FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Technical Report

Detecting Load Imbalance in Massively Parallel Applications

John Christian Linford

FZJ-JSC-IB-2008-09

December 2008
(last change: 26.01.2009)

Detecting Load Imbalance in Massively Parallel Applications

Internship Report

John Christian Linford
Department of Computer Science
Virginia Polytechnic Institute and State University

2201 KnowledgeWorks II
Blacksburg, VA 24060 USA
jlinford@vt.edu

http://people.cs.vt.edu/~jlinford/

Marc-André Hermanns Markus Geimer David Böhme Felix Wolf
Jülich Supercomputing Centre
Research Centre Jülich

Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH
52425 Jülich, Germany
{ m.a.hermanns, m.geimer, d.boehme, f.wolf } @fz-juelich.de

http://www.fz-juelich.de/jsc/

This report is the product of a 2008 summer internship at the Jülich Supercomputing Centre (JSC). The internship supervisors are Marc-André Hermanns and Markus Geimer of JSC, and Prof. Dr. Felix Wolf of JSC and RWTH Aachen.

January 26, 2009

Contents

1 Introduction

Recent studies of HPC platforms highlight better software environments as a key enabler of productive HPC systems. The 2007 Workshop for Software Development Tools for Petascale Computing defined "detection of load imbalance" as a high-risk, high-impact technical challenge [31]. It is the petascale manifestation of Amdahl's law, that a single thread can delay hundreds of thousands of others in petascale systems.

The majority of existing software environments aware of load imbalance focus on metrics, rather than causes. At the petascale, merely detecting load imbalance may not provide enough information to address the problem. The cause of the imbalance should also be detected and presented by a scalable method. [31] also lists "substantial advances in automation of diagnosis, optimization, and anomaly detection" as a high-risk, high-impact technical challenge.

This report describes a method for addressing both of these high-risk, high-impact technical challenges. We present a scalable method for correlating load imbalance with process idle time through post-mortem analysis. It is a *cum hoc* logical fallacy that correlation proves causation, thus our correlative method is incomplete without a scientific method for verifying causal connections. We combine our correlative method with the SILAS simulator, which can be used to verify hypotheses of different performance phenomena at very large scales [15, 16]. Our method automatically generates hypotheses of load imbalance causality as input files for SILAS, allowing us to automatically detect and verify load imbalance at large scales.

Our method makes several important contributions, not addressed by existing methods, including:

1. quantifying both load imbalance and the severity of the imbalance, in a large number of processes,
2. reporting load imbalance as one of many factors of process wait time,
3. requiring no modification of the original program code,
4. detecting complex imbalances involving different kinds of synchronizations,
5. providing a wealth of statistical information which could be used for more than load imbalance detection,
6. scaling to any number of processes manageable by the SCALASCA toolkit.

Our method consists of three sequential parts: gathering measurements, calculating correlation, and generating hypotheses. In Section 3 we define load imbalance and its related metrics. The way measurements are gathered depends on the correlation algorithm, so we first present Part 2 of our method – correlating load imbalance with process wait time – in Section 4. Part 1 is described in Section 5, and Part 3, generating hypotheses is described in Section 6.

2 Related Work

Our method combines performance analysis, load imbalance detection, and statistical correlation. This section provides a sample of literature related to performance analysis and load imbalance.

2.1 Performance Analysis

A wealth of performance analysis tools and toolsets exit to help developers fine-tune large-scale applications. These range from simple profilers, such as `gprof` [13], to sophisticated tracers based on hardware performance counters, PMPI, or a combination of methods. A brief list of these, roughly organized by increasing complexity, includes Perfctr [9], PAPI [23], mpiP [32], SvPablo [28], Perfsuit [20], HPCView [22], HPCToolkit [27], TAU [25], KOJAK [12], SCALASCA [35], and the Cray Performance Measurement Analysis Tools [7]. In general, these tools do not focus on load imbalance detection (though many have been augmented to address load imbalance of some kind), or they detect load imbalance but focus on metrics, rather than causes. We refer the reader to the literature for details on these tools.

As parallelism increases in high-performance computers, the scalability of performance analysis methods is an increasing concern. Supinski et al. developed ScalaTrace, a scalable MPI tracing tool set [5]. ScalaTrace exploits the SPMD nature of MPI applications to extract highly-compressed, full communication traces through the PMPI interface. ScalaTrace trace files often maintain a nearly constant size, regardless of node count or application run length, yet preserve structural information and temporal event order.

SCALASCA is specifically designed for trace-based performance analysis of large-scale systems [35]. SCALASCA provides a suite of tools and libraries for detecting harmful wait-states in applications, and developing scalable performance analysis tools. Our load imbalance detection method described here, and the SILAS simulator, are based on the SCALASCA toolkit.

Performance tool interoperability is an important factor in efficient performance analysis. Huck et al. developed the Performance Data Management Framework (PerfDMF) which provides a common foundation for performance data storage [19]. PerfDMF's primary design goals are integration, reusability, and portability, and it has been used successfully with ParaProf [1], PerfExplorer, and many others. PerfDMF could form the backend of a future work to record more general statistical data about load imbalance for use in visualization.

2.2 Load Imbalance Detection in Parallel Applications

Gamblin et al. used ScalaTrace to develop a tool for scalable system-wide load balance tracing using low-error compression techniques [11]. Their *Effort Model* expresses load in terms of the high-level application semantics in MPI programs. Two types of loops are identified and traced to quantify the load generated by

application code. Their method relies on filtering MPI traces to detect load imbalance. Therefore it is only applicable to MPI applications and can only measure computational imbalance (i.e. call paths are not considered).

DeRose, Homer and Johnson extended CrayPat and CrayApprentice[2] from the Cray Performance Measurement Analysis Tools to automatically identify sources of performance imbalance and present this information in an insightful way [6]. Their method uses the *imbalance percentage* to quantify the malignancy of an imbalance as a percentage of potentially-wasted resources. Their approach has two important advantages: it is generally applicable (i.e. call-path imbalance could be quantified), and it distinguishes between the size of an imbalance and the impact of an imbalance. However, it uses a global view of the system, which may be too expensive to construct in petascale environments.

