Selecting Directions for Parallel RAMS Performance Optimization

Celso L. Mendes[1] , Jairo Panetta[2]

[1] INPE / LAC
Av. dos Astronautas, 1758, São José dos Campos, SP, Brazil
{celso@lac.inpe.br}
[2] INPE / CPTEC
Rod. Presidente Dutra, Km 40, Cachoeira Paulista, SP, Brazil
{panetta@cptec.inpe.br}

Abstract—
This paper presents the results from a preliminary performance evaluation of parallel RAMS, a numerical weather prediction model designed to simulate atmospheric phenomena at a regional level. The main goal in our work was to study in detail the performance of the current RAMS version, and to uncover aspects of its code where opportunities for optimization exist. We present our observations on both computation and communication performance of RAMS executing on a distributed memory parallel platform, and analyze their contributions to total program performance. From the observed data, we present simulations that predict bounds on potential performance gains for possible load balancing strategies.

Keywords— **Performance analysis, Numerical weather prediction, Performance measurement.**

I. Introduction

Finding optimization opportunities in large scientific software codes is an enterprise as old as writing the code itself. There are many reports of successful optimizations published in the open literature. But how does one select where to optimize, among a set of potential opportunities? In fact, to evaluate optimization gains before implementation is a very hard task.

In this paper, we describe the mechanisms used to select optimization directions for a large, commercially available regional weather prediction production code, the Regional Atmospheric Modeling System (RAMS). This effort is being conducted as part of the FINEP-RAMS project, a project sponsored by FINEP (Financiadora de Estudos e Projetos), an agency of the Brazilian Ministry of Science and Technology (MCT).

The FINEP-RAMS project, being executed by Elebra Systems, aims to produce an improved version of parallel RAMS. Improvements will occur in many directions, including new meteorological functionalities, better documentation, improved coding practices and strategy, and better speedups on parallel systems with up to 64 processors. In this paper, we limit our presentation to just one of these aspects, namely speedup improvement.

The current parallel RAMS implementation is targeted to a modest number of processors. It is immediate, once the parallel computation is understood, to devise optimization opportunities that result in better speedups for the devised range of processors. The hard task is to choose which opportunity to pick.

We describe a mechanism to evaluate opportunites, composed of an instrumentation scheme with minimal intrusion, performance data sets obtained from instrumented code execution, and simulation procedures that, in some cases, anticipate gains in optimization candidates. This set of techniques permits a higher level of confidence in achieving project goals than the usual round-table discussion.

The remaining of this paper is organized as follows. In §II, we comment related work. We analyze RAMS performance in a global form in §III and in much deeper detail in §IV. §V presents promising candidates for optimization. Finally, we conclude and comment on future work in §VI.

II. Related Work

Since the introduction of vector machines, in the 60's, implementation of numerical weather forecasting models has been one of the major applications of high performance systems. Traditionally, most of those applications were based on global models [DRA 95]. More recently, however, regional models, like RAMS, emerged as an interesting alternative to improve the accuracy in the forecast for a limited region [BAI 95].

Like in other scientific applications, there are a number of computational issues involved in the parallel implementation of regional models similar to RAMS. Load balancing is one of such issues: Schälter and Krenzien quantify the load imbalance in model physics in the Deutschland Model [SCH 97]. Michalakes designed and implemented dynamic load balancing in the parallel version of MM5 [MIC 95]. Although the RAMS code has been designed with load balancing facilities in mind [TRE 97], there is no such support in its current version.

Portability has also been a main target for various designs. Some authors discuss software techniques aimed at enhanc-

ing portability of atmospheric codes, especially across parallel architectures [SAT 97], [WAL 97]. This includes the ability to maintain a single model to run on diverse computing platforms. In this regard, RAMS has fully achieved its goal: it is available as a single source code, which can be used both on sequential and on parallel systems from a variety of vendors.

Because high performance is the main motivation for the use of parallel systems, performance analysis is an active area of research in the parallel computing community. While the current state of the art comprises real-time performance optimization of heterogeneous systems [REE 98], post-mortem analysis techniques remain important to provide the user with valuable information about program execution on existing homogeneous systems. The instrumentation and analysis techniques that we developed in this work were based on our experience with the Pablo system [REE 93]. Although Pablo provides a powerful infrastructure, our technique attained much lower instrumentation overhead, and fully met our requirements of exposing critical performance issues in RAMS.

