Detailed Load Balance Analysis of Large Scale Parallel Applications

Kevin A. Huck and Jesús Labarta
Barcelona Supercomputing Center
Centro Nacional de Supercomputación
Barcelona, Spain
Email: kevin.huck@bsc.es, jesus.labarta@bsc.es
http://www.bsc.es/

***Abstract*—Balancing the workload in parallel applications is a difficult task, even in conventional cases. Many computing cycles are wasted when the load is not evenly balanced across processing nodes. Global load balance analysis may determine that an application is well balanced, when in fact the application has hidden inefficiencies. In this paper, we consider the load balance of parallel applications which present unique challenges in the analysis process. We have performed trace analysis and simulation to demonstrate the existence of otherwise undiscovered performance issues. We also demonstrate that by collecting dynamic phase profiles, we are able to approximate the analysis results of trace analysis and simulation, and more accurately represent the performance behavior of complex parallel applications than through flat or callpath profiles alone.**

Keywords: parallel performance analysis, performance tools, data mining, micro load imbalance.

I. Introduction

The primary goal in parallel computing is to compute solutions to large problems in less time by harnessing the power of highly parallel systems. Performance analysis with respect to single node performance is certainly key to improving overall system throughput for all computer systems, parallel or not. Problems such as memory access patterns and ineffective use of hardware resources are common in all types of performance analysis. In parallel applications, problems such as excessive synchronization, communication overhead and algorithmic inefficiencies can chip away at the overall system efficiency. One of the main challenges in efficiently utilizing these systems is an even distribution of work across all compute nodes. Certainly, as long as there has been parallel processing, there has been measurement of computational load balance. Nearly all parallel performance analysis tools and methods include at least some rudimentary method for measuring and analyzing load balance. Examples of such tools are Paraver [21], Kojak [18], SCALASCA [25], Vampir [4] and TAU [23].

One drawback to many of the tools mentioned above is that the load balance measured represents a *global load balance* measurement. That is, if the load balance is consistent throughout the time that the application is running, over all timesteps, then the problem can be easily identified and measured. This is particularly true for performance profiling tools, which aggregate all measurements for a given region of code over all calls to that region on that processor. If there is a large variance per call in the amount of time spent in that region of code on that processor, then that time is aggregated away to an average time spent in that region on that processor for all calls.

Trace analysis tools are better suited for measuring time variant load balance issues in applications. In fact, Casas et al. [6] have introduced a method for computing the *micro load balance* in an application. The minor drawback to their method is the amount of manual steps required to compute the micro load balance measurement. Once the trace is collected and analyzed, it must then be used to generate a simulated execution of the application with an ideal network configuration (infinite bandwidth and zero latency) in order to measure the ideal iteration time, which is a key term in computing the micro load balance. Gamblin et al. [12] present a framework for scalable collection of trace data for the purposes of load balance analysis, however the analysis presented is limited to visual examination of wave representations of the data, not quantitative measurement. Because it is a trace, the data collection method does retain event ordering information.

Another problem to consider is that the

aforementioned tools all assume that the application is a Single Instruction, Multiple Data application (SIMD). When presented with a Multiple Instruction, Multiple Data (MIMD) application, current methods fail unless the performance data for each sub-task is separated before performing the load balance analysis. Otherwise, the analysis is meaningless, as the processes are essentially running different programs.

Tuning and Analysis Utilities (TAU) includes support for phase-based profiles [20]. In a *dynamic* phase profile, one or more region of code can collect a distinct flat or callpath profile for each invocation of that region. Each invocation is given a unique identifier in the measurement library, and the identifiers also provide a mechanism for retaining some event ordering properties. Our goal in this work was to use the phase profiles as a lower-overhead, smaller footprint measurement system, and apply the computation of the micro load balance metrics in an automated system, requiring as few manual steps as possible. Our motivation for this automation is to include the analysis as part of an expert system performance analysis tool, such as PerfExplorer [16] from TAU or HPCST [8] from IBM. Quantifying the global and micro load balance for an application is a first step in automated analysis and evaluation of load balance problems. In addition, PerfExplorer includes mechanisms for clustering the performance data before analysis, providing a solution for MIMD applications. We have tested this measurement technique on three parallel applications, and in two cases, measurably improved the performance of the application through code changes or parameter selection.

The remainder of this paper is outlined as follows. An overview of the micro load balance computation is outlined in Section II. The method for computing the micro load balance using phase profiles is described in Section III. Experimental results are outlined in Section IV. Evaluation of our method and conclusions are described in Section V.

II. Micro Load Balance

As mentioned in Section I, the formula for computing the global and micro load balance is from by Casas et al. [6]. Figure 1 shows a simple global load balance example. Time T is defined as the total time spent in the measured region of code. Time T_p is defined as the non-MPI time for that region of code on process p, where $p \in P$, the set of all processes. The efficiency for process p is defined as

$$eff_p = T_p/T \tag{1}$$

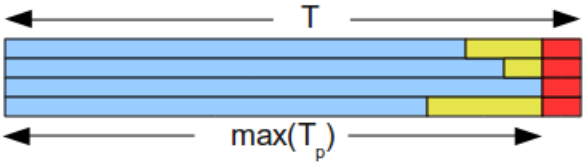


Fig. 1. Global load balance for a simple code region. The blue regions represent time spent in computation, the olive regions represent time spent waiting for synchronization, and the red regions represent actual collective communication time. Added together, the olive and red regions represent time spent in MPI.