Carnival, presented by Meira et al., detects waiting time and determines its causes via trace file analysis [33]. Carnival can easily detect simple cases of load imbalance with good reliability, and distinguishes between load imbalance and communication as causes of process waiting time. Carnival's visual analysis tool links source code with performance hot-spots, and provides basic statistics to assist the programmer in quantifying performance issues, even suggesting the "importance" of load imbalance in the program. In comparison with our method, Carnival is not automatic so programmers must manually deduce the severity of an imbalance. They must also modify their code to check their hypothesis, which can be impractical in large-scale production codes. Also, Carnival cannot properly identify more complex cases of synergistic imbalances involving both collective and point-to-point communication, and it requires a global view of the data.

2.3 Load Imbalance in Other Domains

Load imbalance is a well known problem in virtually all domains involving parallelism. An example from hardware design is superscalar clustering. Out-of-order superscalar execution exploits both instruction- and memory-level parallelism. A large issue width and window size improves out-of-order potential, but this method is bounded by clock frequency [26]. Clustering overcomes this barrier by creating a large aggregate issue width and window size, but at the cost of inter-cluster communication delay and inter-cluster contention for shared resources. Palacharla et al. found that load balancing in clustered superscale processors can improve performance by as much as 20%, but that load balancing in this domain is a generally hard problem [30].

Parallelizing compilers typically seek to automatically parallelize loops while estimating the overhead of parallelization. Load imbalance is a well-studied source of diminished performance [4, 2]. Sakellariou and Gurd discuss load imbalance in loop nests from a compilers perspective, and do so in general terms easily translated to other domains [29].

When static analysis is not feasible, speculative parallelization allows code sections that cannot be fully analyzed by the compiler to be aggressively executed in

parallel. The overhead of speculative parallelization may be so high that it outweighs any benefit of parallelization. Dou and Cintra present a static analysis technique for considering this overhead and predicting the performance of a speculative parallelization [8]. They found the performance impact of load imbalance to be far greater in speculative parallelization than in traditional methods, due to the speculative nature of the imbalanced thread. A thread must become non-speculative and commit before the processor can continue useful work, compounding the damage caused by an imbalance.

On the system level, load imbalance between processes (not within the processes themselves) becomes significant. Dynamic load balancing (DLB) seeks to minimize imbalance by scheduling system nodes to processes. DLB is particularly challenging, since load imbalances internal to a process can effect the scheduling of other processes [17]. Willebeek-LeMair et al. discuss the behavior, benefits, and limitations of five DLB strategies [34]. Scheduling methods for mitigating load imbalance on the system level (i.e. [10]) are important to our method, since the SILAS simulator must operate with the same performance reliability as the original run. A programmer using a replay-based or simulation-based tool may need to be aware of any DLB system and compensate for its behavior.

3 Definitions of Load Imbalance

Load imbalance has been widely discussed in literature and is often redefined to reflect the system in question. To avoid ambiguity, we define our terms by restating some common definitions of load imbalance and its related metrics.

3.1 General Definitions

In the most general sense, a load imbalance in a parallel code is the difference in work on two or more processes between two of their synchronization points [14]. This may be a difference in computation, communication, call-paths followed, I/O, etc. The difference can be measured as the deviation from the average workload between the two synchronization points. Working from definitions given in [29], we define the following imbalances. Given a total workload W_{tot}, distributed among P processes such that each process has a workload W_i, $1 \leq i \leq P$, and the average workload $W_{avg} = \frac{W_{tot}}{P}$, then the *process load imbalance* $\mathcal{I}_i$ is

$$\mathcal{I}_i = W_i - W_{avg}, \tag{1}$$

the *maximum load imbalance*, $\mathcal{I}_{max}$ is

$$\mathcal{I}_{max} = \max_i(W_i - \frac{W_{tot}}{P}) = \max_i(\mathcal{I}_i), \tag{2}$$

and the *minimum load imbalance*, $\mathcal{I}_{min}$ is

$$\mathcal{I}_{min} = \min_i(W_i - \frac{W_{tot}}{P}) = \min_i(\mathcal{I}_i). \tag{3}$$

If $\mathcal{I}_i > 0$, then process i is said to be *overbalanced*. If $\mathcal{I}_i < 0$, then process i is said to be *underbalanced*. If $|\mathcal{I}| < \delta$, for delta "close" to zero, then the workload is said to be *balanced*. If $\mathcal{I} = 0$, then the workload is said to be *perfectly balanced*. (We prefer the terms "over/underbalanced" to"over/underworked" because they reflect the use of the average workload, rather than the total workload, as our reference metric.)

One measure of the severity of an imbalance is the *imbalance percentage*, $\mathcal{I}_{\%}$ [6]:

$$\mathcal{I}_{\%} = \frac{\mathcal{I}}{W_{max}} \times \frac{P}{P-1}. \tag{4}$$

$0 \leq \mathcal{I}_{\%} \leq 100$, where $\mathcal{I}_{\%} = 0$ for a perfectly balanced workload, and $\mathcal{I}_{\%} = 100$ for a serial workload executed on a parallel system. When W_{tot} refers to computational time, the imbalance percentage corresponds to the percentage of time that a given process p is not engaged in useful work, if that process is not the most overbalanced process. If p is completely idle, this is the "percentage of resources available for parallelism" that is wasted.

3.1.1 Global-View Dependent Definitions

It is possible to define other useful metrics which are dependent on a global view of the system state. Post-mortem performance analysis tools and dynamic load balancers are two examples where a global view may be available. In this case, the global load imbalance is measured by the global standard deviation σ of all W_i, normalized to the average workload:

$$\sigma = \frac{1}{W_{avg}} \sqrt{\frac{1}{P} \sum_{i=1}^{P} \mathcal{I}_i^2}. \tag{5}$$

This definition of load imbalance is particularly useful when trying to maintain a balanced system while adjusting the global load.

