Optimizing an MPI Weather Forecasting Model via Processor Virtualization

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 Forecasts and
Climate Studies / INPE
Cachoeira Paulista - Brazil
jairo.panetta@cptec.inpe.br

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

***Abstract*—Weather forecasting models are computationally intensive applications. These models are typically executed in parallel machines and a major obstacle for their scalability is load imbalance. The causes of such imbalance are either static (e.g. topography) or dynamic (e.g. shortwave radiation, moving thunderstorms). Various techniques, often embedded in the application's source code, have been used to address both sources. However, these techniques are inflexible and hard to use in legacy codes.**

In this paper, we demonstrate the effectiveness of processor virtualization for dynamically balancing the load in BRAMS, a mesoscale weather forecasting model based on MPI parallelization. We use the Charm++ infrastructure, with its over-decomposition and object-migration capabilities, to move sub-domains across processors during execution of the model. Processor virtualization enables better overlap between computation and communication and improved cache efficiency. Furthermore, by employing an appropriate load balancer, we achieve better processor utilization while requiring minimal changes to the model's code.

I. INTRODUCTION

Weather forecasting models are undoubtedly an important class of applications. Currently, they have received even more attention because they are an indispensable tool to study climate change. These applications are also computationally intensive and the computer power demand is expected to increase due to higher resolutions, longer simulated periods and more complex models of the atmospheric processes [1]. Therefore, to run these models in a feasible amount of time, they are usually executed in parallel machines. However, a major obstacle to their scalability is load imbalance.

The causes of load imbalance in weather models can be either static or dynamic. One example of a static factor is topography. Many weather models represent the atmosphere with a three-dimensional grid of points, and distribute those points across the processors according to a division of domains in the latitude/longitude plane. Each processor receives the full columns corresponding to the points in its domain. Thus, locations with a high topography will have fewer grid points associated to atmosphere, hence less work to be computed by the model. Although such imbalance changes with the input dataset, it is conceivable that one could derive in advance an ideally balanced decomposition if the model is routinely used on the same region.

The dynamic sources of load imbalance in weather models are much more involved. Some of those sources are predictable, while others are unpredictable. Consider, for example, the effects caused by earth rotation. Two domains with the same latitude receive different values of solar incidence at a given moment; this results in distinct amounts of computation for the radiative components of the model on the two underlying processors. This kind of imbalance repeats with a periodicity of twenty-four hours in simulated time.

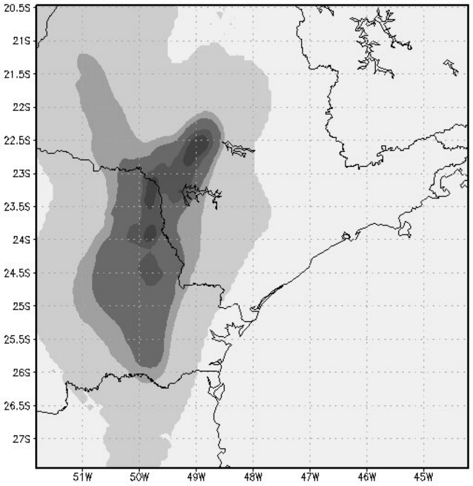
Meanwhile, other imbalance factors lack such predictability. As an example, running a certain weather model (described in Section-III) on 64 processors resulted in the forecast depicted in Figure 1a. Instrumentation of the model revealed that the computational loads in the 8×8 set of processors were as indicated by the grayscale-coded distribution of Figure 1b. There is a clear correlation between rain and computational load: domains containing rain correspond to overloaded processors.

If the simulation of Figure 1a is allowed to proceed, the rain may "move" across domains, changing the distribution of overloaded and under-loaded processors. This movement is unknown a priori (since predicting where the rain will go is precisely what the model is designed for!). Responding to such unpredictable sources of load imbalance in weather models remains mostly an open problem.

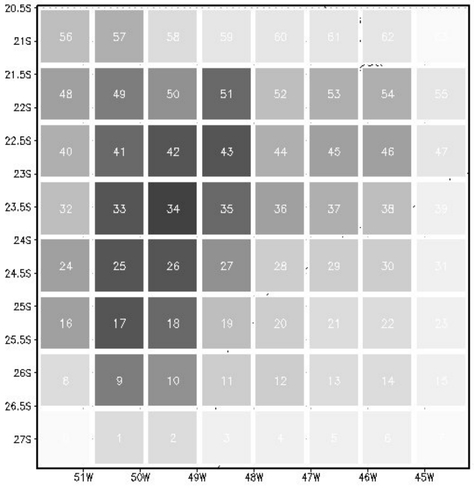
In this paper, we propose a novel technique to address the load balancing problem in weather models. Our technique rests on the concept of processor virtualization, provided by the Charm++ infrastructure [2]. Charm++ empowers MPI applications with a proven runtime system that can dynamically migrate work across the processors of a parallel machine. By leveraging this migration capability, Charm++ supports a dynamic, measurement-based load balancing mechanism.

Our approach can handle both predictable and unpredictable sources of load imbalance in a uniform fashion. Moreover, minimal changes to the original application code are required.

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.



(a) Example of a real weather forecast



(b) Grayscale-coding of observed computational load on 64 processors

Fig. 1: Dynamic source of load imbalance

We present experimental results with an existing weather forecasting model, using real atmospheric data, to show that it is possible to improve the model's performance by employing appropriate load balancers.

The main contribution of this paper is to demonstrate that processor virtualization can be effectively used to improve performance of MPI applications that suffer from load imbalance, which is typically the case in weather and climate models. The rest of the paper is organized as follows. In Section-II, we review previous work in load balance for weather and climate codes. Section-III presents the tools that we use, namely a particular weather code (BRAMS) and the main features of Charm++. Section-IV describes how we adapted BRAMS to exploit processor virtualization. Section-V contains the results from our experiments, and Section-VI presents our conclusions and future work.