The communication efficiency for this region of code is defined as

$$CommEff = max(eff_p) \tag{2}$$

The global load balance for this region of code is defined as

$$\begin{aligned} LB &= avg(eff_p)/max(eff_p) \\ &= avg(T_p)/max(T_p) \end{aligned} \tag{3}$$

The load balance term should be familiar, as it is a commonly used measurement of load balance - the average efficiency across all processes over the maximum efficiency across all processes.

In order to define the *micro load imbalance*, or μLB, the communication efficiency term, $CommEff$, can be expanded into two terms, the micro load balance

$$\mu LB = max(T_p)/T_{ideal} \tag{4}$$

and the real communication efficiency, or

$$Transfer = T_{ideal}/T \tag{5}$$

$max(T_p)$ is defined as the maximum across all processors of the summed time spent in computation for all iterations. T_{ideal} is defined as the *ideal execution time* for the region of code, assuming all communication is instantaneous, with zero latency and infinite bandwidth. Therefore, the final computation efficiency for this region of code is defined as

$$\eta = LB \times \mu LB \times Transfer \tag{6}$$

In the original work, the decomposition of the Communication Efficiency term requires the collection and analysis of a performance trace, and simulated execution of the trace with an ideal network to compute the T_{ideal} term. The simulation is performed with Dimemas [19], a application simulation tool.

Because phase profiles provide a sub-profile for each iteration of the loop of interest, then we can estimate

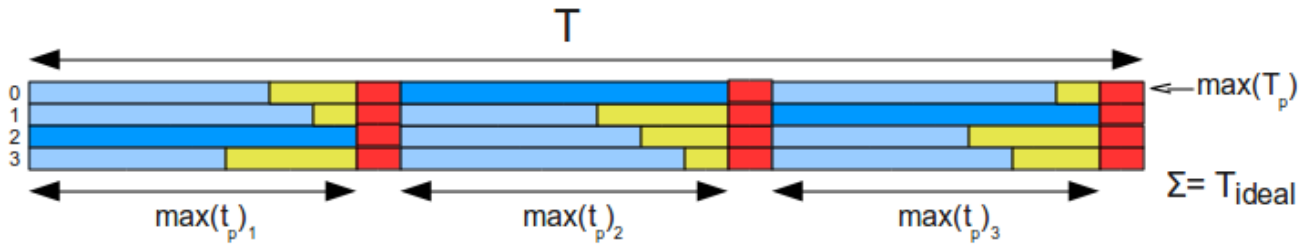


Fig. 2. Micro load balance components for an iterative code region. The blue regions represent time spent in computation, the olive regions represent time spent waiting for synchronization, and the red regions represent actual collective communication time. T_{ideal} is the sum of the darker blue regions.

T_{ideal} as the sum of all $max(t_p)$ for each iteration. This is not to be confused with $max(T_p)$, the maximum among processes of the sum of all iterations. Put another way, $max(T_p)$ represents the one process that did the most computation in the whole execution, and T_{ideal} represents the sum of all the longest running processes from each iteration. Examining Figure 2, $max(T_p)$ is the time spent computing by process 0, and $T_{ideal} = t_{2,1} + t_{0,2} + t_{1,3}$ (time computing by process 2 in the first iteration, process 0 in the second iteration, and process 1 in the third iteration).

We say that this is an estimation, because the computation of T_{ideal} is likely to be overly optimistic; because each sub-profile lacks any event ordering information, it is impossible to maintain synchronization between processes except at the outer phase boundary and any other collective operations during the iteration. This error is highly correlated with the amount of point-to-point communication per iteration. A conservative upper bound on the error in each iteration of the loop when computing T_{ideal} is the sum of all point-to-point communication performed by the process with the longest computation time for that iteration. The total error is the sum of the error from each iteration. In our examples, the estimation of T_{ideal} did not exceed 5.61%.

However, the error in computing T_{ideal} is minimized and in some cases eliminated if the region being measured has a small enough resolution, as we shall show in the inner loop analysis of Section IV-A. The phase profile approach also has the added benefit of only computing the load balance for the main iteration, ignoring initialization and termination when applications are commonly serialized because the root process usually performs most of the work.

III. Analysis Framework

As mentioned in the introduction, we wanted to examine whether the automated computation of these load balance metrics was possible with phase profile data. The original work was performed collecting performance traces, analyzing them with Paraver, generating a Dimemas trace from the original trace, simulating the application with Dimemas, and evaluating the simulated performance trace with Paraver.

The phase profile performance analysis tool we used is PerfExplorer [16], a parallel performance data mining framework with automated scripting support. The script interface includes an API for performing common data manipulation tasks for parallel performance profiles, including clustering common profile behavior, splitting trials, and computing statistics. The micro load balance computation scripts were written in Python using this framework.

In order to directly compare the measurement results using the two methods, we wrote a simple Paraver trace to TAU profile conversion utility. This utility allows us to specify one or more instrumented user functions or regions that should be interpreted as phases, and generate a TAU phase profile accordingly. This way, the same performance data from the same experiment is used in the computation for both methods.