[3] gives two other metrics of interest. *Load dispersion*, δ, describes the concentration of imbalanced processes over the system topology. It is calculated as the global load standard deviation computed as if each node had a load equal to the average of its load and its neighboring processes:

$$\delta = \frac{1}{W_{avg}} \sqrt{\frac{1}{P} \sum_{i=1}^{P} (W_{avg}^i - W_{avg})^2}, \tag{6}$$

where W_{avg}^i is the average workload of a neighborhood D_i, composed of P_i processes,

$$W_{avg}^i = \sum_{h \in D_i} \frac{W_h}{P_i}. \tag{7}$$

When δ is small compared to σ, each neighborhood reflects a local imbalance comparable to the global system imbalance, i.e. the imbalance is evenly distributed in the system. However, when δ is close to σ, the load varies in a continuous way and the local imbalance is very small in any neighborhood, because even a significant global imbalance can consist of only minimal contributions in any neighborhood [3].

Load variation describes the frequency of load changes over time. It is calculated as the normalized standard deviation of the load changes between time t and $t+\Delta t$, computed as if they were the only load to take into account.

$$\Delta\sigma(t, t+\Delta t) = \frac{1}{W_{avg}(t)} \sqrt{\frac{1}{P} \sum_{i=1}^{P} (\Delta W_i(t, t+\Delta t))^2}, \tag{8}$$

where $\Delta W_i(t, t+\Delta t)$ is the load change of the process in the Δt interval and $W_{avg}(t)$ is the average workload at time t.

3.2 Specifying General Definitions for Computational Load Imbalance

Our general definitions can be specified for computational load imbalance by considering the workload, W_{tot}, to be the CPU time spent in a code region (computation time). A *code region* is any programmer- or user-defined collection of instructions (typically a function or an annotated region). *Regional load imbalance* is defined by letting W_{tot} be the execution time of the correlating region instances on two or more processes [14]. In this case, W_{avg} is the average workloads of the regions, instead of the average over all processes. Unless otherwise specified, our workload metric in this paper is computation time, and our regions are functions. We use the term "region" and "function" interchangeably.

Regional load imbalance is easier to detect than global load imbalance. In order to detect a global imbalance, a global view of the system state is required. This view may be unavailable or prohibitively expensive to construct for applications involving several thousand processes. For this reason, we focus on detecting and quantifying regional load imbalance.

Note that perfectly balanced regions do not imply a balanced process or a balanced system. The region may only be balanced with respect to two specific processes (as in $sync_1$ of Figure 1(d)), or the regional imbalance may be negated by a complementary regional imbalance (as in $sync_2$ of Figure 1(d)). Thus, regional imbalance is a necessary but insufficient condition for global load imbalance.

4 Correlating Load Imbalance with Process Wait Time

A process synchronizing with one or more other processes may be forced to wait if all the communicating processes do not reach the synchronization point in a timely manner. This wait time can be attributed to a complex combination of factors,

including load imbalance. By forming a statistical correlation between regional load imbalance and process wait time, we are able to hypothesize the severity of load imbalance in a function as a factor of process waiting time. [1]

To calculate this correlation, we form the correlation matrix $R = [r_{ij}]$ where $r_{ij} = r(i, j)$ is the coefficient of correlation between random variables i and j. In our case, the variables are the time a process spends waiting for other processes, $\mathcal{T}$, and the regional imbalance in each function, $\mathcal{I}_{f_1} \ldots \mathcal{I}_{f_F}$, where F is the number of functions. Thus,

$$R = \begin{bmatrix} r(\mathcal{T},\mathcal{T}) & r(\mathcal{T},\mathcal{I}_{f_1}) & \ldots & r(\mathcal{T},\mathcal{I}_{f_F}) \\ r(\mathcal{I}_{f_1},\mathcal{T}) & r(\mathcal{I}_{f_1},\mathcal{I}_{f_1}) & \ldots & r(\mathcal{I}_{f_1},\mathcal{I}_{f_F}) \\ \vdots & & \ddots & \vdots \\ r(\mathcal{I}_{f_F},\mathcal{T}) & r(\mathcal{I}_{f_F},\mathcal{I}_{f_1}) & \ldots & r(\mathcal{I}_{f_F},\mathcal{I}_{f_F}) \end{bmatrix}$$

is a $(F + 1) \times (F + 1)$ positive semidefinite matrix with 1's on the diagonal.

We know from basic statistics that if $C = [c_{ij}]$, where $c_{ij} = c(i, j)$ is the covariance of random variables i and j, then R is related to C by

$$r_{ij} = \frac{c_{ij}}{\sqrt{c_{ii}\ c_{jj}}} \tag{9}$$

and

$$c(i, j) = E((i - E(i))(j - E(j))) = E(ij) - E(i)E(j), \tag{10}$$

where E is the expected value operator. Taking the vector definition of E,

$$E = \frac{1}{N} \sum_{i=0}^{N-1} V_i, \tag{11}$$

for a vector V of N measurements taken from a random variable, then C, and subsequently R, can be formed easily from a *measurement matrix* $M = [m_{ij}]$ who's columns are vectors of measurements of a random variable. In our case,

$$M = \begin{bmatrix} \mathcal{T}_1 & \mathcal{I}_{f_1 1} & \ldots & \mathcal{I}_{f_F 1} \\ \mathcal{T}_2 & \mathcal{I}_{f_1 2} & \ldots & \mathcal{I}_{f_F 2} \\ \vdots & & \ddots & \vdots \\ \mathcal{T}_N & \mathcal{I}_{f_1 N} & \ldots & \mathcal{I}_{f_F N} \end{bmatrix}.$$

is an $N \times (F + 1)$ matrix.

4.1 Forming the Measurement Matrix

To form measurement matrix M, we need to sample the regional imbalance and the process waiting times during the execution of a program. The sampling can

[1] All the mathematics in Section 4 can be found in any introductory statistics book. [24] provides a working introduction to statistics for English speaking scientists and engineers.

be done post-mortem by examining a trace file taken during program execution. We use the SCALASCA performance analysis toolkit [35] to gather and manipulate these trace files.

Waiting time and load imbalance measurements are gathered by replaying SCALASCA trace files using the PEARL replay mechanism. When a thread replaying the trace file encounters a synchronization point (i.e. MPI_Recv or MPI_Send), the synchronizing threads exchange information about time spent waiting for other threads and time spent in regions of interest *since these threads last synchronized*. This information is sufficient to calculate the regional imbalance accumulated since these threads last synchronized. Logically, the imbalance measurement, along with the time spent waiting at this synchronization point, is recorded as a row of the measurement matrix. This is not done explicitly in practice (see Section 4.3). Note that M is unlikely to be dense for large F, since a thread must visit every region between every synchronization point for the duration of the program to produce a dense M.