III. Total Program Performance

In this work, we analyzed RAMS version 4.26. This version comprised a complex set of procedures written mostly in Fortran77, with a few routines in C. The part of the code where numerical integration is performed contains more than 36,000 lines of source code, spread across more than 50 files. The original program was built to run on parallel machines with either the PVM or MPI communication environments. In our experiments, we have always used MPI only.

The actual configuration of the desired execution is implemented through a configuration file previously created by the user. This ascii file, named RAMSIN, defines the data set, size of the grid representing the atmosphere, integration timesteps, etc. It is also in this file that the user selects which physical phenomena will be simulated. There are flags, for example, to control the extent of cloud microphysics simulation.

A. Code Structure

The parallel RAMS code has a *master-slave* structure: there is a master process that does the initialization, reading input data and dividing the work among slave processes. At the end, the master collects results from slaves and writes the data to output files.[1] Slave processes perform the numerical integration, simulating the state of the atmospherical entities (temperature, wind, etc.) at each point of a 3-D grid representing the atmosphere. The simulation is conducted for a certain number of timesteps, as programmed by the user in the RAMSIN control file.

[1] The recording of partial results is also available, and is selected by appropriate commands in the RAMSIN file.

TABLE I
NUMBER OF GRID COLUMNS ASSIGNED TO SLAVE PROCESSES

Number of Slaves	Smallest	Greatest	Ratio Great./Small.
1	6084	6084	1.00
2	3042	3042	1.00
3	2028	2028	1.00
4	1521	1521	1.00
5	1175	1269	1.00
6	975	1053	1.08
7	825	897	1.09
8	725	810	1.12
9	625	729	1.17
10	575	648	1.13
11	525	580	1.10
12	475	540	1.14
13	427	513	1.20
14	399	460	1.15
15	375	440	1.17

Parallelization of the work across processors is implemented in a domain-decomposition fashion: each slave process is assigned to a 3-D sub-grid above a given area on Earth's surface. In the current RAMS version, such areas are rectangular. RAMS tries to ensure that the various processes receive sub-grids with nearly the same number of vertical columns; this is difficult to achieve in some cases, causing potential performance problems, as we will show.

Assuming a grid with $80\times80\times35$ points, Table I shows the resulting smallest and greatest numbers of vertical columns assigned by RAMS to any slave process, as a function of the number of slaves. As one can see, the difference between the numbers of assigned grid columns is often larger than 10%, and can be as large as 20%. This means that a given slave will receive 20% more columns to process than another one.

B. Speedup

An important feature of any parallel program is how performance improves as more processors are added. In RAMS, we always map each master or slave process to an exclusive processor; thus, the processors do not have to be timeshared between RAMS processes. Because all the program I/O is performed solely by the master, and occurs only at a few points across program execution, the slaves spend most of their processing time computing or exchanging data among them. Figure 1 shows RAMS speedup on PAD,[2] for a data set comprising an $80\times80\times35$ grid and an integration period of six hours, corresponding to 480 timesteps.

The speedup curve for the current RAMS version shows good performance for small numbers of processors. There

[2] The PAD system (*Processador de Alto Desempenho* [BER 99]), manufactured by Elebra, is a Beowulf class parallel machine, comprising dual Pentium-II boards interconnected by Myrinet.

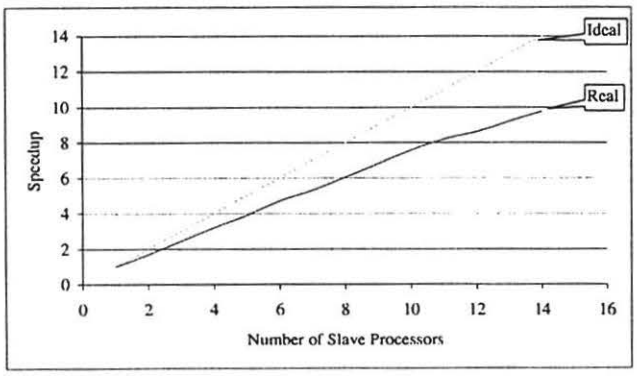

Fig. 1. RAMS Speedup on PAD

is, however, a decrease in efficiency as the number of processors grows. This indicates that we might have a serious performance problem when we port the program to system configurations with larger numbers of processors, as originally intended.

IV. Detailed Performance Analysis

Understanding the main causes of inefficiency is essential to optimize a parallel program and achieve good scaling with the number of processors. However, for a complex code like RAMS, there are many optimizing possibilities, comprising varying degrees of both implementation effort and improvement return. Selecting the *best* aspect to be optimized becomes nearly impossible without a detailed analysis of program behavior during a real execution.