For each application in our experiments, we first profiled the application with TAU or gprof [14] in order to determine the most time consuming regions in the application. Using the PerfExplorer framework, we constructed a script for loading the profile and selecting the top 10 functions with respect to both inclusive and exclusive time. We excluded lightweight functions that were called more than 1000 times, in order prevent excessive measurement overhead. We then constructed a source to source compilation wrapper that instruments the code using the TAU Instrumentor [13].

The instrumentation was from the Extrae API, the trace measurement library for Paraver. In addition to the most time consuming functions, we also instrumented the main iteration loops in the applications. In the CPMD case, we collected phase profiles directly using TAU instrumentation.

After the traces were collected, the micro load balances were computed using the equations defined in Section II, including generating and analyzing Dimemas traces in order to compute T_{ideal}. Using the trace to profile conversion utility, we converted the collected traces to phase profiles, and computed the load balances using the PerfExplorer scripts.

IV. Analysis Examples

In order to test our method for computing global and micro load balance using phase based profiles, we examined the performance of three parallel applications that are commonly used at the Barcelona Supercomputing Center (BSC): CPMD, GADGET and GROMACS. These applications are also members of the core benchmarks for the Partnership for Advanced Computing in Europe (PRACE) [22].

In all cases, the applications were executed on MareNostrum, the supercomputing cluster at BSC. Some experiments were also executed on Jugene, a large-scale supercomputer at the Jülich Supercomputing Center.

MareNostrum [2] is an IBM cluster with 2560 processing nodes connected with both Myrinet interconnect and Gigabit Ethernet. Each node is a JS21 Blade with four PPC970PM 2.3 GHz processors and 8 GB of main memory. The total system has 10240 processors, 20 TB of main memory, and a peak performance of 94.21 Teraflops.

Jugene [17] is a 72 rack IBM BlueGene/P cluster. Each rack has 32 nodecards, and each nodecard has 32 compute nodes (73728 nodes total). Each compute node contains four 850 MHz PowerPC 450 processors and 2 GB of main memory. The total system has 294912 compute cores, 144 TB of main memory, and a peak performance of 1 Petaflop. The compute nodes are connected with two networks, a three-dimensional torus for the compute nodes and a global tree collective network.

A. Gadget

GADGET [24] is a freely available code for cosmological N-body / Smoothed Particle Hydrodynamics (SPH) simulations on massively parallel computers with distributed memory. GADGET uses an explicit communication model that is implemented with MPI. GADGET implements a parallel TreePM [1] algorithm for efficient and accurate processing of the forces between particles in the system. The TreePM algorithm divides the forces into long and short range components. The long range forces are computed with an efficient Particle Mesh (PM) algorithm, in which the computation of the forces are performed in Fourier space. However, the force resolution of the Particle Mesh method is poor, so the short range forces are computed with a version of the Barnes & Hut [3] tree method, with boundary conditions added.

The benchmark configuration and input data we used to execute GADGET is from the PRACE benchmarks. The configuration simulates 65536000 particles for only 3 timesteps, and the PM grid is configured to have 1024 cells. The application was executed on MareNostrum and on Jugene. Our goal was to perform detailed load balance analysis of the GADGET application.

1) Micro Load Imbalances for Inner Loops: When examining the global load balance of the iterative computations in GADGET, it appears that the application is fairly well balanced. The load balance metrics for the executions are shown in Table I. A comparison of the phase analysis method to the Paraver/Dimemas is in Table II. In addition, the micro load balance looks very good, as at the end of each timestep, the processes are well synchronized. However, in our measurements, the second most time consuming function is MPI_Sendrecv. In fact, during the long range force computation, there are four loops where pairs of MPI_Sendrecv are called. When examining these loops in isolation, we discovered a severe load imbalance is caused by out-of-order communication. As shown in Figure 3(a), during these MPI_Sendrecv loops, some processes are waiting a long time for their first communication. This pattern repeats itself during the second half of the iterations.

The code executed in these loops was performing a butterfly pattern, performing pairwise communication with seemingly every other process. After examining the code and the runtime behavior in the trace data we collected, we soon realized that for the 32 processor case, each processor only performed 24 MPI_Sendrecv calls, and essentially performed a “continue” the other eight iterations. By reordering the loop for half of the processes, we were able to execute the “continue” calls for each process all at the same time, and eliminate roughly 1.4 seconds per execution of this loop, per timestep, as shown in Figure 3(b). As this loop is executed four times each timestep, that is an anticipated

Processes	32	64	128
T	466.68	229.84	139.48
$max(T_i)$	361.41	169.19	78.29
$avg(T_i)$	312.82	154.17	76.38
$commEff$	0.774	0.736	0.561
LB	0.866	0.911	0.976
T_{ideal}	361.41	172.95	78.52
μLB	1.0	0.978	0.997
$Transfer$	0.774	0.752	0.563
η	0.670	0.671	0.548

TABLE I
LOAD BALANCE METRICS FOR GADGET. ABSOLUTE TIMES ARE IN SECONDS.

Version	Original	Modified
T	105.78	123.77
$max(T_i)$	93.23	104.25
$avg(T_i)$	87.18	100.22
T_{ideal}	94.82	104.84
$commEff$	0.881	0.842
LB	0.935	0.961
μLB	0.983	0.994
$Transfer$	0.896	0.847
η	0.824	0.810

TABLE III
ANALYSIS OF LOAD BALANCE FOR THE ORIGINAL AND RECONFIGURED ITERATIONS FOR THE 2048 PROCESSOR EXECUTION OF GADGET ON JUGENE.