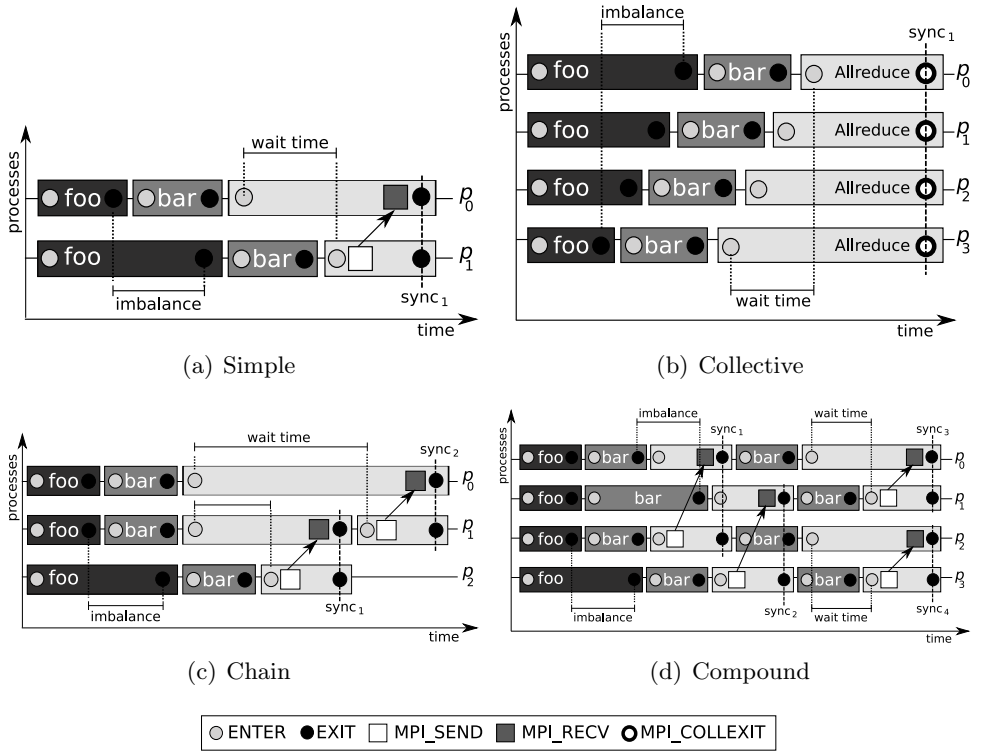
In the case of collective operations, a more accurate average workload can be calculated since more threads participate in the synchronization. However, it is possible to view collective operations as a set of point-to-point synchronizations composed from pairs of synchronizing threads. This is the approach taken by Carnival [33]. In this case, the average will be less accurate, but it will be normalized consistently across all measurements (i.e. the denominator is always 2). Different implementations may favor one method over the other. Both should be explored.

4.2 Simple Examples

Consider the simple case shown in Figure 1(a). The wait time at synchronization point $sync_1$ is directly attributable to the load imbalance in function `foo`. During replay, processes p_0 and p_1 record the time spent in function `foo` and function `bar`, beginning at the far-left of the graph. Both processes record the same length of time for `bar`, but p_1 records approximately twice as much time for `foo` as p_0. When the communication at $sync_1$ is replayed, the data communicated is the cumulative time in `foo` and `bar`, rather than whatever was sent in the original program run. These function times are compared, an imbalance is detected, and the waiting times are recorded. The measurement matrix for this case is shown in Figure 2(a).

Collective communication (Figure 1(b)) is a logical extension of the simple case. When the collective communication is replayed, all threads involved in the communication receive the cumulative function times from all other communicating threads. The imbalance and wait times are calculated similar to the simple case and recorded (see Figure 2(b)). In this example, we do not interpret collective communication as a collection of point-to-point communications.

An imbalance may have far-reaching effects when it delays a thread which other threads depend on. An example of this kind of "chained" communication is shown in Figure 1(c). When processes p_0 and p_1 communicate at $sync_2$, there is no imbalance in `foo` and `bar` between these threads. To correctly attribute this wait state to the



(a) Simple (b) Collective

(c) Chain (d) Compound

Figure 1: Example load imbalance cases.

imbalance in `foo` on p_2, we should compare the function times in p_0 to those in p_2. Notice, that since the function times are equal in p_0 and p_1, the comparison has already been done by comparing p_1 with p_2 at $sync_1$. Therefore, when wait time is detected and the synchronizing processes are regionally-balanced, we only need to duplicate the previously recorded imbalance information. This can be achieved by remembering a function's last observed remote imbalance and passing it forward at the next synchronization. The wait time should be the difference of the wait times in the receiving thread (p_0) and the sending thread (p_1) to avoid multiple records of the same wait time. The measurement matrix for Figure 1(c) is shown in Figure 2(c).

It is possible for imbalances to synergistically cause a thread to wait. Communication at $sync_2$ in Figure 1(d) is not delayed by the imbalances in `foo` or `bar` in processes p_1 and p_3 because the imbalances are complementary. We call this a *benign imbalance* between `foo` and `bar`. However, the wait time detected in p_0 at $sync_3$ is partly attributable to the imbalance in `foo` on p_3. If we treat communication at $sync_3$ as an instance of the simple case, this waiting time will be attributed to the imbalance in `bar` alone.