In the process of analyzing RAMS performance, our first step was to conduct a series of experiments where we instrumented the code and measured its performance under varying system configurations.

A. Instrumentation Infrastructure

There are many instrumentation packages available, with different levels of capability and portability [DER 98], [HOL 96]. Many of these packages, however, either are limited to specific platforms or languages, or employ techniques that are more suited to systems with very high numbers of processors. Our emphasis was to keep instrumentation intrusion low, yet being able to observe RAMS behavior in detail on systems with small and medium numbers of processors. Thus, we decided to build our own instrumentation infrastructure, such that we could make it as capable as we needed and still meet our low intrusion requirements.

We developed a generic instrumentation tool that is able to track computation and communication events during execution of the underlying program. The current version of our tool operates completely by software, although we envision expanding it to use hardware monitoring resources available in modern microprocessors. By using MPI's profiling interface [PAC 97], we are able to trace communication events without changing a single line in the source program; all that is required to create an instrumented executable is to relink the RAMS object files.

For code fragments with pure computation, we used a facility that already existed in RAMS (structure *acctimes*), which accumulates the time spent inside each meteorological functionality for every simulation timestep. All we had to change in the RAMS source code was to add one line, at the end of the timestep, with a tracing call to record the set of such durations.

In our tracing scheme, each processor produces an individual trace file that can be processed after the end of execution. We developed scripts and programs that read those trace files and compute various statistics about the execution. We also plan to create programs to transform these traces into the format of other existing performance analysis tools, such that we can use the visualization facilities of those tools in the future.

To keep intrusion as low as possible, our tool stores trace data in a memory buffer, and postpones writing to the trace file until the end of the execution, or until the buffer gets full. Practical experience with RAMS has shown that even with moderate buffer sizes (around 2 MBytes), dumping a full buffer to disk prior to the end of execution rarely happens. Also, our expectations of low tracing overhead have been fully achieved: in all conducted experiments, an instrumented execution took at most 0.7% more elapsed time than the corresponding execution without instrumentation.

B. Observed RAMS Performance

We started our experiments by executing an instrumented version of RAMS on six PAD processors, with one processor running the master process and five processors running slave processes. We used the same data set as before, with a grid of $80 \times 80 \times 35$ points and a simulation of six hours. After executing the program and obtaining the corresponding traces, we processed them to extract performance data.

Figure 2 illustrates the type of information produced by one of our analysis programs. It shows communication data collected from the trace of the first slave process (running on processor P1). In this figure, for each MPI function, we present (in parenthesis) the number of times the function was invoked, and the accumulated time inside that function for the complete execution.

In the RAMS code, communication between a slave and the master is implemented with blocking MPI functions (MPI_Send,MPI_Recv), while communication among the slaves is implemented with nonblocking functions (MPI_Isend,MPI_Irecv) followed by MPI_Wait. Thus, one can see from the data in Figure 2 that most of the communication time in this processor corresponds to waiting for messages from slave P2. In fact, slave processor P1 exchanges

```
Trace File: Trace_1.bin
Total Times (sec):
 MPI_Send (570) : 2.272128
 MPI_Recv: (182) 23.342901
 MPI_Isend: (5760) 2.088392
 MPI_Irecv: (5760) 0.134289
 MPI_Wait: (17280) 97.233113
     Wait-dummy: (5760) 0.082852
     Wait-isend: (5760) 0.079963
            to P2: (2880) 0.049684
            to P4: (2880) 0.030279
     Wait-irecv: (5760) 97.070298
        By Source-Processor:
          from P2: (2880) 93.010456
          from P4: (2880) 4.059842
        By Msg-Tag:
          tag=20001 (960) 0.443949
          tag=11000 (960) 51.792250
          tag=10004 (1920) 37.097457
          tag=10005 (960) 0.533148
          tag=10006 (960) 7.203494
 MPI_Barrier: (4) 29.912332
 Total Prog: 1025.367786
```

Fig. 2. Information extracted from RAMS trace.

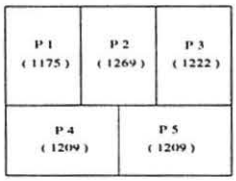

Fig. 3. RAMS domain decomposition with 5 slaves.

messages with two neighbors, P2 and P4, as indicated by the domain decomposition in Figure 3. This figure also shows the number of grid columns assigned by RAMS to each slave.

