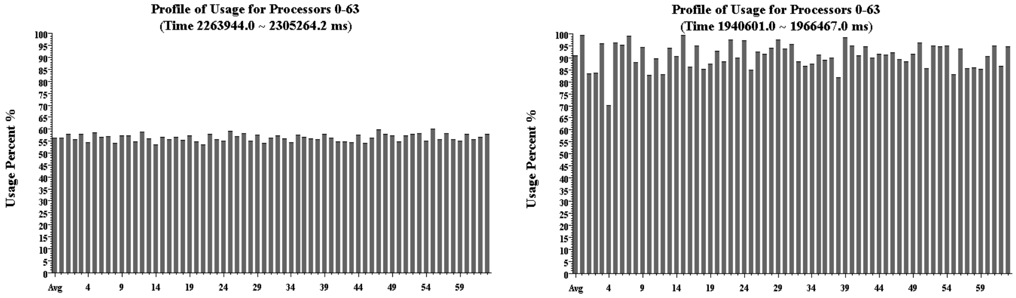
In the previous set of experiments, a fixed load-balancing invocation scheme was used and migrations would occur whenever there was imbalance, regardless of the amount of imbalance. Since the cost of executing the load balancing algorithm is usually low but the thread migration cost may be high, an adaptive balancing scheme can be more effective. In this new scheme, the load balancer is invoked more frequently and migrations occur only if the observed imbalance is beyond a given threshold.

In this subsection, we evaluate if the performance of BRAMS can be improved by using this adaptive scheme. We conducted executions invoking the load balancer every 100, 10 or 1 timestep(s), respectively. Also, we selected



(a) GreedyLB

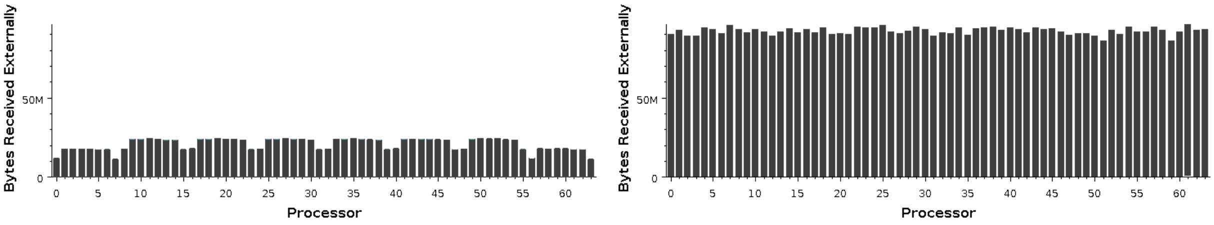
(b) RefineCommLB



(c) RecBisectBfLB

(d) MetisLB

Figure 2. CPU usage under different load balancers



(a) No load balancer

(b) *GreedyLB* balancer

Figure 3. Cross-processor communication volume

the following imbalance thresholds to trigger migrations: 50%, 20%, 10% and 5%. We quantify the imbalance by how much the load in the most loaded processor was above the average processor load. Here, only the *HilberLB* load balancer was used. Table 3 presents the BRAMS execution times from these experiments.

It is possible to see that a high threshold hurts execution time. The threshold of 50% produced worse results than the fixed invocation scheme used previously. That is because this threshold was too high and did not trigger enough migrations to neutralize the imbalance. Decreasing the load balancer invocation interval in this case reduces execution time because the few occasions where the imbalance reaches 50% are detected sooner. However, a very low invocation interval should not produce any benefit, because the load is highly unlikely to reach an imbalance of 50% or more in such a few timesteps after rebalancing.

A threshold of 20% improves performance even with a frequency of one invocation per timestep. The reason for this is that the cost of simply executing the *HilbertLB* algorithm without any migration is very low. Furthermore, a threshold of 20% is a good trade-off between imbalance and migration cost, i.e. migration (with its high cost) will not occur so frequently even if the balancing algorithm is being called at every timestep.