To correctly identify the cause of the wait at $sync_3$, we must consider the correlation not only between the wait time and the function times, but also between the

function times themselves. Figure 3(d) shows the correlation matrix for the compound case. The coefficients of correlation for `foo` and `bar` with respect to wait time are equal, but the coefficients of correlation for `foo` and `bar` with respect to each other show that these function times are complementary. Thus, an imbalance attributed to `foo` may also be partially due to an imbalance in `bar`. Or to put it another way, if we were to balance `foo`, we may also be required to balance `bar` to achieve a balanced program.

$$\begin{bmatrix} 4 & 3-\frac{3+6}{2} & 3-\frac{3+3}{2} \\ 0 & 6-\frac{3+6}{2} & 3-\frac{3+3}{2} \end{bmatrix}$$
(a) Simple

$$\begin{bmatrix} 0 & 6-\frac{6+5+4+3}{4} & 3-\frac{3+3+3+3}{4} \\ 1 & 5-\frac{6+5+4+3}{4} & 3-\frac{3+3+3+3}{4} \\ 2 & 4-\frac{6+5+4+3}{4} & 3-\frac{3+3+3+3}{4} \\ 3 & 3-\frac{6+5+4+3}{4} & 3-\frac{3+3+3+3}{4} \end{bmatrix}$$
(b) Collective

$$\begin{bmatrix} 4 & 3-\frac{3+6}{2} & 3-\frac{3+3}{2} \\ 0 & 6-\frac{3+6}{2} & 3-\frac{3+3}{2} \\ 8-4 & 3-\frac{3+6}{2} & 3-\frac{3+3}{2} \\ 0 & 3-\frac{3+3}{2} & 3-\frac{3+3}{2} \end{bmatrix}$$
(c) Chain

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 3-\frac{3+6}{2} & 6-\frac{3+6}{2} \\ 0 & 6-\frac{3+6}{2} & 3-\frac{3+6}{2} \\ 4 & 3-\frac{3+3}{2} & 6-\frac{9+6}{2} \\ 0 & 3-\frac{3+3}{2} & 9-\frac{9+6}{2} \\ 4 & 3-\frac{3+6}{2} & 6-\frac{6+6}{2} \\ 0 & 6-\frac{3+6}{2} & 6-\frac{6+6}{2} \end{bmatrix}$$
(d) Compound

Figure 2: Measurement matrices for the common load imbalance cases shown in Figure 1 when *bar* is three time units on all processes (except the first call on p_1 in the compound case). The first column is $\mathcal{T}$, the second is $\mathcal{I}_{foo}$, and the third is $\mathcal{I}_{bar}$.

$$\begin{bmatrix} 1.0 & -1.0 & NaN \\ -1.0 & 1.0 & NaN \\ NaN & NaN & 1.0 \end{bmatrix}$$
(a) Simple

$$\begin{bmatrix} 1.0 & -1.0 & NaN \\ -1.0 & 1.0 & NaN \\ NaN & NaN & 1.0 \end{bmatrix}$$
(b) Collective

$$\begin{bmatrix} 1.0 & -1.0 & NaN \\ -1.0 & 1.0 & NaN \\ NaN & NaN & 1.0 \end{bmatrix}$$
(c) Chain

$$\begin{bmatrix} 1.0 & -0.4082 & -0.4082 \\ -0.4082 & 1.0 & -0.5 \\ -0.4082 & -0.5 & 1.0 \end{bmatrix}$$
(d) Compound

Figure 3: Correlation matrices corresponding to the matrices shown in Figure 2.

The NaN ("Not A Number") values in Figure 3 result from a variable having no variance over all measurements. This is revealing, and should not be interpreted like a zero value. Having no variance suggests that the state of the program has absolutely no impact on the value of this variable. A value of zero reveals that the variable is changing over the course of the program, yet there is no correlation. This is apparent in the examples where `bar` is always 3 time units.

4.3 Reducing Memory Consumption

Storing the entire measurement matrix M in memory may be impractical, due to memory limitations. Each synchronization point in each thread in a trace file produces at least one measurement, so petascale systems with hundreds of thousands of threads will produce an M of considerable size. We can reduce the memory requirements of the method by recognizing that we do not need to form M explicitly. Instead, the correlation matrix R can be formed from running sums reflecting the same statistical information provided by M.

Working from Equations 9, 10, and 11, we can define $r(i,j)$ as

$$r(i,j) = \frac{s^1_{i,j} - \frac{1}{N} s^1_i s^1_j}{\sqrt{(s^2_i - \frac{1}{N}(s^1_i)^2)(s^2_j - \frac{1}{N}(s^1_j)^2)}} \tag{12}$$

where

$$s^k_x = \sum_{n=1}^{N} x^k_n, \qquad s^k_{x,y} = \sum_{n=1}^{N} (x_n y_n)^k.$$

From Equation 12 it is clear that only N, s^1_i, s^1_j, s^2_i, s^2_j, and $s^1_{i,j}$ are needed to calculate $r(i,j)$. Therefore, we only need to store s^1_i, s^2_i, and $s^1_{i,j}$ for every measured variable, i.e. every function and the wait time. This gives us an upper limit of $\frac{1}{2}(F^2 + F) + 2F$ values to store.

Although the memory usage of this method grows quickly in F, the F^2 term can be significantly reduced in practice. $\frac{1}{2}(F^2 + F)$ values are required to store $s^1_{i,j}$ for every possible i and j. However, we only need to store $s^1_{i,j}$ when we are interested in the correlation of this particular i and j. If we are only interested in correlating wait time with all other functions, then only F of these terms are required, reducing the upper limit to $3F$.

In the compound case (Figure 1(d)), the correlation between functions `foo` and `bar` is required, due to the benign imbalance at $sync_2$. If we only store $s^1_{i,j}$ when a benign race manifests, we can significantly reduce the memory requirements of the method. Under this condition, the probability of reaching the upper limit is inversely proportional to F, since every function must manifest a benign imbalance with every other function in order to reach the upper limit. However, there may be cases other than benign imbalance when the inter-function correlation is desired. This should be explored.

In this paper, we extend this syntax for the sake of readability. For `foo`, an arbitrary function on the local process:

$$s^k_{foo} = \sum_{i=1}^{N} \mathcal{I}^k_{foo\ i}, \qquad s_{\mathcal{T},foo} = \sum_{i=1}^{N} (\mathcal{T}_i \mathcal{I}_{foo\ i}) \tag{13}$$

5 Gathering Measurements of $\mathcal{T}$ and $\mathcal{I}_{foo}$

When a synchronization point between two or more threads is reached, performance metrics of interest since these threads last synchronized must be calculated.[2] These metrics are typically the wall clock time spent in functions of interest, however, they could also be time spent in communication or synchronization functions, number of times a function is called, branch degree of a call tree, etc. For this paper, we take our measurements to be the regional imbalance in functions of interest ($\mathcal{I}_{foo}$), and wait time ($\mathcal{T}$). Both inclusive and exclusive times can be calculated trivially, so we suggest the user be allowed to specify which is used. This requires either recording the function times at every synchronization so they are available at the next synchronization, or tracing backwards through the trace file to calculate this time.