Using another analysis program, we can inspect how much time the slave processors spent on the computation of meteorological functionalities. We can do this both in a functionality by functionality basis, or in a collective form for all functionalities. Table II shows the five functionalities with the largest time contributions, as well as the total computation time for all functionalities. As expected, slave processor P2, which received the largest number of grid columns (see Figure 3), had the greatest computation time. Meanwhile, slave processor P1, containing the smallest number of grid columns, had the smallest computation time.

One of the most important uses of trace data is to analyze the dynamics of program execution. As an example, we can check how the total computation times, presented at the bottom of Table II, are divided along the execution. Figure 4 presents the computation times for the slaves at every timestep of the integration. This same kind of analysis can be carried out for any of the individual functionalities. By doing that, we found that the peaks in Figure 4 are due to the RADIATE functionality: it is activated only at a few timesteps, as programmed by the user in RAMSIN; when it is active, however, it has a duration that is larger than any other functionality.

TABLE II
RAMS COMPUTATION TIMES (SECONDS) FOR SLAVE PROCESSORS.

Functionality	P1	P2	P3	P4	P5
ACOUSTIC	172.33	190.90	178.13	179.61	179.04
DIFFUSE_2	172.41	186.53	168.73	178.61	178.97
ADVECTv	145.33	162.56	149.49	151.28	150.77
ADVECTs	85.43	94.11	87.24	91.02	90.91
DIFFUSE_1	63.60	69.63	61.68	67.17	66.95
All Functionalities	866.97	945.20	872.22	902.41	901.18

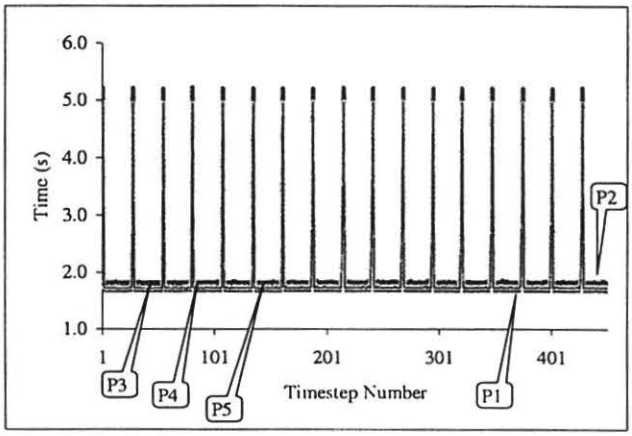

Fig. 4. Computation times across RAMS iterations.

C. Data Set Variations

Our next step in observing RAMS performance was to modify the input data set. In particular, we decided to request a higher level of microphysics analysis employed by the program. With that higher level, a wider variety of physical phenomena is considered during the simulation.

Execution of RAMS with the same grid as before, under the new microphysics level, produced the data presented in Figure 5. These plots show the computation times for each slave processor.[3] There are at least two features in these plots that make them remarkably different from the data in the lower part of Figure 4: the shape of the curves and their relative positions along the execution.

Because of microphysics processing, all the slave computation times in Figure 4 grow in the first half of the execution. After that, most of them tend to assume a nearly constant value, although at a higher level than in Figure 4. The relative positions of some of the curves change across the execution. Meanwhile, one of the slave processors (P3) remains always much below the other slaves.

All these varying features clearly indicate that micro-

[3]The computation times in Figure 5 include only functionalities that are active on every timestep. Thus, functionalities like RADIATE are *not* represented.

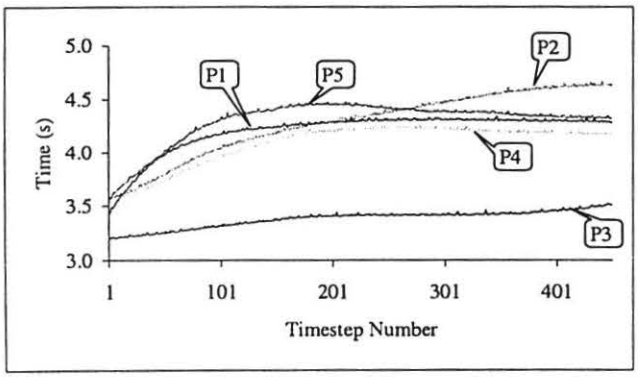

Fig. 5. RAMS computation times with microphysics.

physics processing introduces a degree of variability in the execution. There are significant differences between the computation times of certain slaves (of more than 20% in some cases). Those differences are expected: regions of the atmosphere containing clouds will require more processing work than regions where no clouds exist. This fact represents a potential source of load imbalance between processors, as confirmed by Figure 5 . We will return to this point in §V-C.