	Paraver/ Dimemas	Perf-Explorer	Error
T	467.14	466.68	0.10%
$max(T_i)$	361.89	361.41	0.13%
$avg(T_i)$	314.76	312.82	0.61%
T_{ideal}	382.87	361.41	5.61%
$commEff$	0.775	0.774	0.04%
LB	0.870	0.866	0.48%
μLB	0.945	1.000	-5.80%
$Transfer$	0.820	0.774	5.51%
η	0.674	0.670	0.52%

TABLE II
ACCURACY OF LOAD BALANCE COMPUTATION, T_{ideal} ESTIMATION, AND MICRO LOAD BALANCE COMPUTATION FOR A 32 PROCESSOR RUN OF GADGET ON MARENOSTRUM.

savings of 5.6 seconds per timestep, or 3.9%. The load balance comparison for both the original and the modified loop is shown in Table IV.

2) Large Scale imbalance: FFTW: When executing this configuration on Jugene with with more than 1024 processors, we discovered some interesting behavior with respect to load imbalance. In this case, the application is showing poor global load balance. During each iteration, only half of the processes have useful work to do for one region of the long range force computation. As configured for this benchmark, FFTW [11] is used to perform the FFT operations. Figure 4 shows a Paraver trace that reveals the load imbalance. The lower ranked processes perform more work than the higher ranked processes, which are waiting in MPI.

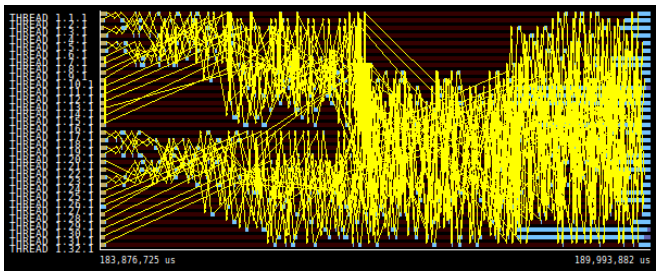
The cause for this imbalance is due to a configuration setting. The long range forces are computed in Fourier space, which requires a transformation of particle coordinates to and from Fourier space. The PMGRID parameter defines the number of cells in the grid for the particle-mesh (PM) method, used to compute the long-range forces. According to the FFTW documentation [1], when using a slab decomposition, each process gets a subset of the number of rows of the data, and as a consequence, you cannot take advantage of more processors than you have rows. This benchmark is configured to only decompose the space into 1024 slabs for FFTW. That means there are more processors than slabs, which means there are unused processors for the transformation.

Table III shows the load balance metrics for the 2048 processor run when using the increased grid resolution. Changing the grid size re-balances the application, but at a significant initialization and computation cost. In addition to the increased time computing per timestep, the pm_init_periodic() method goes from an average time of 23.006 seconds per process (46.012 seconds for processors that actually work) to 361.22 seconds per process. This time reflects the excess overhead in initializing the particle grid when doubling the size of the grid in all dimensions. In this case, the imbalance is preferable to the increased computation time overall.

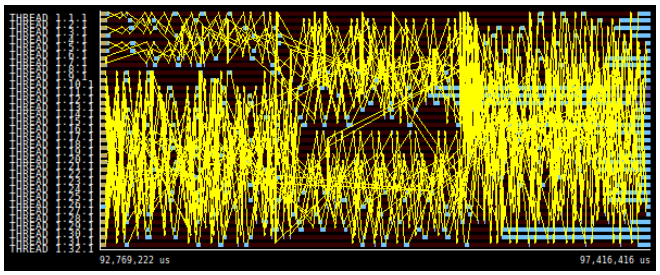
B. GROMACS

GROMACS [15] is a software package to perform molecular dynamics. It simulates the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids that have a lot of complicated bonded interactions,

[1] http://www.fftw.org/fftw2_doc/fftw_4.html#SEC58



(a) Original behavior.



(b) Modified behavior, after loop reordering.

Fig. 3. Inner MPI_Sendrecv loop behavior as seen in a Paraver trace of GADGET for 32 processes on MareNostrum. The time scale is reduced from 5.88 seconds on the original to 4.51 seconds for the modified loop.

Version	Inner Original	Inner Modified
T	5.881	4.507
$max(T_i)$	0.576	0.335
$avg(T_i)$	0.336	0.321
T_{ideal}	1.735	1.068
$commEff$	0.098	0.074
LB	0.583	0.958
μLB	0.332	0.313
$Transfer$	0.295	0.237
η	0.057	0.071

TABLE IV
ANALYSIS OF INNER LOOP PERFORMANCE FOR THE ORIGINAL AND REORDERED LOOPS.