II. RELATED WORK

Various researches have studied the load balancing problem in weather and climate models. Ghan *et al* [3], for example, found that orography, which is a static input, causes load imbalance to the NCAR/CAM model.

Foster and Toonen [4] identified that physics computation also causes load imbalance in climate codes. Examples of physics computation are radiation, which changes with the movement of the planet, and cloud and moisture, which are transported with the movement of the atmosphere. They proposed a dynamic scheme to balance the load based on a carefully planned exchange of data across processors at each timestep. Their rationale was that the model employed three types of timesteps, with varying degrees of radiative calculations, and a good decomposition for one kind of timestep was not as good for the other kinds. When applying their scheme to the PCCM2 climate model, 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 the load balancing scheme. Also, implementing this scheme requires intimate knowledge of the application's code, to determine which variables must be exchanged between processors.

Xue *et al* [5] stated 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 the use of load balancing techniques.

In previous work by some of us [6], simulations had shown the maximum possible gains achievable with dynamic load balancing in RAMS, a predecessor of BRAMS. That study included simulations using an oracle with knowledge about the load throughout the entire execution. However, that ideal scenario assumed that the grid points could be flexibly assigned to any processor; in practice, both RAMS and BRAMS require rectangular domains per processor. These rectangles impose limitations to the distribution of grid points. Our present work, as we show in Section-IV, removes that limitation: the over-decomposition scheme enabled by Charm++ creates sub-domains that are "mini-rectangles" dividing each original domain. The ability to migrate these mini-rectangles enables balancing strategies with much finer granularity. Furthermore, this approach requires no changes to the original algorithm employed by the application.

III. BACKGROUND

A. BRAMS Model

BRAMS (Brazilian developments on the Regional Atmospheric Modeling System, RAMS) is a multipurpose regional numerical prediction model designed to simulate atmospheric circulations at many scales. It is used both for production and research world wide. It has its roots on RAMS [7], which solves the fully compressible non-hydrostatic equations described by Tripoli and Cotton [8], and is equipped with a multiple grid nesting scheme that allows the model equations to be solved simultaneously on any number of two-way interacting computational meshes of increasing spatial resolution. It has a set of state-of-the-art physical parameterizations appropriate to simulate important physical processes such as surface-air exchanges, turbulence, convection, radiation and cloud microphysics.

BRAMS started as a research project aimed to tailor RAMS to the tropics and to modernize its software structure. BRAMS modeling features extended the original RAMS to include cumulus convection representation as part of an ensemble version of deep and shallow cumulus scheme based on the mass flux approach [9], daily soil moisture initialization data [10] and a specific surface scheme that allows the representation of important tropical phenomena. More recently, a coupled aerosol and tracer transport model (CATT-BRAMS [11]) was developed to allow the study of emission, transport and deposition of gases and aerosols associated with biomass burning, such as those originated at the Amazon. CATT-BRAMS has been used in daily production mode at CPTEC to forecast air quality for the entire South America (see http://meioambiente.cptec.inpe.br/).

BRAMS uses Fortran 90 features to eliminate dusty deck software constructs from the original RAMS code, including static memory allocation and the heavy use of Fortran 77 commons, achieving production quality code while maintaining research flexibility. BRAMS is open source code freely available at http://brams.cptec.inpe.br/, supported and maintained by a modest software team at CPTEC that continuously transforms research contributions into production quality code to be incorporated at future code versions. It is also a platform for computer science research in themes such as grid computing [12], [13].

This work uses the current research version (BRAMS 5.0) that has enhanced parallelism when compared to the current production version. Up to the current production version, BRAMS used the original RAMS master-slave parallelism that partitions the horizontal projection of the 3D domain into rectangles as close to squares as possible, assigning one rectangle to each slave process. The current research version eliminates the master-slave parallelism to avoid memory contention on the master process, using all processes on the original domain decomposition. Resulting code eliminated the master memory bottleneck, enhancing parallel scalability up to O(1000) processors [14]. Load balancing became the major scalability bottleneck, partially due to rectangular domain decomposition but mainly due to the dynamic load variation during integration.

B. Processor Virtualization with Charm++

Charm++ [2] is an object-oriented parallel programming system aimed at improving productivity in parallel programming while enhancing scalable parallel performance. A guiding principle behind the design of Charm++ is to automate what the system can do best, while leaving to the programmers what they can do best. It is assumed that programmers can specify what to do in parallel relatively easily, while the system can best decide which processors own which data units, as well as which work units each processor executes. This approach requires an intelligent runtime system, which Charm++ provides.

At its core, Charm++ employs the idea of *processor virtualization*, based on migratable objects. In this approach, the programmer decomposes a problem into a set of N objects that will execute on P processors, where ideally $N >> P$. The programmer's view of the execution is of N objects and their interactions. Meanwhile, the underlying runtime system, implemented by Charm++, maps those objects to the P processors. This mapping is dynamic and objects can migrate across processors during execution, under control of the runtime system. That over-decomposition scheme effectively decouples the partitioning of the problem from the physical machine where the program will run. This, in turn, provides many opportunities for runtime optimizations, such as better overlap between computation and communication, or improved communication strategies.

Object-based virtualization leads to programs that automatically respect locality, in part because objects provide a natural encapsulation mechanism. At the same time, it empowers the runtime system to automate resource management. The combination of features in Charm++ has made it suitable for the expression of parallelism over a range of architectures, from multi-core desktops to existing petaFLOP-scale parallel machines. Moreover, it has enabled scaling real applications to thousands of processors on several scientific areas, such as molecular dynamics [15], quantum chemistry [16], computational cosmology [17], rocket simulation [18] and others.

C. Adaptive MPI

Adaptive MPI (AMPI) is an implementation of the MPI standard based on Charm++ [19]. In AMPI, each MPI task is embedded in a Charm++ object and implemented as a user-level thread. Differently from kernel-threads, these user-level threads are lightweight and result in very short context-switch times [20]. Like any Charm++ object, those threads can migrate across processors as well. Hence, by using AMPI, many of the benefits from processor virtualization become available to *legacy* MPI applications, written in C/C++/Fortran.