V. Directions for Optimization

Given the observed RAMS behavior, we now show possible directions to be followed as an attempt to optimize its performance.

A. Master Process

We start the exploitation of optimization directions by understanding how the master process spends its execution time. Table III shows the components of master's execution time for two configurations of the input data set and for five and fourteen slave processes. Configurations labeled "Regular" are the ones with a lower level of microphysics. Configurations labeled "Micro" stand for a higher level of microphysics.

Master's computation starts by reading problem data and computing domain decomposition (Table III row "Initialization"). It follows by sending to the slaves problem data and the domain partition they should compute (Table III row "Master sends data"). In sequence, the master waits while slaves compute during timesteps that do not require output (Table III row "Slaves compute"). If the current timestep requires output (a user specified input argument), then the master collects slave's results, outputs them to appropriate files and restarts slaves computation (Table III rows "Slaves send results" and "Intermediate I/O"). At the end, the master outputs final results and finalizes the computation (Table III row "Final data/trace I/O"). Master-slave communication at input/output timesteps is barrier synchronized (Table III row "Barrier synch").

TABLE III
Components of RAMS master execution time, in seconds.

Phase	5 Slaves		14 Slaves	
	Regular	Micro	Regular	Micro
Initialization	26.83	52.38	26.70	47.53
Master sends data	2.30	2.97	5.54	5.83
Barrier synch	0.04	0.04	0.08	0.08
Slaves compute	967.22	2317.22	399.00	897.32
Slaves send results	2.46	4.10	1.20	3.07
Intermediate I/O	24.04	87.25	25.09	43.30
Final data/trace I/O	22.70	51.82	42.86	62.89

Table III accounts exactly for all master's execution time in all cases. Instrumentation interference in execution times is restricted to the final row, where trace output is performed. Due to this interference, we neglect the values presented at that row.

There are two clear messages from Table III. First, execution time is dominated by the slaves computation (minimum of 86,5%, maximum of 94.3% over all cases). Consequently, that is where optimization efforts should be centered. The second most important factor is input/output (minimum of 4.7%, maximum of 10.6%), which is entirely performed by the master.

Optimization of RAMS input/output execution time is, for us, an open problem. One obvious solution is to use parallel I/O. But there are serious doubts that a portable and efficient (across all target machines) implementation exists.

To improve RAMS efficiency without loosing portability, an intermediate solution to the I/O problem is to overlap slave computation with master I/O. That requires detaching the slave computation from the master computation as much as possible. A solution that increases slave's intelligence is currently under study. Its impact on the execution time clearly increases with the number of processors, since it alleviates the master-slave dependency.

B. Slave Processes

We now concentrate on slave computations, trying to explain the numbers at row "Slaves compute" in Table III. We conducted a detailed analysis using trace files from slaves. From those files, we extracted computation timings and selected the events where slaves communicate with each other and wait for actions of other slaves.

The summation of computation times plus communication and waiting, in the traces from slaves, accounts for 99.76% to 99.93% of the "Slaves compute" row in Table III. That is impressive, since slave and master traces are independent. It also shows that we can confidently explain the data in Table III by just looking at slave behavior. For example, in the case of five slaves with high microphysics, our slave traces account for 2315.7 of the 2317.2 seconds previously reported in Table III.

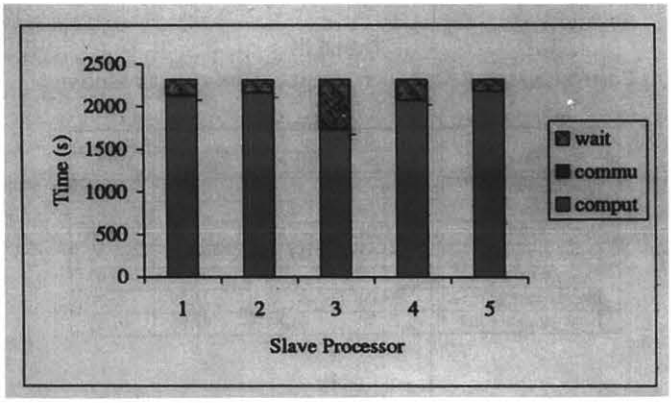

Fig. 6. Components of RAMS execution time with high microphysics.

Figure 6 shows how each slave spends its execution time, in the case of five slaves with high microphysics. Again, this is the part of the slave's execution time that is independent of the master's execution. For example, the time where slaves wait while the master performs I/O is not reported.