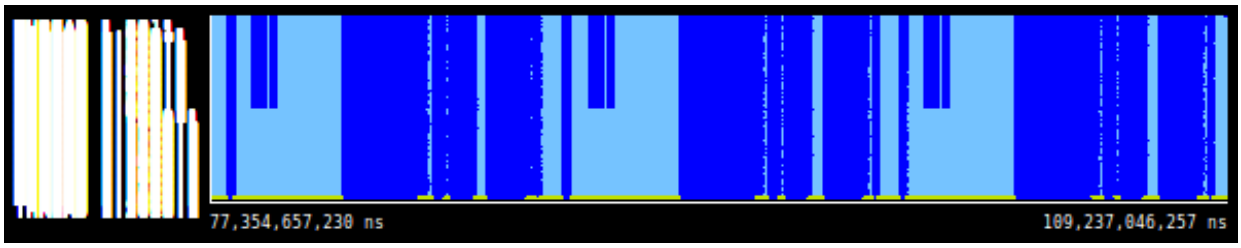


Fig. 4. Paraver view of 2048 processor run of GADGET on Jügene. The darker areas represent computation bursts. The FFT transfers on only half of the processes are clearly visible.

but since GROMACS is well suited for calculating non-bonded interactions it is also used in for research on non-biological systems, such as polymers.

As mentioned in Section I, computational simulations are typically designed as an SIMD application, where all processes execute the same algorithm on different partitions of the overall data set. By recognizing that some sub-tasks in the current implementation are independent and can be computed in parallel with the main force computation, the authors of GROMACS have designed it to optionally compute in an MIMD mode. In this mode, a subset of the processes are dedicated Particle Mesh Ewald [9] (PME) nodes, while the remainder are used to compute direct space interactions and integration. This MIMD configuration presents a challenge for load balance analysis, as the compute nodes are intentionally not computing the same code, and from a naïve global perspective are not load balanced. In this case, conventional load balancing is meaningless without first partitioning the performance data into PME and non-PME nodes.

In the benchmark configuration, 145732 atoms are simulated over 1000 timesteps on MareNostrum. There are several configuration parameters that affect the runtime performance of GROMACS, such as the use of domain or particle decomposition, the dimensionality

of the domain decomposition, the number of dedicated PME nodes. For these parameters, the application uses a heuristic to determine the default values to use. However, these values are not necessarily the optimal values for the given architecture and dataset. For our example, we used 32 processors, and the application chose to use 12 dedicated PME nodes, and the remaining 20 nodes were domain decomposed as a $4 \times 5 \times 1$ grid.

In order to perform the load balance analysis on the 12 PME nodes and 20 non-PME nodes separately, we constructed a PerfExplorer script that first clusters the performance data into 2 natural clusters using the DBSCAN [10] algorithm, then extracts each iteration of the main computational phase, and then computes the load balance and micro load balance. The load balance metrics are shown in Table V, using both methods.

Figure 5(a) shows a cut of a Paraver trace of GROMACS, representing 10 iterations. Every tenth iteration requires extra processing overhead, so those iterations are longer than the others. Only the non-PME processes are shown in this view. As can be seen in the figure, the load is poorly balanced, despite the attempts by the application to evenly distribute the computational load. Working under the assumption that the problem decomposition was at fault, particle decomposition was tried, and the performance was worse. The decomposition in GROMACS is constrained by the relationships between particles, and so evenly balanced particle decomposition is not always possible. Trying different 2D and 3D decompositions did not improve the performance, however a 1D decomposition did improve the performance, as shown in Table V and Figure 5(c) As this benchmark executes for 1000 iterations, the overall runtime for the application is reduced from 79.52 seconds to 58.64 seconds, a savings of over 26%.

C. CPMD

The CPMD [5] application is a plane wave/pseudopotential implementation of Density Functional Theory, particularly designed for ab-initio molecular dynamics. CPMD is a very regular application, with frequent synchronization. In the benchmark configuration, 20 iterations of the main computation loop are computed. During the main computation loops, only two MPI functions are used: MPI_Alltoall and MPI_Allreduce, both of which are collective operations. Therefore, CPMD is frequently globally synchronized. The load balance metrics for a 32 processor run of CPMD are shown in Table VII, and a profile view of one timestep is shown in Figure 6.

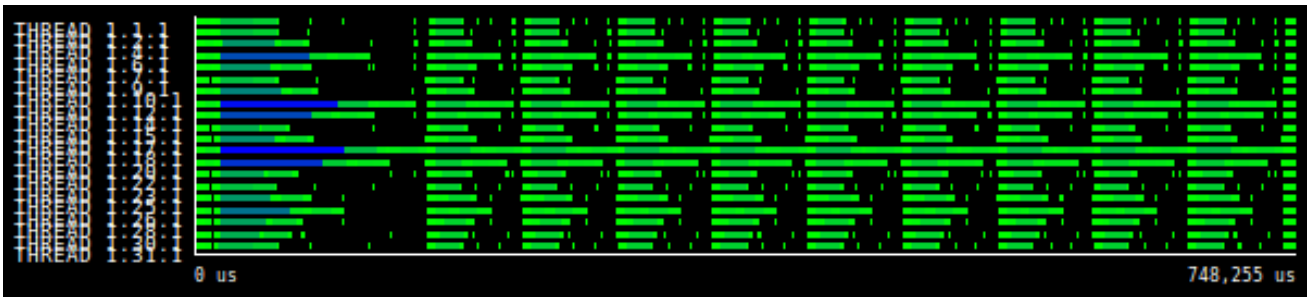
According to the CPMD manual [2], the most important bottleneck in the distributed memory parallelization of CPMD is the load-balancing problem in the FFT. Like the FFTW implementation in GADGET, the real space grids in CPMD are only distributed over the first dimension (i.e. slabs). The size of the grid is configurable, and in our benchmark input file the grid dimensions are $128 \times 128 \times 128$. Because this limits the scalability of the parallel implementation, CPMD also has the option to parallelize not only across data but also across tasks. CPMD has a TASKGROUPS option, which potentially adds additional computational overhead but reduces the communication overhead, effectively reducing the runtime.