Tracing backwards through the trace file conserves memory, but it may complicate the replay logic and computation complexity. Since memory is the primary concern of modern petascale systems (such as the Blue Gene [21]), this is our recommended method. The local process only records the identifier of the last process it last synchronized with. If a synchronization point is reached, and the remote process is not the same as the last synchronized process, the local process traces back through the trace file to the last time these threads synchronized, calculating the function times since last synchronization. Communication is not replayed, so this can be done as quickly as the local process can calculate. This method is described algorithmically in Section 7.2.

The second method, shown in Section 7.3, records function times easily and efficiently via a stack method. Function times are pushed onto the stack as synchronizations take place, and a bit-field records the thread IDs of threads the local thread has synchronized with. If a synchronization point is reached, and the synchronizing threads have each other in their bit fields, then the time since their last synchronization can be calculated by summing downwards from the top of the stack until the record of their last synchronization is reached. This method has the obvious disadvantage that a thread is required to store at most $P \times F$ double-precision values, where P is the number of processes. Any petascale software must be cautious when complexity grows by the number of threads.

[2]The synchronizations do not need to be of the same type. For example, the first may be a collective synchronization involving all threads, and the second may be a point-to-point synchronization between two threads. There will always be a pair of synchronizations, since the program start and program termination are interpreted as collective synchronizations across all threads

6 Generating Hypotheses

After the correlation matrix has been constructed, we can use heuristic methods to generate hypothetical improvements for the code. This section describes briefly how hypotheses are generated from the correlation matrix and communicated to the SILAS simulator.

6.1 Justification for Simulation

It is well known that "correlation does not prove causation." Let $Y_t(u)$ be a response variable representing the response of unit u when exposed to t. In order to observe causation, we must have two response variables, $Y_t(u)$ and $Y_c(u)$, representing two potential responses. The effect of the cause t on u as measured by Y, relative to cause c is

$$Y_t(u) - Y_c(u). \tag{14}$$

However, it is impossible to observe the value of $Y_t(u)$ and $Y_c(u)$ on the same unit and, therefore, it is impossible to observe the effect of t on u [18]. This is known as the *Fundamental Problem of Causal Inference*. In other words, it is impossible, in general, to "rewind history" and replay events, having made controlled changes. Thus, we cannot simultaneously observe the effect of both t and c on u.

Fortunately, the Fundamental Problem does not imply that causal inference is impossible. Although simultaneous observation of $Y_t(u)$ and $Y_c(u)$ is impossible, the relevant knowledge can still be gathered. Post-mortem tracefile analysis overcomes the Fundamental Problem by exploiting invariance assumptions in the replay. That is, if the value of $Y_c(u)$ measured for the original run is equal to the value of $Y_c(u)$ during replay, then we can expose u to t in replay and measure $Y_t(u)$, thus overcoming the Fundamental Problem.

The SILAS simulator exhibits an error of less than $0.3 \times 10^{-2}\%$ in replay [14]. For most purposes, this error is small enough to effectively assume that the replay is equal to the original run. Therefore, if we can establish a correlation between process wait time and load imbalance, we can determine if load imbalance is a cause of process wait time by using SILAS to replay the trace file.

Naturally, a programmer could modify the original code rather than use a simulator. This is the approach taken by Carnival [33] and ScalaTrace [11]. However, the complexity of petascale production codes makes this infeasible. It is quite possible that the modifications will introduce bugs which harm performance more than the corrected load imbalance, and the effect of the bugs will be misinterpreted as being related to the imbalance. Furthermore, unless several versions of the code are maintained, only one hypothesis can safely be tested at a time with this method. Using a simulator, we eliminate the possibility of bugs, simplify experiments, and open the possibility of running multiple experiments just by starting multiple instances of the simulator.

6.2 Forming Hypotheses

Given the correlation matrix R, we can form hypotheses on the effect of balancing functions by using simple heuristics. A large correlation between wait time and a function `foo` indicates that `foo` was often imbalanced when waiting time was observed. This suggests the hypothesis "If foo is balanced, then wait time will be reduced". Hypotheses of this sort can be easily derived by taking the largest value from the first row or column of R. The index of this value indicates which function we should try to balance. If the correlation is negative, then `foo` is underbalanced, otherwise `foo` is overbalanced.

The complexity of production systems requires examination of all the factors related to balancing `foo`. If `foo` frequently exhibits benign imbalance with a function `bar` then the coefficient of correlation between `foo` and `bar` will be large. If i is the index of the row/column corresponding to correlations in `foo`, then correlation with bar can be easily checked by looking up the value in row/column i corresponding to `bar`. Naturally, `bar` may also have a high correlation with another function. We can continue this search to a predefined depth or until no correlation is found.

After forming one or more hypotheses, a SILAS *optimizing transformation file* is written following the format given in [14]. This file is passed to the simulator along with the original program trace files. The simulator modifies the traces according to the stated transformation, simulates the modified program behavior, and outputs a new event trace that can be analysed by the Scalasca trace analyser. Changes in performance metrics of the simulated application behaviour can then be visualized in Cube, the SCALASCA analysis report browser.

7 Algorithm Listings

Here, we outline the algorithms of our detection method using both a stack-based and a trace-back method. First, we show how to calculate the correlation between wait time and an arbitrary function `foo` given t^p_{foo}, the cumulative time spent in `foo` on process p between two synchronization points involving the same threads. Then we show two methods for calculating t^p_{foo}.

7.1 Correlation Algorithm

In this section, $\mathcal{T}^p_i$ is the time process p waited at synchronization point $sync_i$, and P is the number of processes participating in synchronization at $sync_i$. $H\mathcal{I}^p_{foo}$ is the *historical imbalance time* of function `foo` on process p. It is the last detected imbalance in `foo` on p and is required to correctly process imbalance chains (Figure 1(c)).