Each slave spends most of its time computing, and then, waiting for data originated at neighbor slaves. Communication time is negligible – it averages 0.22% of the accounted time. This is a classical case of load imbalance. The variation of computing times among the processors shows that load is far from beeing perfectly distributed. Based upon this analysis, a better load balancing strategy is a strong candidate for program optimization.

C. Load Balancing

We carefully studied the strategy to improve load balancing. A better static load balance strategy was first considered, since it is easier to implement than dynamic load balance. But Figure 5 shows that the execution time of each slave processor varies with the timestep. That is due to the dynamic nature of the meteorological phenomena being modeled, as we explained before. Since one cannot predict where the heavy load will be located as the computation evolves — one should know where the meteorological phenomena that causes the load will go, and that is exactly what RAMS computes — it is impossible to predict the best possible static load distribution scheme. Consequently, one should consider dynamic load balancing.

But what are the gains in each case? Is it worth all the effort to implement dynamic load balancing? To answer these questions, we developed a program that simulates the execution time behavior of RAMS slaves. It predicts the wall clock execution time of every slave at each timestep, given the computation time of each slave at each timestep and the slave's communication graph. It is a simple calculation: the wall clock time at each slave at the end of a timestep is the wall clock time of this slave at the end of the previous timestep, added with computation time of this slave at the current timestep, plus a delay due to load imbalance. The load imbalance delay is how long this slave waits, at the end of a given timestep, for data from other slaves that are not ready to send data yet (because they are still computing, that is, their wall clock value at the moment of data transmission is greater than this slave's current wall clock value). The communication graph stores information about which slaves communicate with any given slave.

This simulation procedure is a grossly simplified model of the slave's execution time. In RAMS, communication among slaves is spread over the timestep; in our model, communication among slaves is deferred to the end of the timestep. In RAMS, two slaves may communicate many times per timestep; in our model, there is at most a single communication among two slaves per timestep. Even with such a simplification, simulation results are meaningful. Wall clock values obtained from the simulation differ from measured wall clock values by a constant factor in each case. That difference is due to the portion of the waiting time that is not considered by the simulation model. Our simulation captures 81% of the slaves' waiting time in the 'Regular' case with five slaves, 64% in the 'Microphysics' case with five slaves, 47% in the 'Regular' case with fourteen slaves and 70% in the 'Microphysics' case with fourteen slaves.

The main power of simulation is that one can forecast program behavior with static or dynamic load balancing schemes without actually implementing them in the real code. Any load balance strategy will change computation time. Given the observed computation times, one can approximately state what will be the computation times when a particular load balance strategy is applied. By assuming these new computation times in the simulation model, one can obtain an approximation of the slaves' wall clock time under that particular strategy. Consequently, one can foresee and compare the benefits of different strategies. We proceed by explaining how each strategy was implemented and show our simulation results.

C.1 Static Load Balancing

What will be the total computation time, on each slave, when a perfect static load balance strategy is implemented? If the load balance is perfect, it means that the original domain is partitioned in such a way that all slaves end up with exactly the same total computation time. Assuming that the overall computation time will not vanish (or be created) by moving a set of vertical grid columns from one slave to another, the sum of total computation times across all slaves in the static load balance case should be the same as in the available measured execution. Thus, the total computation time of a slave in a perfect static load balance execution should be the average, over all slaves, of the measured computation time.

Given the predicted total computation time of each slave under perfect static load balance, the next problem is how

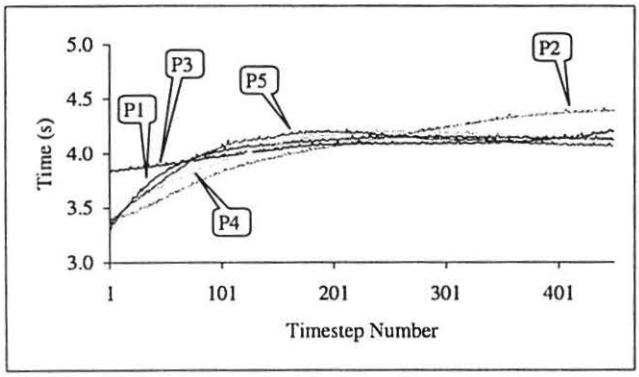

Fig. 7. Computation times in RAMS with static load balance.

to predict its computation time at each timestep of the execution. We first compute a multiplication factor that, when applied to the measured total computation time of a slave, produces the perfect static load balance. Once that factor is known, it is applied to the measured computation times of each timestep for this slave.