In AMPI, because N Charm++ objects are used to implement the original MPI tasks, it is common to refer to those objects as *Virtual Processors* (VPs). Thus, an MPI code designed with a decomposition into N tasks has each of those tasks presented with the "illusion" of owning an AMPI virtual processor. Many VPs can share a physical processor during execution: each VP is associated to one of the user-level threads comprising the process that is running on that processor. The ratio between the number of VPs and the number of physical processors (P) available for the program is called the *virtualization ratio*. The ideal values for this ratio depend strongly on the underlying application.

A potential problem that may arise from the sharing of a physical processor by multiple VPs is a conflict in the access to global and static variables in the application. This is because the VPs are implemented as threads, therefore they share the same address space. With the original MPI, this conflict does not exist because each task has its own address space, hence a given global or static variable can be accessed by only one task. To resolve that conflict in AMPI, it is necessary to *privatize* those variables. There are a few different mechanisms for such privatization, with varying degrees of automation. Although the privatization process can result in some overhead due to context-switching during execution, there are techniques that keep this overhead low, regardless of the number of globals or statics in the application's code [21]. We demonstrate, in Section-IV, possible ways to handle this privatization issue.

D. Load Balancing in Charm++ and AMPI

The Charm++ runtime system has a powerful load-balancing infrastructure. This infrastructure contains instrumentation that, when enabled, automatically captures information about computational load and communication traffic from the various application sections. That information is typically used by the runtime system to decide how to redistribute the load across the underlying processors. The rationale in this approach is that observations from the recent past provide a good predictor of the execution in the near future.

Through this measurement-based load-balancing technique, several different balancing policies have been developed and become available for use by any Charm++ application [22]. Some of those policies focus on computation only, others favor communication, and there are also policies that combine those two aspects or consider the topology of the interconnection network [23]. It is also very simple to write new balancers that implement other policies. The specific policy to be used in an execution is selected in the command line.

In addition to the measured factors, the balancers can also employ application-specific information. There is an application interface that allows the programmer to pass arbitrary values to be considered in the balancing decisions. Those values can be derived from characteristics of the particular application.

The load-balancing infrastructure of Charm++ is made available to MPI programs through an AMPI extension implemented with the *MPI_Migrate()* call. This function is a collective operation that, when invoked, triggers the load balancing mechanism. At this invocation, AMPI threads may migrate across processors, if the policy being used determines that such migration will improve application's performance.

IV. BRAMS ADAPTATION TO AMPI

In general, adapting a given MPI code to use AMPI is a simple process. The only issue that requires attention is the proper handling of global and static variables, as explained in the previous section. In this section, we show what had to be done with BRAMS in regard to that. Additionally, we describe a new load balancer that we developed in Charm++, to implement a balancing policy more convenient for the characteristics of codes with a two-dimensional domain decomposition like BRAMS.

A. Over-Decomposition and Variable Privatization

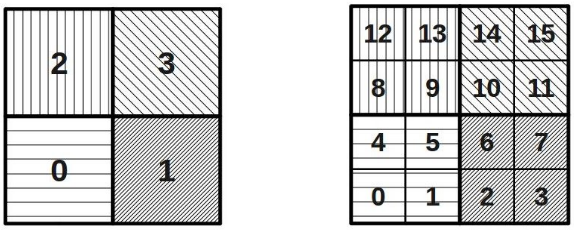
For a given MPI code to benefit from the advantages of AMPI, it must exploit processor virtualization. In this approach, the idea is to replace MPI's task decomposition with a new scheme that over-decomposes the same domain into a larger number of AMPI threads. Figure 2 illustrates that: on the left, we represent a regular domain decomposition with four MPI tasks, whereas the right side corresponds to a possible decomposition of the same domain into sixteen AMPI threads. With AMPI, there will be sixteen ranks, each associated to one thread. In this particular case, AMPI would have VP=16 and a resulting virtualization ratio of four. Other values of VP and virtualization ratio could be used as well. From the application's perspective, the execution with AMPI will behave similarly to an execution under sixteen MPI ranks.

Two issues have to be considered when using the over-decomposition scheme: (1) the application must have the property of binary reproducibility, which means that the numerical behaviour of the code is independent of the number of MPI ranks in use; and (2) the user has to consider the increase in memory usage due to the over-decomposition. This is because the amount of ghost cells and stack increases with the virtualization ratio. In our experiments, we measured an increase in memory usage. However, the benefits were still enough to justify the use of this technique.

In the example of Figure 2, AMPI will start the execution with threads $\{0,1,4,5\}$ mapped to the first processor, threads $\{2,3,6,7\}$ mapped to the second processor, and so on. As the execution progresses and the load balancer is invoked, threads may migrate across the four processors. Actual migrations depend on the observed behavior prior to load balancing and on the policy of the particular load balancer in use. Given the iterative behavior of BRAMS, a natural place to invoke the load balancer is between a certain number of timesteps in the simulation. To perform that invocation at every K timesteps, we simply added the following line at the end of the main loop in the BRAMS source code:

```
if ( mod(iteration,K) == 0 ) call MPI_Migrate()
```

When using the AMPI decomposition shown on the right of Figure 2, four threads share the same physical processor. As we observed in Section-III, this may create problems for global and static variables. On platforms that support the ELF format, AMPI provides a build-time flag that can automatically handle the privatization of global variables. This flag (*-swapglobals*) ensures that each thread will have its own version of a given global. At thread context-switch time, those versions are automatically switched by the Charm++ runtime system. Unfortunately, this method does not work for static



(a) 4 MPI tasks (b) 16 AMPI threads

Fig. 2: Domain decomposition

variables. A possible workaround is to create a module, insert all static variables into that module, and replace their original declarations in the source code. This scheme would effectively transform the statics into globals, which could then be handled with *-swapglobals*.