To test the effect of the TASKGROUPS parameter, we ran CPMD on MareNostrum with 256 processors, and the $128 \times 128 \times 128$ real space grid. The test ran for 20 iterations, and we instrumented the main loop as a TAU dynamic phase. We executed CPMD with no taskgroups, 2, 4, 8 and 16 taskgroups. By adding the use of taskgroups we saw a slight decrease in the amount of computing overhead, but a more significantly, a large reduction in communication overhead. However, with more taskgroups, the computing overhead grows and eventually catches up with the reduction in communication overhead, which also is increasing with the number of taskgroups. The load balance and efficiency metrics are shown in Table VI. Included with the load balance metrics are the average time (per iteration) taken for each iteration, and the average time spent only in computation and only in communication. As emphasized in the table, the run with 2 taskgroups had the shortest runtime, and yet the worst load balance, and second lowest efficiency. If we were to select the configuration with the highest efficiency and best load balance, we would select the configuration with 16 taskgroups – but our performance with respect to time would be no better than the configuration with the worst efficiency and load balance. The efficiency and load balance metrics are misleading in this case, because increasing the number of taskgroups only adds to processing overhead, which is misinterpreted as productive computation.

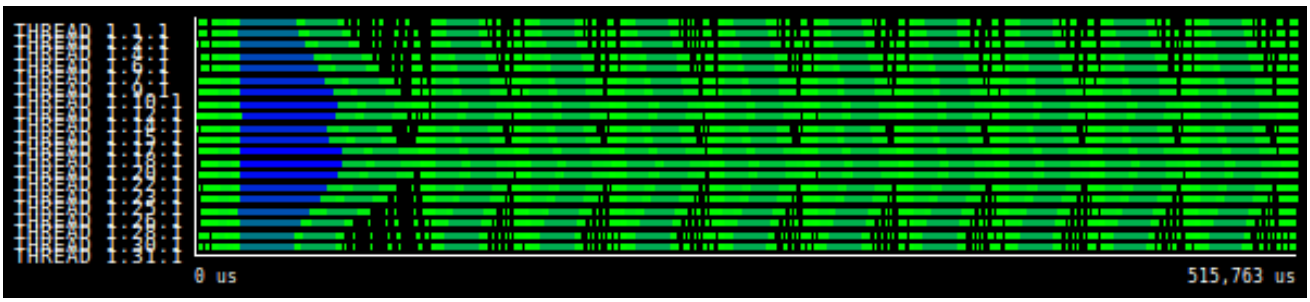
[2] http://www.cpmd.org/cpmd_on_line_manual.html

	$4 \times 5 \times 1$ **Decomposition**			$20 \times 1 \times 1$ **Decomposition**		
Metric	**Paraver/Dimemas**	**PerfExplorer**	**Error**	**Paraver/Dimemas**	**PerfExplorer**	**Error**
T	0.749	0.740	1.25%	0.525	0.504	3.95%
$max(T_i)$	0.700	0.699	0.13%	0.452	0.448	0.94%
$avg(T_i)$	0.364	0.363	0.27%	0.372	0.368	0.86%
T_{ideal}	0.708	0.699	1.25%	0.466	0.448	3.84%
$commEff$	0.934	0.945	-1.13%	0.861	0.888	-3.13%
LB	0.520	0.519	0.14%	0.822	0.823	-0.09%
μLB	0.989	1.000	-1.14%	0.971	1.000	-3.02%
$Transfer$	0.945	0.945	0.00%	0.887	0.888	-0.11%
η	0.486	0.491	-0.99%	0.708	0.731	-3.21%

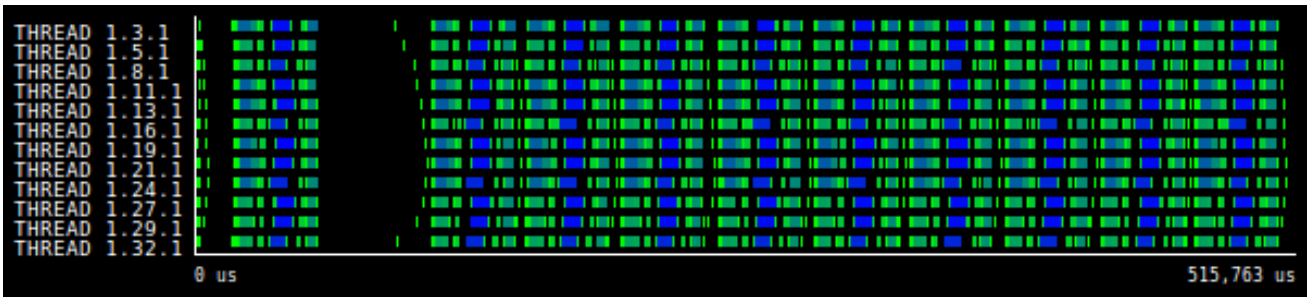
TABLE V
ANALYSIS OF LOAD BALANCE FOR 10 ORIGINAL AND RECONFIGURED ITERATIONS OF GROMACS.