We inserted the computation times obtained in this way into the simulation procedure, for the case of five slaves with high microphysics. Table IV reports simulation results and measured original data. It shows that static load balance decreases wall clock execution time, by composing improved computation times with reduced delays. Computation times are perfect, but there still exists a delay at each slave, due to waiting.

To understand the presence of delays under the perfect static load balance situation in Table IV, one must consider that the amount of computation at each grid point changes with time. We can see this variation in Figure 7, which shows the new computation times assuming perfect static load balance. Although the sum of computation times over all timesteps is now constant across slaves (see Table IV), at a fixed timestep computation times vary from slave to slave. This difference generates delay at the end of each timestep.

Comparing Figures 7 and 5 (that is, perfect static load balance against original data), one can see that perfect static load balance packs slave computation times closely. But it also shows the need for dynamic load balancing, due to the variations of computation times across slaves along the timesteps.

TABLE IV
TIMES (SECONDS) FOR STATIC LOAD BALANCE IN RAMS.

	Original Run			Static Load Balance		
	Comp	Wait	Wall	Comp	Wait	Wall
P1	2129.09	88.25	2217.34	2054.78	74.02	2128.81
P2	2165.25	52.08	2217.34	2054.78	74.02	2128.81
P3	1728.58	488.75	2217.34	2054.78	74.02	2128.81
P4	2072.93	144.40	2217.34	2054.78	74.02	2128.81
P5	2178.06	39.27	2217.34	2054.78	74.02	2128.81

TABLE V
EXPECTED EFFECT OF RE-BALANCING ON RAMS.

Interval	Wall Clock Time (s)
original	2217.34
240	2188.57
200	2168.75
160	2156.53
120	2135.95
80	2117.08
40	2093.78
1	2118.95

C.2 Dynamic Load Balancing

With dynamic load balancing, slave domains are rearranged during execution, according to a load imbalance metric. The idea is to measure load imbalance at certain timesteps. Whenever this measured value exceeds a certain threshold, the domain is re-partitioned over slaves to evenly distribute the computation.

We simulate the effect of domain re-partitioning at a given timestep by multiplying computation times of the remaining timesteps by the re-partitioning factor. By applying this procedure at evenly spaced timesteps, we simulate dynamic load balancing. It is important to observe that we do not account for the cost of re-partitioning. In other words, our estimate is optimistic.

Table V shows how the slave's wall clock time varies with the frequency of re-balancing. The column labeled 'Interval' reports the number of timesteps in which a certain partitioning remains unchanged. One can see from the data in Table V that there is a continuous decreasing trend in execution time as the frequency of re-balancing increases. until re-balancing is performed at every timestep. At this point, wall clock time increases; there are two reasons for this phenomena, both due to the fact that the computation time in the next timestep is corrected by the re-balancing factor of the previous timestep.

The first reason is the wild variation of computation time per timestep due to the processing of the RADIATE function (remember, from Figure 4, that RADIATE is active only at some timesteps). The computation time in a timestep immediately after the execution of RADIATE was corrected by the variation of computation time under RADIATE. Second, and most important, one should consider the history of computation times in previous timesteps to extrapolate future behavior, instead of simply assuming that the next timestep will be similar to the previous one.

Suppose we could solve these problems and had a mechanism that perfectly predicted computation times in future timesteps. In that case, what would be the total wall clock time? Since we have all computation times, we could simulate an oracle. That is, at any timestep, we know the computation times of slaves in the next timestep, and thus, we could compute and use a perfect re-balancing factor. Since

we are using dynamic balance, this re-balancing factor would change on every timestep.

We simulated this procedure, named "Dynamic Balancing with Oracle". It results in a total computation time of 2054.78 seconds. Not surprisingly, this is the same computation time as with the "Perfect Static Load Balancing" case. However, now there is no delay and wall clock time is the same as computation time.

VI. Conclusion and Future Work

We presented an experimental performance analysis of the current parallel RAMS version. This analysis was based on instrumented executions of the program, detailed analysis of obtained traces and simulation of possible optimization schemes based on load balancing. By using the technique of event tracing, we were able to capture low level information on the dynamics of program execution, and use such information to locate and quantify critical performance issues.

Our instrumentation and simulation infrastructure allowed us to restrict the effort of improving parallel RAMS performance to two directions:

1. Overlap master's I/O with slave computation. That requires increasing the intelligence of the slave, implementing a distributed scheme to replace the current master's centralized control;
2. Select dynamic load balance over an improved static load balance. Within dynamic load balance, the extrapolation of future computation times across timesteps is crucial, due to the dynamic nature of load distribution, originated by the spatial and temporal traveling nature of the atmospheric phenomena being modeled.