While the *-swapglobals* scheme effectivelly privatizes global variables, it may not be very efficient in some cases, because it makes the time of thread context switch proportional to the number of globals in the code. The reason for this fact is that the *-swapglobals* flag forces the code to be compiled as shared library. Consequently, the linker creates a global offset table (GOT) that contains pointers to all global variables in the application. To privatize globals, at context switch the Charm++ runtime system changes every entry of the GOT to the corresponding data of the resuming thread. In some applications, however, the number of static and global variables is very large. Table I shows those numbers for BRAMS and for WRF, another popular weather forecasting code. For codes like these, with thousands of globals and statics, the context switch time becomes excessively large, as many operations are needed to update the GOT.

To eliminate this context switch overhead, we developed a new privatization strategy [21] based on Thread-Local Storage (TLS). TLS is employed by kernel-level threads to privatize data. It is used, for example, to implement the C directive *__thread* in compilers. Because there is no such directive in Fortran, we modified the gfortran compiler to adopt the TLS scheme for all global and static variables. We also modified the Charm++ runtime system such that TLS storage implied using user-level threads, rather than kernel-level threads. These changes are generic and remain useful for any other Fortran code that needs to be used with AMPI. The TLS-based strategy has two advantages: (a) it makes context switches O(1) and (b) it also privatizes static data. For all experiments reported in this current paper, we used this TLS strategy.

Model	Globals	Statics	Commons
BRAMS	10205	519	32
WRF-v.3	8661	550	70

TABLE I: Number of global and static variables in two meteorological models.

B. New Load Balancer in Charm++

Charm++ provides a series of load balancers implementing different balancing policies. As already explained, those balancers can be applied to any AMPI program via the trivial insertion of the *MPI_Migrate()* call; no other application changes are required. Because they were designed to be general-purpose, most of the existing Charm++ balancers consider only computational load or communication traffic in their balancing decisions. This generality favors wide applicability, but misses opportunities for optimizations in particular cases.

In a typical BRAMS execution under MPI, a processor associated to a given domain exchanges information with the processors of domains around it at each timestep. This communication occurs to allow the processors to exchange data near the domain borders. Several atmospheric variables are contained in those exchanges. There are also broadcasts and reductions for I/O, but those operations are much less frequent; they occur at the beginning of the execution and between a certain number of timesteps. In general, these I/O operations do not affect the code's performance.

The existing set of load balancers in Charm++/AMPI includes two balancers that could, in principle, be useful in BRAMS: *RefineCommLB* and *RecBisectBfLB*. The first balancer (*RefineCommLB*) moves threads away from the most overloaded processors, taking communication into account as well; it limits the number of threads that actually migrate. Meanwhile, *RecBisectBfLB* uses the thread communication graph to recursively partition the threads, according to their observed loads, until there is one partition for each processor. Hence, both balancers consider computational load and communication traffic. However, as we show in the next section, none of these balancers fits well the two-dimensional spatial domain decomposition of BRAMS.

We developed a new Charm++ balancer based on the recursive bisectioning idea. We place the various threads with their loads into a two-dimensional Hilbert space-filling curve [24], and then recursively bisect that curve until the number of segments is equal to the number of processors. Because the Hilbert curve preserves spatial locality, threads corresponding to sub-domains that are close in space are likely to be assigned to the same processor. This implies that a significant amount of communication between threads will be local to the same processor, which benefits application performance. Figure 3 shows the Hilbert curve for the example of Figure 2.

V. EXPERIMENTAL RESULTS

In this section, we evaluate the impact of applying processor virtualization to BRAMS. We divided our tests in two parts: first, we simply varied the virtualization ratio, without introducing any thread migration; next, we used migration to balance the load across processors. For both cases, we conduct a quantitative evaluation of the overall application performance and of the factors leading to that observed performance.

All our tests were performed on a Cray XT5 system (*Kraken*), whose nodes have two 2.6GHz six-core AMD Opteron processors. This machine uses the Cray SeaStar2+ network to connect its nodes. As our test case, we employed a grid with 40 vertical levels and 512×512 points for a region in the southeast of Brazil (the same region depicted in Figure 1a), corresponding to a horizontal resolution of 1.6 Km. We conducted four-hour forecasts in BRAMS with a timestep of 6 seconds.

A. AMPI Virtualization Effects

We started our experiments by analyzing the impact of using solely processor virtualization in BRAMS, i.e. we simply varied the number of virtual processors for executions on a fixed number of physical processors. As the number of virtual processors increases, some overhead is expected to occur, because over-decomposition also means that control code (e.g. ghost zone exchange) will increase as well. On the other hand, a virtualized execution can benefit from automatic overlap of computation and communication, even without explicit use of nonblocking MPI calls: when a certain virtual processor blocks on a receive, another virtual processor can execute. In addition, this approach allows for better cache use, because each sub-domain is smaller than it would be in a non-virtualized environment. Consequently, these smaller sub-domains can more easily fit in cache.

Using 64 physical processors, we conducted executions of BRAMS with AMPI employing, respectively, 64, 256, 1024 and 2048 virtual processors. These executions corresponded to virtualization ratios of 1, 4, 16 and 32, respectively. The mapping of virtual processors to physical processors was in a blocked fashion similar to what had been shown in Figure 2b.

Table II shows the results of these experiments. As it can be seen from this table, the execution time decreases 22.4% with 256 virtual processors. The decrease with 1024 virtual processors is slightly better: 25.3%. However, with 2048 virtual processors the reduction is only 10.7%. Therefore, there seems to exist a stagnation point beyond which adding more virtual processors does not improve performance.