With a threshold of 10%, the performance reaches

LB Interval	**Imbalance Threshold**			
(Timesteps)	50%	20%	10%	5%
100	3639.54	3290.72	3211.10	-
10	3554.07	3179.31	3128.54	3245.03
1	-	3248.85	3872.11	-

Table 3. BRAMS execution time (in seconds) with adaptive load-balancing invocation and *HilbertLB* balancer

its best result, but only with an invocation interval of 10 timesteps. Lowering the invocation interval actually causes performance to be worse than in the case of fixed invocations (Table 1). This is because there are too many migrations and their cost surpasses the benefits from load balance. A similar but milder effect occurs with the threshold of 5% and a balancing interval of 10 timesteps.

In summary, there is an optimal point with this adaptive scheme that is reached when the load balancer invocation frequency is high enough to detect the load variation. In addition, the threshold must be tuned according to that frequency, considering the typical imbalance that may arise within the invocation period. In our experiments, invoking the *HilbertLB* balancer every 10 timesteps and enabling migrations when the imbalance was higher than 10% resulted in a performance gain of 37.3% over the non-virtualized BRAMS execution reported in Table 1.

4.3 Arbitrary Size Domains

Despite being simple, the load balancer based on the Hilbert curve was very effective, as demonstrated by the performance observed in Table 1. However, this original algorithm has a quite strong restriction concerning domain geometry: the domain must be a square and its side must be a power of two. Chung, Huang and Liu [3] proposed an algorithm to overcome this limitation, allowing the use of rectangular domains of any size. Those authors used that algorithm in image processing. Here, we implemented the same algorithm to perform load balancing. In addition, we compared this approach with the *MetisLB* load balancer, which had also resulted in good application performance.

The algorithm starts by finding the biggest square inside the original domain. This square is placed at the upper left corner of the domain. This step is applied recursively to the remaining area of the original rectangle, as illustrated in Figure 4(a). Each square of the previous step is further decomposed into smaller squares whose side is a power of two. In order to do that, a "snake-scan" approach is used, as shown in Figure 4(b). Finally, each smaller square is filled with the regular Hilbert curve following the direction used in the previous step (Figure 4(c)).

We ran another experiment with an atmospheric grid consisting of 448×348 horizontal points and 40 vertical levels. In this experiment, each thread had also 16×16 atmospheric columns, like before. The domain was decomposed into 28×24 sub-domains, for a total of 672 threads, and we used 32 physical processors. The BRAMS forecast was for a region with the same thunderstorm of the previous sections. The same forecast duration and timestep were used.

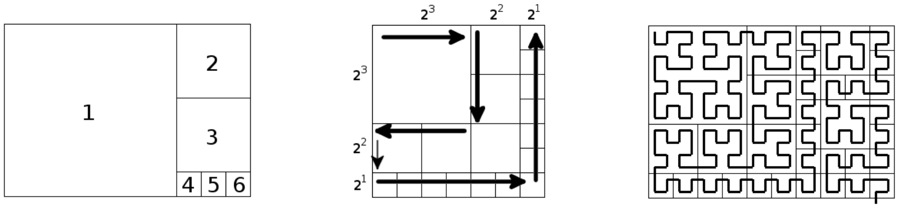
The execution time with the generalized *HilberLB* load balancer was 2.7% larger than with *MetisLB*. The reason for this fact was communication: an analysis with Projections (not shown here) indicated a small increase of cross-processor communication in the execution with the new *HilbertLB* in comparison to *MetisLB*. This result was expected, since, with this generalization of the Hilbert curve, the threads in each processor will not form such tight clusters as they did before. Consequently, the contact surface between processors will be larger and that explains the increase in communication and decrease in performance.

Despite that slightly lower observed performance, *HilbertLB* has an advantage: it can be implemented in a distributed fashion. Each processor can compute the Hilbert sequence locally and the bisectioning algorithm can be efficiently implemented with parallel prefix [1]. Consequently, as the number of processors increases to many thousands, this approach is expected to produce better performance, because in a machine of that size a centralized load balancer may become a bottleneck.

5. Conclusions

Load imbalance is a key obstacle to the scalability of weather and climate models. While some causes of imbalance can be handled by static methods, other causes are highly unpredictable and may vary with the input data. Hence, dynamic load balancing techniques are needed to obtain good scalability. Adding dynamic load balancing to an existing model, however, is a very challenging task.