(a) 4x5x1 load balance behavior, non-PME nodes. Black regions are time spent in MPI.



(b) 20x1x1 load balance behavior, non-PME nodes. Black regions are time spent in MPI.



(c) 20x1x1 load balance behavior - PME nodes. Black regions are time spent in MPI.

Fig. 5. The load balance behavior for 10 iterations of the GROMACS application executed on MareNostrum. By changing from the $4 \times 5 \times 1$ 2D decomposition to a $20 \times 1 \times 1$ 1D decomposition, the load balance and runtime for the application improved. The time scale is reduced from 0.749 seconds on the original to 0.525 seconds for the modified loops. The PME nodes are well balanced, and shown for comparison.

Taskgroups	None	2	4	8	16
T	33.775	**18.579**	23.815	26.840	32.917
$max(T_i)$	6.432	5.678	11.103	12.862	18.083
$avg(T_i)$	5.586	4.620	9.875	12.251	17.117
T_{ideal}	7.377	6.150	11.562	13.715	18.844
$commEff$	0.165	0.249	0.415	0.456	0.520
LB	0.868	**0.814**	0.889	0.953	0.947
μLB	0.872	0.923	0.960	0.938	0.960
$Transfer$	0.218	0.331	0.485	0.511	0.572
η	0.165	**0.249**	0.415	0.456	0.520
Total Time per Iteration	1.690	**0.930**	1.190	1.340	1.650
Time computing per Iteration	0.280	**0.230**	0.490	0.610	0.860
Time communicating per Iteration	1.410	**0.700**	0.700	0.730	0.790

TABLE VI
PARAMETER STUDY OF CPMD, VARYING THE NUMBER OF TASKGROUPS FOR 256 PROCESSORS ON MARENOSTRUM.

	Paraver/ Dimemas	Perf-Explorer	Error
T	83.71	83.68	0.03%
$max(T_i)$	64.87	64.91	-0.07%
$avg(T_i)$	63.56	63.61	-0.07%
T_{ideal}	62.94	64.92	-3.14%
$commEff$	0.775	0.774	0.07%
LB	0.980	0.980	0.00%
μLB	1.031	1.000	2.98%
$Transfer$	0.752	0.776	-3.17%
η	0.759	0.760	-0.10%

TABLE VII
ACCURACY OF LOAD BALANCE COMPUTATION, T_{ideal} ESTIMATION, AND MICRO LOAD BALANCE COMPUTATION FOR A 32 PROCESSOR RUN OF CPMD ON MARENOSTRUM.

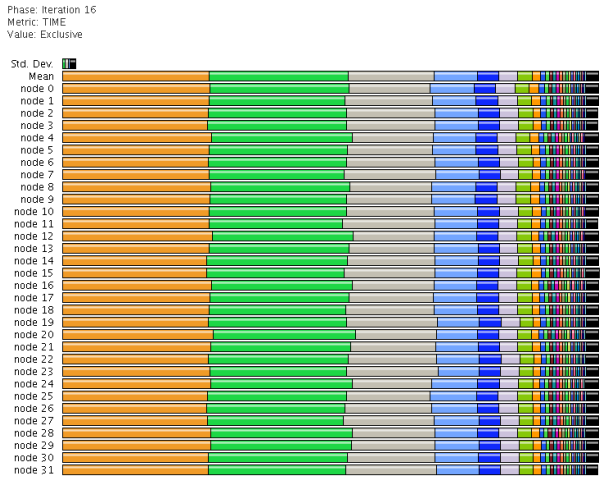


Fig. 6. Paraprof view of 1 major iteration of CPMD with 32 processors on MareNostrum.

V. CONCLUDING REMARKS

In this paper we have described a method for computing the load balance and micro load balance metrics for parallel codes using phase based profiles. Our goal in evaluating this method was to compare it with an existing method for computing micro load imbalance, which requires collecting a trace of the application, simulation of the trace using an ideal network, and manual analysis of the trace and the simulation. With phase profiles, the data can be automatically analyzed using performance analysis scripts in PerfExplorer. In our tests, the accuracy of the estimation of T_{ideal} is within 6%. In iterations with high ratios of point to point synchronous communication, such as in the GADGET case, our method is overly optimistic with how small T_{ideal} can be. In regions with higher ratios of collective communications, such as CPMD, our method is pessimistic with respect to T_{ideal}. Using PerfExplorer also allows us to post process the data in other automated ways, such as with clustering MIMD performance data prior to load balance analysis.

However, our method requires instrumentation of, or at least identification during post-processing of, the main iteration loop or the inner loop of interest. If the code is instrumented for detailed measurement anyway, this is not a burden, but if the code cannot be instrumented or is not instrumented for some reason, this would be a critical problem. However, If the phase profile is generated by post-processing of a trace, there are methods that can identify iteration boundaries [7]. These techniques could be implemented for automatic detection of the iterations. In our future work, we hope to investigate integrating this

method into the post process analysis of trace data. The trace itself could be read by PerfExplorer directly as part of this analysis.