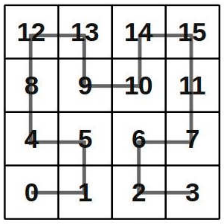


Fig. 3: Hilbert curve for the case of 16 threads

Configuration	**Execution Time (s)**
No Virtualization	4970.59
256 virtual processors	3857.53
1024 virtual processors	3713.37
2048 virtual processors	4437.50

TABLE II: BRAMS execution time on 64 processors

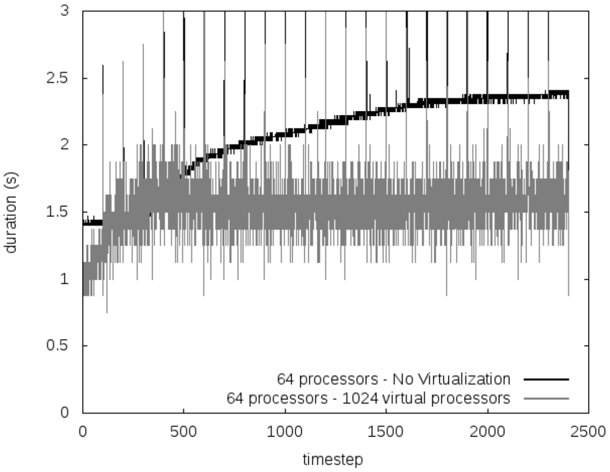
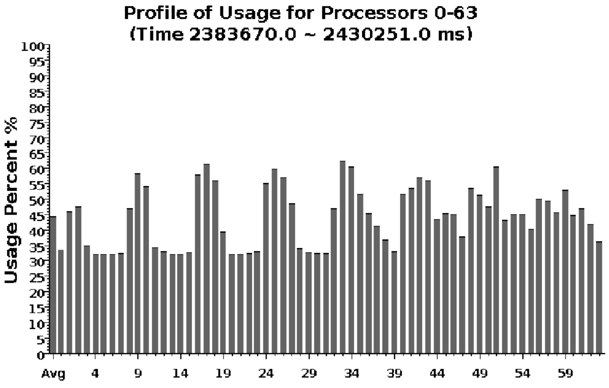


Fig. 4: Effect of virtualization on BRAMS performance

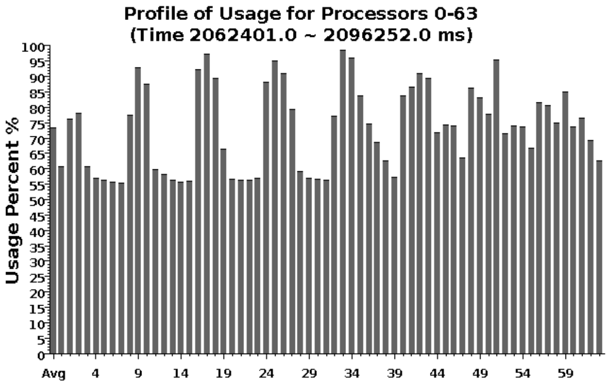
Figure 4 compares the performance of two of those executions, one without virtualization and another with a virtualization ratio of 16. Both executions present peaks that occur with a period of 100 timesteps: these correspond to timesteps in which radiation modeling is active. For the virtualized execution, there is a wide "amplitude" in the observed duration of the timesteps; this occurs because those timing measurements are made on thread zero. In the virtualized execution, that thread shares the physical processor with fifteen other threads. The order of thread execution is determined by the Charm++ scheduler, and may vary across the simulation. Hence, thread zero's slot of execution fluctuates as the simulation progresses.

To determine the reasons for the performance improvements as we raise the number of virtual processors, we compared the cases of 256 and 1024 virtual processors, respectively, to the non-virtualized configuration. We enabled the automatic Charm++ instrumentation to capture detailed performance data during a section of the simulations (i.e. between timesteps 1250 and 1270), and analyzed the obtained data with our *Projections* performance analysis tool [25].

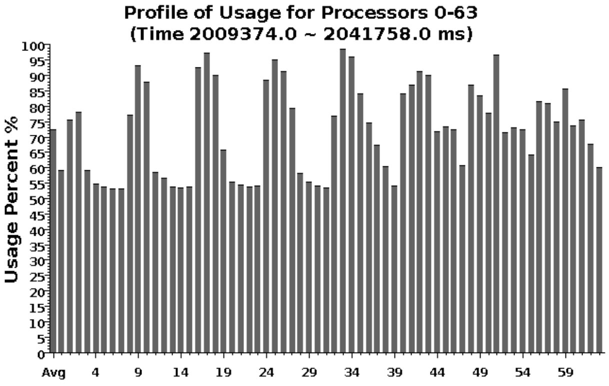
Figure 5 shows CPU usage for the various configurations. The bars represent the amount of CPU used during the



(a) 64 processors - no virtualization



(b) 64 processors - 256 virtual processors



(c) 64 processors - 1024 virtual processors

Fig. 5: CPU utilization for various virtualization ratios

measured period. There is one bar for each physical processor and the first bar is the average CPU usage. Without virtualization (Figure 5a), the average CPU usage was 44%. This CPU use is fairly low and could indicate that the subdomains were too small. However, this experiment represents a typical size of a BRAMS simulation, where each subdomain has $64 \times 64 = 4096$ columns of the atmosphere and the ghost-zone contains only 256 more columns. Furthermore, the experiments were run on a machine with a fast interconnection. Therefore this CPU usage is well representative of what an average user would experience with this application.

When we use four virtual processors per real processor (Figure 5b), the average CPU usage improves to 73%. The reason for this improvement is that the waiting time that each processor would experience is filled with computation of other virtual processors that are ready to execute. As we increase the virtualization degree further to sixteen virtual processors, however, we do not see any additional improvement (Figure 5c). In this case, the average CPU usage is still 73%. This is because the bottleneck is no longer idle time, but load imbalance (notice that some processors in Figure 5b were already near 100% utilization). Nevertheless, this higher degree of virtualization allows for more flexibility when using migration for load balancing. The average CPU usage for the case of 2048 virtual processors (not shown here) is also 73%. As we will show, another reason must exist to explain the performance loss for this case in Table II.