The same infrastructure allows the determination of an upper bound on wall clock time reductions without actually implementing each strategy. By overlapping master's I/O with slave computation, we can reduce wall clock time in the case of five slaves with high microphysics by around 146 seconds (adding selected entries of Table III). On the other hand, implementing a perfect dynamic load balancing scheme can reduce execution time by at most 163 seconds.[4]

Our future work includes improving our execution time model, by spreading slave communication over the timestep, instead of the current concentration at the end of the timestep. It also includes model validation, whenever a dynamic load balancing scheme is available for RAMS. Meanwhile, we intend to explore hardware monitoring techniques to collect performance information that is not obtainable by software based approaches.

[4]This is achieved by subtracting, from the 'original' execution time reported at Table V (2217 seconds), the execution time obtained with a perfect load balancing scheme (2054 seconds), which cannot be better than the oracle.

Acknowledgments

We gratefully acknowledge the support and cooperation from members of the Elebra team involved in the FINEP-RAMS project, in particular Fabio N. Küpper and Leonardo Slhessarenko Filho. We also thank Prof. Saulo R.M. Barros (IME/USP) for discussions regarding load balancing issues.

References

[BAI 95] BAILLIE, C.; MICHALAKES, J.; SKALIN, R. (Eds.) Parallel Computing, vol.23, Special issue: *Regional Weather Models*. North-Holland, 1997.

[BER 99] BERNAL, W.; KOFUJI, S.; SIPAHI, G.; ANDERSON, A.; NETTO, M. PAD Cluster: An Open, Modular and Low Cost High Performance Computing System. In: 11^{th} Symposium on Computer Architecture and High Performance Computing, 1999, Proceedings..., Natal, Brazil, Sep. 1999.

[DER 98] DEROSE, L.; ZHANG, Y.; REED, D.A. SvPablo: A Multi-Language Performance Analysis System. In: International Conference on Computer Performance Evaluation – Modelling Techniques and Tools, 10., 1998, Proceedings..., Palma de Mallorca, Sep. 1998, p.352-355.

[DRA 95] DRAKE, J.B.; FOSTER, I. (Eds.) Parallel Computing, vol.21, Special issue on applications: *Climate and Weather Modeling*. North-Holland, 1995.

[HOL 96] HOLLINGSWORTH, J.K.; MILLER, B.P. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. *Lecture Notes in Computer Science*, V.1124, p.88-98, New York, Springer-Verlag, 1996.

[MIC 95] MICHALAKES, J. MM90: A Scalable Parallel Implementation of the Penn State/NCAR Mesoscale Model (MM5). *Parallel Computing*, V.23, p.2173-2186, 1997.

[PAC 97] PACHECO, P. *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco, 1997.

[REE 93] REED, D.A.; AYDT, R.A.; NOE, R.J.; ROTH, P.C.; SHIELDS, K.A.; SCHWATRZ, B.; TAVERA, L.F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In: SCALABLE PARALLEL LIBRARIES CONFERENCE, 1993. Proceedings... IEEE Computer Society, Oct. 1993, p.104-113.

[REE 98] REED, D.A.; AYDT, R.A.; DeROSE, L.; MENDES, C.L.; RIBLER, R.L.; SHAFFER, E.; SIMITCI, H.; WETTER, J.S.; WELLS, D.R.; WHITMORE, S.; ZHANG, Y.*Performance Analysis of Parallel Systems: Approaches and Open Problems*. Joint Symposium on Parallel Processing – JSPP, Nagoy, Japan, p.239-256, Jun. 1998.

[SAT 97] SATHYE, A.; XUE,M.; BASSETT,G.; DROEGEMEIER,K. Parallel Weather Modeling with the Advanced Regional Prediction System. *Parallel Computing*, V.23, p.2243-2256, 1997.

[SCH 97] SCHÄTTLER, U.; KRENZIEN, E. The Parallel 'Deutschland-Model' – A Message-Passing Version for Distributed Memory Computers. *Parallel Computing*, V.23, p.2215-2226, 1997.

[TRE 97] TREMBACK, C. J.; WALKO, R. L. *The Regional Atmospheric Modeling System (RAMS): Development for Parallel Processing Computer Architectures* Fort Collins, USA, 1997.

[WAL 97] WALLCRAFT, A.; MOORE, D.R. The NRL Layered Ocean Model. *Parallel Computing*, V.23, p.2227-2242, 1997.