In the interest of simplicity, the algorithm ignores measurement error. An implementation should take measurement error and roundoff into account, perhaps with threshold values. For example, when "sum == 0" is written, what is intended is

"sum $\leq \delta$" for a small δ. Inter-function correlation is also ignored. An implementation should keep a list of $s_{i,j}$ factors in the case of detecting a benign imbalance, or whenever a $s_{i,j}$ factor is required.

```
1  // Replay through the entire trace file
2  for(event e = first event ; e ≤ last event ; e = e.next) {
3
4    if(e is function foo enter event) {
5      enter = e.timestamp;
6    }
7
8    if(e is function foo exit event) {
9      t_foo^local = t_foo^local + e.timestamp − enter;
10   }
11
12   if(e is any send event) {
13     // Send all function time information to remote process
14     for (each function foo) {
15       t_foo^local = calculate_time(e, foo);
16       send(t_foo^local, HI_foo^local, T_i^local);
17     }
18
19     // Remember last synchronization event
20     lastsync = e;
21   }
22
23   if(e is any receive event) {
24     for (each function foo) {
25       t_foo^local = calculate_time(e, foo);
26       receive(t_foo^remote, HI_foo^remote, T_i^remote);
27
28       // Average foo for these threads
29       avg = (1/P) Σ_{i=1}^{P} t_foo^i;
30
31       // Calculate local imbalance in foo
32       I_foo^local = t_foo^local − avg;
33
34       // Calculate remote imbalance in foo
35       I_foo^remote = t_foo^remote − avg;
36     }
37
38     // Detect possible chain
39     for(each function foo) {
40       sum = sum + I_foo^local;
41     }
42
43     if(sum == 0) {
44       // Chain was detected, so use chained times
45       for(each function foo) {
```

```
46            I_foo^local = I_foo^remote;
47
48            I_foo^remote = HI_foo^remote;
49          }
50          // Adjust wait time to not duplicate wait time
                measurements
51          T_i^local = T_i^local - T_i^remote;
52        }
53
54        // Update statistical sums
55        s_foo = s_foo + I_foo^local + I_foo^remote;
56
57        s2_foo = s2_foo + (I_foo^local)^2 + (I_foo^remote)^2;
58
59        s_T = s_T + T_i^local + T_i^remote
60
61        s2_T = s2_T + (T_i^local)^2 + (T_i^remote)^2
62
63        s_T,foo = s_T,foo + (T_i^local * I_foo^local) + (T_i^remote * I_foo^remote);
64
65        // Carry chained imbalance forward
66        HI_foo^local = I_foo^remote
67
68        // Remember last synchronization event
69        lastsync = e;
70
71      } // End (e == receive event)
72
73  } // End replay
74
75  // Gather all metrics to a master thread
76  sum_allreduce(s_foo, s2_foo, s_T, s2_T, s_T,foo)
77
78  // Calculate correlation on master thread
79  if(this is master thread) {
80    for(each function foo) {
81      r_T,foo = (s_T,foo - (1/P) s_T s_foo) / sqrt((s2_T - (1/P)(s1_T)^2) * (s2_foo - (1/P)(s1_foo)^2)
              );
82    }
83  }
```

7.2 Trace-based function time calculation

The trace-based function time calculation method is most appropriate when node memory is scarce and compute cycles abundant. This method calculates the inclusive time spent in a given function `foo` between synchronization points $sync_1$ and $sync_2$ as follows:

```
1  calculate_time(event e, function foo) {
2    if(lastsync.id == e.id) {
3      return t_foo^local;
4    } else {
5      t_foo^local = 0;
6
7      // Scan backwards through trace
8      for(event x = e.previous; x.id != e.id; x = x.previous) {
9        if(e is function foo exit event) {
10         exit = e.timestamp;
11       }
12
13       if(e is function foo enter event) {
14         t_foo^local = t_foo^local + exit - e.timestamp;
15       }
16     }
17
18     return t_foo^local;
19   }
20 }
```

7.3 Stack-based function time calculation

The back-and-forth scanning of the trace-based function time calculation method can be eliminated in favor of a single-pass method. We call this a stack-based method because partial function times are accumulated on a stack as synchronization points are encountered. The complexity of this method is $\mathbf{O}(F \times P)$ where P is the number of processes and F is the number of functions. The algorithm is described in the following listing:

```
1  calculate_time(event e, function foo) {
2    if(lastsync.id == e.id) { // We have a pair of
       synchronizations
3      if(stack.top != NULL) {
4        stack.top.time = stack.top.time + t_foo^local;
5      }
6      return t_foo^local;
7    } else {  // We are starting a new pair of synchronizations
8      // Push measured time on the stack
9      newnode.id = lastsync.id;
10     newnode.time = t_foo^local;
11     stack.push(newnode);
12
13     // Add lastsync.id to the bit field
14     bitfield[lastsync.id] = 1;
15
16     // Clean old pairs off the stack
17     for(each node n in stack) {
```

```
18          if(n.id == lastsync.id) {
19            n.nextnode.time = n.nextnode.time + n.time;
20            todelete = n;
21            n = n.nextnode;
22            delete(todelete);
23            // Remove the old id from the bit field
24            bitfield[n.id] = 0;
25          }
26        }
27
28        // Calculate a time for foo
29        if(bitfield[e.id] == 1) {
30          t_foo^local = 0;
31          n = stack.top;
32          do {
33            t_foo^local = t_foo^local + n.time;
34            n = n.next;
35          } while(n.id != e.id);
36        }
37
38        return t_foo^local;
39      }
40    }
```

8 Conclusion and Future Work

We have presented a scalable method for detecting load imbalance and quantifying the impact of load imbalance in petascale systems. Our method uses the SILAS simulator to verify hypotheses of load balancing performance benefits. As this method is still conceptual, there is a great deal of future work to be done.

A prototype implementation should be tested against a range of examples. This will no doubt suggest changes to the method. Implementations of both trace-based and stack-based function time calculation should be carefully analyzed to determine which is more feasible, and a representation of collective communication events should be selected (i.e. a set of point-to-point synchronizations, or a another class of synchronization).

Other statistical metrics should also be considered. The number of times a function is called is likely to be an important metric. A high correlation between a surplus of function calls and process wait time will suggest investigating call path imbalance. Also, metrics such as the p-values and confidence intervals will aid the automatic generation of hypothesis.