We also analyzed cache utilization, by reading the hardware performance counters of the AMD processors. We used the Performance Application Programming Interface (PAPI) library [26] to access those counters. While using this library in a non-virtualized environment is trivial, one cannot use the same instrumentation in a virtualized execution, because the runtime system does not guarantee that the threads (virtual processors) are executed in an appropriate order. Therefore, we developed a scheme to ensure that the first thread entering the code section started the PAPI counters and the last thread leaving that section read those counters. Hence, our measured values account for the execution of all threads on a given processor. We used global variables forced to be shared among threads to control this scheme.

The performance gains of the cache should arise from the Brams code structure. In a time-step of Brams, similarly to other meteorological models, the various physical processes are called in sequence. Each of those processes performs its associated computation for the entire local sub-domain, hence the second physical process can benefit from the fact that the sub-domain is still in cache due to the computations of the first physical process. This effect repeats for the remaining physical processes within the same time-step. This performance gain is maximized when the local sub-domain matches the size of cache.

The measured amounts of cache misses for the period corresponding to the timesteps of interest in the application are presented in Table III. A consistent decrease in cache misses can be seen in both L2 and L3 caches for the cases of 256 and 1024 virtual processors. That confirms the improvement in spatial locality that virtual processors allow.

The cache misses for the case of 2048 virtual processors, however, increased. The reason for this increase, and the corresponding increase in execution time observed in Table II,

Configuration	L2 cache misses	L3 cache misses
No Virtualization	12,416M	8,448M
256 virtual processors	10,560M	4,416M
1024 virtual processors	9,408M	3,904M
2048 virtual processors	13,696M	5,056M

TABLE III: Total number of cache misses on 64 processors in BRAMS

is that the sub-domains for this case are too small and cannot benefit from all cache space available. Since context switch among threads occurs in this virtualized execution, the data of a sub-domain cannot stay in cache for a long period, because it has to give room for data from other virtual processors. The same fact happens with 256 and 1024 virtual processors, but in those cases more cached data is used between context switches. We confirmed this by running a new experiment with 1024 virtual processors but using only 32 physical processors. The resulting numbers of L2/L3 cache misses were 9,372M and 4,038M, respectively. Those numbers are very close to the original results with a virtualization ratio of 16, despite employing a new ratio of 32. Hence, we can conclude that the lower cache utilization measured in the last row of Table III is due to the smaller sub-domain size combined with the use of virtualization. In summary, there is a sweet spot in performance that is reached when the size of the sub-domain assigned to each virtual processor best matches the underlying cache sizes, in particular the size of the L3 cache, which accounts for the most expensive misses on the AMD Opteron.

B. Migration for Load Balancing

In addition to the benefits presented in the previous section, the virtualized implementation of MPI enables migration of virtual processors. This feature allows users to perform load balancing by migrating certain virtual processors from more loaded processors to less loaded ones. The user must still choose (or develop) a strategy that will coordinate migrations.

To evaluate the effectiveness of some load balancing strategies, we present in this subsection the experimental results of a forecast that has a localized thunderstorm. This thunderstorm causes an imbalance that delays the execution of the entire forecast. We started our tests with two existing Charm++ load balancers described in section IV-B: *RefineCommLB* and *RecBisectBfLB*. Since migration costs in BRAMS may be high due to the code's large memory footprint, we chose to test *RefineCommLB* because it favors fewer migrations [22]. *RecBisectBfLB*, in turn, partitions the threads according to their loads and attempts to maintain in the same partition threads that interact with each other; this partitioning scheme should respect the communication pattern existing in BRAMS.

In addition to those two load balancers, we also tested the newly developed balancer based on Hilbert curves (*HilbertLB*). That balancer requires a mapping of threads into the Hilbert curve and a recursive cut algorithm. Both the cut algorithm and the mapping have an efficient implementation [27].

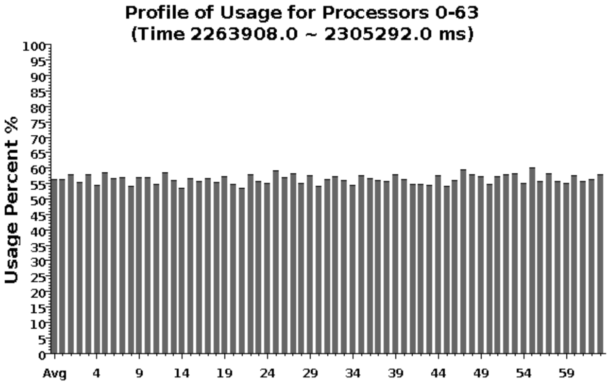


Fig. 6: CPU usage with *RecBisectBfLB* balancer

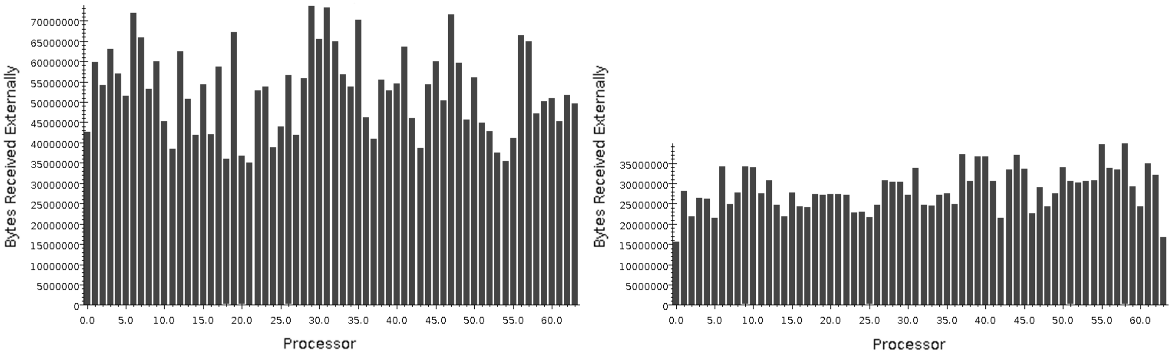
We inserted calls to the load balancer in BRAMS every 600 timesteps (i.e. at every hour of simulated time). In our test case, the thunderstorm happens in the southeastern region of Brazil, as shown in Figure 1a. During the four-hour forecast, we observed that the thunderstorm moved slightly to the south.