In this paper, we presented a comparative analysis of different load balancing algorithms applied to the BRAMS weather forecast model. By leveraging the over-decomposition and processor virtualization techniques provided by Charm++ and AMPI, we were able to employ those balancers without any changes to the BRAMS code. In our reported experiments, we first compared the performance achieved by invoking the various balancers at fixed points in the simulation. The obtained results show that the *MetisLB* and *HilbertLB* balancers produced the best performance. Next, we investigated an adaptive scheme that invokes the load balancing algorithm more frequently, but only migrates work across processors when the imbalance is beyond a certain threshold. With this adaptive scheme, we



(a) Divide the original domain into squares

(b) For each square, apply the "snake-scan" approach

(c) Use the conventional Hilbert algorithm

Figure 4. Hilbert curve for domains of arbitrary size

observed an improvement in performance of 37.3% compared to a non-virtualized BRAMS execution. Finally, we extended the *HilbertLB* algorithm to enable its use with arbitrarily sized domains, and showed that its obtained performance remains comparable to that of *MetisLB*.

While the present results clearly show that dynamic load balancing via processor virtualization is a feasible approach for weather models, we continue to work to further improve its efficiency and usability. One of our current efforts is to investigate the use of information from meteorological variables to guide the balancing decisions. Another direction of interest is the creation of a technique to automatically select the best invocation frequency and imbalance threshold for the adaptive load balancing scheme.

References

[1] A. Bhatele, G. Gupta, L. V. Kalé, and I.-H. Chung. Automated mapping of regular communication graphs on mesh and torus interconnects, 2010. Submitted for publication.

[2] Y.-C. Chow and W. H. Kohler. Models for dynamic load balancing in homogeneous multiple processor systems. In *IEEE Transactions on Computers*, volume c-36, pages 667–679, May 1982.

[3] K. Chung, Y. Huang, and Y. Liu. Efficient algorithms for coding Hilbert curve of arbitrary-sized image and application to window query. *Information Sciences*, 177(10):2130–2151, 2007.

[4] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive Load Balancing Policies for Dynamic Applications. In *IEEE Concurrency*, pages 7(1):22–31, 1999.

[5] J. Drake, I. Foster, J. J. Hack, J. Michalakes, B. D. Semeraro, B. Toonen, D. L. Williamson, and P. Worley. PCCM2: A GCM adapted for scalable parallel computers. In *Proc. AMS Annual Meeting, AMS*, pages 91–98. American Meteorological Society, 1994.

[6] I. Foster and B. Toonen. Load-balancing algorithms for climate models. *Scalable High-Performance Computing Conference*, pages 674–681, 1994.

[7] G. Fox, R. Williams, and P. Messina. *Parallel computing works!* Morgan Kaufmann Pub, 1994.

[8] A. G. Grell and D. A. Dévényi. A new approach to parameterizing convection using ensemble and data assimilation techniques. *Geophysical Research Letters*, 29:1693, 2002.

[9] D. Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[10] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[11] L. Kalé. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, Aug. 1990.

[12] L. V. Kalé. Comparing the performance of two dynamic load distribution methods. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 8–11, St. Charles, IL, August 1988.

[13] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[14] G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. University of Minnesota, 1995.

[15] L. M. Ni and K. Hwang. Optimal Load Balancing in a Multiple Processor System with Many Job Classes. In *IEEE Trans. on Software Eng.*, volume SE-11, 1985.

[16] E. R. Rodrigues, F. L. Madruga, P. O. A. Navaux, and J. Panetta. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In *ISCC*, pages 811–817, 2009.

[17] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, C. L. Mendes, and L. V. Kalé. Optimizing an MPI weather forecasting model via processor virtualization, 2010. Submitted for publication.

[18] M. Xue, K. Droegemeier, and D. Weber. Numerical Prediction of High-Impact Local Weather: A Driver for Petascale Computing. *Petascale Computing: Algorithms and Applications*, pages 103–124, 2007.

[19] G. Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[20] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kalé. Hierarchical Load Balancing for Large Scale Supercomputers. Accepted for the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), September 2010.