The measurement matrix M is never explicitly formed, yet a complete M may be useful to other deductive applications. A scalable algorithm for forming M would perhaps prove to be useful.

References

[1] Robert Bell, Allen D. Malony, and S. Shende. Paraprof: A portable, extensible and scalable tool for parallel performance profile analysis. In *EuroPar'03: Proceedings of the 2007 European Conference on Parallel Computing*, pages 17–26, Berlin, 2003. Springer.

[2] J. M. Bull. *A hierarchical classification of overheads in parallel programs*, pages 208–219. Chapman and Hall, 1996.

[3] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1):22–31, January/March 1999.

[4] Mark E. Crovella and Thomas J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. In *Proceedings of Supercomputing '94*, pages 600–609, Washington, DC, USA, November 1994. IEEE Computer Society.

[5] Bronis R. de Supinski, Rob Fowler, Todd Gamblin, Frank Mueller, Prasun Ratn, and Martin Schulz. An open infrastructure for scalable, reconfigurable analysis. In *International Workshop on Scalable Tools for High-End Computing (STHEC 2008) ACM/SIGARCH*, July 2008.

[6] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting application load imbalance on high end massively parallel systems. In *EuroPar'07: Proceedings of the 2007 European Conference on Parallel Computing*, pages 150–159, 2007.

[7] Luiz DeRose, Bill Homer, Dean Johnson, and S. Kaufmann. The new generation of cray tools. In *CUG'05: Proceedings of Cray Users Group Meeting*, May 2005.

[8] Jialin Dou and Marcelo Cintra. A compiler cost model for speculative parallelization. *ACM Transactions on Architecture and Code Optimization*, 4(2):12, 2007.

[9] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. Technical Report TR09-05, Department of Computer Science, University of Massachusetts Amherst, 2005.

[10] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez. Flexible coscheduling: mitigating load imbalance and improving utilization of heterogeneous resources. In *IPDPS'03: Proceedings of the 2003 International Parallel and Distributed Processing Symposium.*, April 2003.

[11] T. Gamblin, B. R. Supinski, M. Schulz, R. J. Fowler, and D. A. Reed. Scalable load-balance measurement for spmd codes. *Submission*, 2008.

[12] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM/MPI Conference*, Bonn, Germany, September 2006. Springer.

[13] S. Grahm, P. Kessler, and M. McKusic. `gprof`: A call graph execution profiler. In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, pages 120–126, Boston, MA, USA, June 1982. Association for Computing Machinery.

[14] Marc-André Hermanns. Trace-based performance simulation of large-scale applications. Master's thesis, Forschungszentrum Jülich, 2008.

[15] Marc-André Hermanns, Markus Geimer, Felix Wolf, and Brian J. N. Wylie. Verifying causal connections between distant performance phenomena in large-scale message-passing applications. Technical Report FZJ-JSC-IB-2008-05, Forschungszentrum Jülich, April 2008.

[16] Marc-André Hermanns, Markus Geimer, Felix Wolf, and Brian J. N. Wylie. Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *PDP 2009*. IEEE Computer Society, 2009.

[17] A. Hoisie, D. J. Kerbyson, S. Pakin, F. Petrini, H. J. Wasserman, and J. Fernandez-Peinador. Identifying and eliminating the performance variability on the asci q machine. Technical Report LA-UR-03-0138, Los Alamos National Lab, January 2003.

[18] Paul W. Holland. Statistics and causal inference. *Journal of the American Statistical Association*, 81(396):945–960, 1986.

[19] Kevin A. Huck, Allen D. Malony, Robert Bell, and Alan Morris. Design and implementation of a parallel performance data management framework. In *ICPP'05: Proceedings of the 2005 International Conference on Parallel Processing*, 2005.

[20] Rick Kufrin. Perfsuite: An accessible, open source performance analysis environment for linux. In *Proceedings of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, Chapel Hill, NC, April 2005.

[21] International Business Machines. Ibm system blue gene solution. http://www-03.ibm.com/systems/deepcomputing/bluegene/, July 2008.

[22] John Mellor-Crummey, Robert Fowler, and Gabriel Marin. Hpcview: A tool for top-down analysis of node performance. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, October 2001.

[23] P. Mucci, J. Dongara, R. Kufrin, S. Moore, F. Song, and F. Wolf. Automating the large-scale collection and analysis of performance. In *Proceedings of the*

5th LCI International Conference on Linux Clusters: The HPC Revolution, Austin, TX, USA, May 2004.

[24] William Navidi. *Statistics for Engineers and Scientists*. McGraw-Hill Science/Engineering/Math, 1^{st} edition, December 2004.

[25] University of Oregon Performance Research Labs. Tuning and analysis utilities: Tau. http://www.cs.uoregon.edu/research/tau, July 2008.

[26] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[27] Apan Qasem, Ken Kennedy, and John Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, 36(9):183–196, May 2006.

[28] Luiz De Rose, Ying Zhang, and Daniel A. Reed. SvPablo: A multi-language performance analysis system. *Lecture Notes in Computer Science*, 1469:352, 1998.

[29] Rizos Sakellariou and John R. Gurd. Compile-time minimisation of load imbalance in loop nests. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 277–284, New York, NY, USA, 1997. ACM.

[30] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *MICRO'05: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture.*, November 2005.

[31] Jeffrey Vetter, editor. *Workshop on Software Development Tools for Petascale Computing*, Washington, D.C, August 2007. Department of Energy.

[32] Jeffrey Vetter and Chris Chambreau. mpip: Lightweight, scalable mpi profiling. http://mpip.sourceforge.net/, April 2008.

[33] Jr. Wagner Meira, Thomas J. LeBlanc, and Alexandros Poulos. Waiting time analysis and performance visualization in carnival. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 1–10, New York, NY, USA, 1996. ACM.

[34] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[35] Felix Wolf, Daniel Becker, Markus Geimer, and Brian J. N. Wylie. Scalable performance analysis methods for the next generation of supercomputers. In *Proceedings of the John von Neumann Institute for Computing (NIC) Symposium*, volume 39 of *NIC-Series*, Jülich, Germany, February 2008.