Table IV compares the execution times of five different types of configurations, including different load balancers. The *Speedup* column corresponds to the speedup with respect to the non-virtualized case. According to Table IV, the balancer that best improved performance was the one based on the Hilbert curve (*HilbertLB*). We analyzed some portions of these executions with *Projections*, to identify the reasons for the inefficacy of the *RefineCommLB* and *RecBisectBfLB* load balancers. Using *RefineCommLB*, a good load balance was not achieved. The resulting CPU usage (not shown here) was similar to that without any load balancer. This means that *RefineCommLB* did not migrate enough threads to neutralize the imbalance.

Meanwhile, *RecBisectBfLB* did produce a well balanced load, as shown in Figure 6. However, the CPU usage was quite low. One key reason for this fact is communication. If threads corresponding to neighbor sub-domains migrate to different processors, the cross-processor communication volume will be high, which hurts CPU usage. Although *RecBisectBfLB* considers the communication graph to rebalance load, it does not attempt to minimize communication across processors. Hence,

Configuration	Execution Time (s)	Speedup
No virtualization	4987.51	-
No load balancer - 1024 VP	3713.37	25.55%
RefineCommLB - 1024 VP	3714.92	25.52%
RecBisectBfLB - 1024 VP	4527.60	9.23%
HilbertLB - 1024 VP	3366.99	32.50%

TABLE IV: Load balancing effects on BRAMS (all experiments were run on 64 real processors)



(a) *RecBisectBfLB* balancer

(b) *HilbertLB* balancer

Fig. 7: Cross-processor communication volume in BRAMS with different load balancers

RecBisectBfLB does not necessarily partition the sub-domains into square blocks, which would minimize the communication between blocks for BRAMS. Conversely, *HilbertLB* keeps the sub-domains in approximately square blocks (see Figure 3) and consequently the external communication is kept low. This is confirmed by Figure 7, which shows the cross-processor communication volumes with *RecBisectBfLB* and *HilbertLB*. Due to the lower communication volume, *HilbertLB* achieves much better processor utilization, and also very good load balance, as demonstrated in Figure 8.

The overall performance of the BRAMS execution with *HilbertLB* can be seen in Figure 9. The timestep durations in this execution are significantly shorter than those observed in Figure 4, where no virtualization or no load balancing had been employed. The load balancing calls in Figure 9, in particular the first one at timestep 600, were quite effective in improving performance. Notice, though, that we should avoid calling the load balancer too often, since there is a cost for that call and for corresponding thread migrations: in the execution of Figure 9, timesteps 600/1200/1800 had durations of 24.6s, 14.7s and 12.2s, respectively. Those durations are much higher than in other timesteps of that execution. Hence, there is a relatively heavy cost in each load balancing call. Nevertheless, that cost is well amortized by the gains observed in subsequent timesteps, leading to the improved performance of Table IV.

VI. CONCLUSIONS

Although meteorological models typically use a regular three-dimensional grid and, in a parallel execution, each processor executes the same code, this type of application is subject to load imbalance. However, most models currently do not perform load balance, due to the complexity involved with this task. This paper described the use of processor virtualization in a weather model to simplify the adoption of load balancing strategies in MPI legacy codes. The amount of change to the original application is minimal.

Our approach relies on AMPI, a virtualized implementation of MPI, where the domain is decomposed in more sub-domains than available processors. Each sub domain is assigned to a virtual processor and each real processor handles a set of virtual processors. A virtual processor can migrate from a real processor to another in order to rebalance load. In our experiments, we found that just the over-decomposition is already beneficial for the employed application: we obtained up to 25.5% reduction in execution time. This reduction can be attributed to the overlap of communication and computation and to better use of the cache hierarchy.

Furthermore, by carefully choosing a load balancing strategy, we managed to further reduce the execution time by 7%, reaching a total gain of 32.5%. For the BRAMS meteorological model, the appropriate balancing strategy was one that takes into account the communication across the sub-domains. We used a Hilbert curve to map the 2-D domain decomposition to a 1-D space. The curve is then cut into segments so that each segment has approximately the same load. Due to properties of the Hilbert curve, the corresponding sub-domains on a given segment should be close in the 2-D space and, consequently, the cross-processor communication is reduced.

As an ongoing work, we are investigating strategies to determine the best moment to invoke the load balancer. Although simple, a fixed-rate approach may not be the most adequate: the processor loads may vary quite abruptly in some occasions, due to the movement of thunderstorms that cross the boundaries of regions assigned to some of the processors. Therefore, using a varying invocation scheme may produce better results. We are also analyzing if meteorological information, available in the model, could be used to further improve the balancing decisions that are taken when a load balance is performed.

REFERENCES

[1] J. Kinter III and M. Wehner, "Computing issues for WCRP weather and climate modeling," in *WCRP Modeling Panel*, Exeter, UK, 2005.

[2] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, "Programming Petascale Applications with Charm++ and AMPI," in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. Chapman & Hall / CRC Press, 2008, pp. 421–441.