Finally, our experiments with GADGET and CPMD have also demonstrated that load balance is one of many parameters to consider when optimizing large scale parallel applications. Tuning for load balance does not necessarily improve performance with respect to time. However, with respect to true computational load, the more balanced the decomposition is, the better the application will perform.

REFERENCES

[1] BAGLA, J. TreePM: A code for cosmological n-body simulations. *Journal of Astrophysics and Astronomy*, 23 (2002), 185–196.

[2] BARCELONA SUPERCOMPUTING CENTER. Marenostrum supercomputer. http://www.bsc.es, 2010.

[3] BARNES, J., AND HUT, P. Tree. *Nature 324*, 446 (1986).

[4] BRUNST, H., KRANZLMÜLLER, D., AND NAGEL, W. E. Tools for scalable parallel program analysis - Vampir NG and DeWiz. *Distributed and Parallel Systems, Cluster and Grid Computing 777* (2004).

[5] CAR, R., AND PARRINELLO, M. Unified approach for molecular dynamics and density-functional theory. *Phys. Rev. Lett. 55*, 22 (Nov 1985), 2471–2474.

[6] CASAS, M., BADIA, R., AND LABARTA, J. Automatic analysis of speedup of MPI applications. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing* (New York, NY, USA, 2008), ACM, pp. 349–358.

[7] CASAS, M., BADIA, R. M., AND LABARTA, J. Automatic phase detection of MPI applications. In *In: Proceedings of the 14th Conference on Parallel Computing (ParCo 2007)* (2007).

[8] CONG, G., CHUNG, I.-H., WEN, H., KLEPACKI, D., MURATA, H., NEGISHI, Y., AND MORIYAMA, T. A holistic approach towards automated performance analysis and tuning. In *Euro-Par 2009* (2009), vol. Volume 5704/2009, Springer Berlin / Heidelberg, pp. 33–44.

[9] ESSMANN, U., PERERA, L., , BERKOWITZ, M. L., DARDEN, T., LEE, H., AND PEDERSEN, L. G. A smooth particle mesh Ewald method. *J Chem Phys 103*, 19 (1996).

[10] ESTER, M., PETER KRIEGEL, H., S, J., AND XU, X. A density-based algorithm for discovering clusters in large spatial databases with noise. AAAI Press, pp. 226–231.

[11] FRIGO, M., AND JOHNSON, S. G. FFTW: An adaptive software architecture for the FFT. IEEE, pp. 1381–1384.

[12] GAMBLIN, T., DE SUPINSKI, B. R., SCHULZ, M., FOWLER, R., AND REED, D. A. Scalable load-balance measurement for SPMD codes. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–12.

[13] GEIMER, M., SHENDE, S., MALONY, A., AND WOLF, F. A generic and configurable source-code instrumentation component. In *Proc. of the International Conference on Computational Science (ICCS)* (Baton Rouge, LA, USA, May 2009), G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds., vol. 5545 of *Lecture Notes in Computer Science*, Springer, pp. 696–705.

[14] GRAHAM, S., KESSLER, P., AND MCKUSICK, M. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction* (1982), ACM Press, pp. 120–126.

[15] HESS, B., KUTZNER, C., VAN DER SPOEL, D., AND LINDAHL, E. GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. In *Journal of Chemical Theory and Computation* (2008), vol. 4, pp. 435–447.

[16] HUCK, K. A., MALONY, A. D., SHENDE, S., AND MORRIS, A. Knowledge support and automation for performance analysis with PerfExplorer 2.0. *Scientific Programming, special issue on Large-Scale Programming Tools and Environments 16*, 2-3 (2008), 123–134.

[17] JÜLICH SUPERCOMPUTING CENTER. Jugene supercomputer. http://www.fz-juelich.de/jsc/jugene, 2010.

[18] KOJAK. Kojak. http://www.fz-jeulick.de/zam/kojak/, 2006.

[19] LABARTA, J., GIRONA, S., PILLET, V., CORTES, T., AND GREGORIS, L. DiP: A parallel program development environment. In *Euro-Par 1996* (2009).

[20] MALONY, A. D., SHENDE, S. S., AND MORRIS, A. Phase based parallel performance profiling. In *In Proceedings of the PARCO 2005 conference* (2005), vol. 33, pp. 203–210.

[21] PILLET, V., PILLET, V., LABARTA, J., CORTES, T., CORTES, T., GIRONA, S., GIRONA, S., AND COMPUTADORS, D. D. D. Paraver: A tool to visualize and analyze parallel code. In *In WoTUG-18* (1995), pp. 17–31.

[22] PRACE. Partnership for advanced computing in europe (prace). http://www.prace-project.eu/, 2010.

[23] SHENDE, S., AND MALONY, A. D. The TAU parallel performance system. *The International Journal of High Performance Computing Applications 20*, 2 (Summer 2006), 287–331.

[24] SPRINGEL, V. The cosmological simulation code GADGET-2. *Mon. Not. R. Astron. Soc.*, 364 (2005), 1105–1134.

[25] WOLF, F., WYLIE, B., ÁBRAHÁM, E., BECKER, D., FRINGS, W., FÜRLINGER, K., GEIMER, M., HERMANNS, M., MOHR, B., MOORE, S., PFEIFER, M., AND SZEBENYI, Z. Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Proc. of the 2nd HLRS Parallel Tools Workshop* (July 2008), Springer, pp. 157–167.