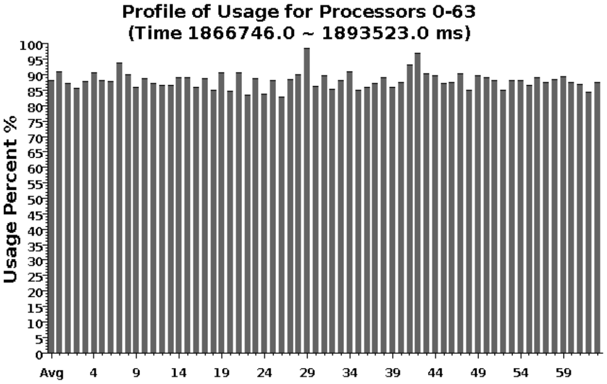


Fig. 8: CPU usage with *HilbertLB* balancer

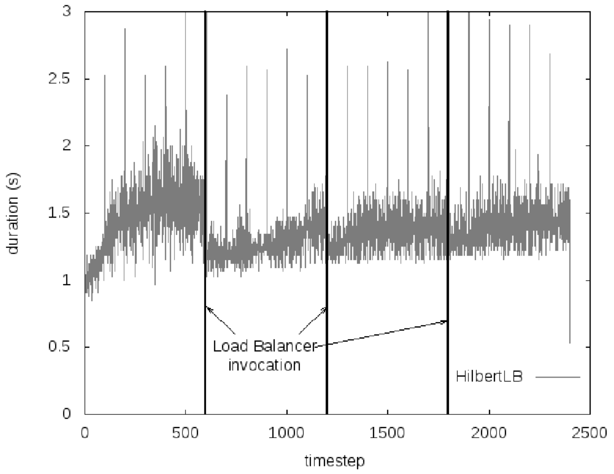


Fig. 9: BRAMS performance with *HilbertLB* balancer (1024 virtual processors)

[3] S. Ghan, X. Bian, A. Hunt, and A. Coleman, "The thermodynamic influence of subgrid orography in a global climate model," *Climate Dynamics*, vol. 20, no. 1, pp. 31–44, 2002.

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

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

[6] C. L. Mendes and J. Panetta, "Selecting directions for parallel RAMS performance optimization," in *Proceedings of the* 11^{th} *Symposium on Computer Architecture and High Performance Computing - SBAC-PAD*, Natal, Brazil, 1999, pp. 85–92.

[7] R. Walko, L. Band, J. Baron, T. Kittel, R. Lammers, T. Lee, D. Ojima, R. Pielke Sr, C. Taylor, C. Tague *et al.*, "Coupled atmosphere–biophysics–hydrology models for environmental modeling," *Journal of Applied Meteorology*, vol. 39, no. 6, 2000.

[8] G. Tripoli and W. Cotton, "The Colorado State University three-dimensional cloud/mesoscale model1982. Part I: General theoretical framework and sensitivity experiments," *J. Rech. Atmos*, vol. 16, no. 3, pp. 185–220, 1982.

[9] G. Grell and D. Devenyi, "A generalized approach to parameterizing convection combining ensemble and data assimilation techniques," *Geophysical Research Letters*, vol. 29, no. 14, pp. 38–1, 2002.

[10] R. Gevaerd, S. R. Freitas, and K. M. Longo, "Numerical simulation of biomass burning emission and trasportation during 1998 roraima fires," in *International Conference on Southern Hemisphere Meteorology and Oceanography (ICSHMO) 8*, 2006.

[11] S. Freitas, K. Longo, M. Silva Dias, R. Chatfield, P. Silva Dias, P. Artaxo, M. Andreae, G. Grell, L. Rodrigues, A. Fazenda *et al.*, "The Coupled Aerosol and Tracer Transport model to the Brazilian developments on the Regional Atmospheric Modeling System(CATT-BRAMS)- Part 1: Model description and evaluation," *Atmospheric Chemistry and Physics*, vol. 9, no. 8, pp. 2843–2861, 2009.

[12] S. Eliane, E. Araújo, W. Cirne, G. Wagner, N. Oliveira, E. Souza, C. Galvão, and E. Martins, "The SegHidro Experience: Using the Grid to Empower a HydroMeteorological," in *Proceedings of the First International Converence on e-Science and Grid Computing (e-Science/05)*, 2005, pp. 64–71.

[13] R. Souto, R. Avila, P. Navaux, M. Py, N. Maillard, T. Diverio, H. Velho, S. Stephany, A. Preto, J. Panetta *et al.*, "Processing mesoscale climatology in a grid environment," in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid–CCGrid*, 2007.

[14] A. L. Fazenda, J. Panetta, P. Navaux, L. F. Rodrigues, D. M. Katsurayama, and L. F. Motta, "Escalabilidade de aplicação operacional em ambiente massivamente paralelo," in *Anais do X Simpósio em Sistemas Computacionais (WSCAD-SCC)*, 2009, pp. 27–34.

[15] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[16] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. J. Martyna, "Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L," *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, vol. 52, no. 1/2, pp. 159–174, 2008.

[17] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively Parallel Cosmological Simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[18] X. Jiao, G. Zheng, O. Lawlor, P. Alexander, M. Campbell, M. Heath, and R. Fiedler, "An integration framework for simulations of solid rocket motors," in *41st AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, Tucson, Arizona, July 2005.

[19] 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.

[20] G. Zheng, O. S. Lawlor, and L. V. Kalé, "Multiple flows of control in migratable parallel programs," in *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*. Columbus, Ohio: IEEE Computer Society, August 2006, pp. 435–444.

[21] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, and C. L. Mendes, "A new technique for data privatization in user-level threads and its use in parallel applications," in *ACM 25th Symposium On Applied Computing (SAC), Sierre, Switzerland*, 2010.

[22] G. Zheng, "Achieving high performance on extremely large parallel machines: Performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[23] A. Bhatelé, L. V. Kalé, and S. Kumar, "Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications," in *23rd ACM International Conference on Supercomputing*, 2009.

[24] D. Hilbert, "Über die stetige abbildung einer linie auf ein flächenstück," *Mathematische Annalen*, vol. 38, pp. 459–460, 1891.

[25] 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.

[26] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Proceedings of Supercomputing'00*, Dallas, Texas, 2000.

[27] X. Liu and G. Schrack, "Encoding and decoding the Hilbert order," *Software-Practice and Experience*, vol. 26, no. 12, pp. 1335–46, 1996.