Scalable Critical-Path Based Performance Analysis

David Böhme
and Felix Wolf
German Research School
for Simulation Sciences
52062 Aachen, Germany
Email: {d.boehme,f.wolf}@grs-sim.de

Bronis R. de Supinski
and Martin Schulz
Lawrence Livermore National Laboratory
Livermore, California
Email: {bronis,schulzm}@llnl.gov

Markus Geimer
Jülich Supercomputing Centre
52425 Jülich, Germany
Email: m.geimer@fz-juelich.de

***Abstract*—The critical path, which describes the longest execution sequence without wait states in a parallel program, identifies the activities that determine the overall program runtime. Combining knowledge of the critical path with traditional parallel profiles, we have defined a set of compact performance indicators that help answer a variety of important performance-analysis questions, such as identifying load imbalance, quantifying the impact of imbalance on runtime, and characterizing resource consumption. By replaying event traces in parallel, we can calculate these performance indicators in a highly scalable way, making them a suitable analysis instrument for massively parallel programs with thousands of processes. Case studies with real-world parallel applications confirm that—in comparison to traditional profiles—our indicators provide enhanced insight into program behavior, especially when evaluating partitioning schemes of MPMD programs.**

I. Introduction

Rising numbers of cores per socket in combination with an architecture ecosystem characterized by increasing diversification and heterogeneity continue to complicate the development of efficient parallel programs. Particularly load imbalance, which frequently appears during simulations of irregular and dynamic domains—a typical scenario in many engineering codes—presents a key challenge to achieving satisfactory parallel efficiency. This challenge creates a strong need for effective performance-analysis methods that highlight load balancing issues occurring at larger scales. To be useful to an application developer, these methods must not only be scalable themselves but also require as little effort as possible to interpret the results.

Originally developed as a tool for planning, scheduling, and coordinating complex engineering projects [1], which usually involve concurrent activities, the notion of the critical path is also helpful in understanding the performance of parallel programs. Critical-path analysis models the execution of a parallel program as a program activity graph (PAG), which represents the duration and precedence order of individual program activities. The *critical path*, which is the longest path through the graph, determines the length of program execution. The critical path itself does not have any wait states but may induce them in other execution paths. Increases in the time for *critical activities*, which are those on the critical path, prolong the overall execution, while shortening critical activities reduces it. However, the overall reduction may be less than that of the activity as the change may shift the critical path to include other activities. Thus, while knowledge of the critical path can identify activities for which optimizations will prove worthwhile, knowledge of the impact of those optimizations on the critical path can guide optimization efforts more precisely.

However, in spite of its apparent utility, as of today critical-path analysis plays only a minor role in the optimization of supercomputing applications. The *single program multiple data* (SPMD) paradigm that is used in many, if not most, current parallel codes contributes to the failure to embrace such analysis. Since every process has a nearly identical sequence of activities, critical-path analysis often fails to reduce the activities under consideration significantly for optimization. The iterative nature of numerical applications, which causes many activities to appear on the path multiple times, further reduces the value of critical-path analysis. These two characteristics also set supercomputing applications apart from typical engineering projects, for which the methodology was originally invented. Also, the sheer length of the critical path in realistic programs makes it an unwieldy data structure to analyze manually with reasonable effort.

Despite these shortcomings, the fundamental properties of the critical path remain. A critical path measurement contains essential information for the optimization and load balance of parallel codes. However, we need new ways to interpret and to analyze it without sacrificing scalability.

In this paper, we present a novel and scalable performance-analysis methodology based on knowledge of the critical path. We propose several compact *performance indicators* that illuminate the relationship between critical and non-critical activities to guide the analysis of complex load-imbalance phenomena intuitively. Similar to economic indicators such as consumer price index or gross domestic product, which characterize things as complex as a nation's economic well-being in terms of a few numbers, performance indicators improve the understanding of labyrinthine program behavior without letting the user drown in a flood of performance details. The main difference to classic performance metrics such as number of messages is the higher level of abstraction that these indicators provide. While also offering insight into classic SPMD programs, our indicators especially suit programs with a *multiple program multiple data* (MPMD) structure, which is popular among the increasing number of multi-physics codes.

Specifically, we make the following contributions:

- *Three novel performance indicators:* We base our three performance indicators on critical-path profiles, a compact representation of critical activities and their distribution across the process space. The first one, for SPMD programs, targets general parallel efficiency, while the other two, for MPMD programs, characterize load imbalance within and across different partitions .
- *A scalable method for calculating these performance indicators:* We design and implement a method to extract the critical path and to calculate our indicators based on a parallel replay of event traces that scales naturally with the target application.
- *Evaluation of our indicators on a broad range of real-world applications:* We first show the general applicability and utility of our method across a broad range of applications, before we present in-detail analyses of both an SPMD and an MPMD code to demonstrate how our performance indicators provide critical insights into complex cases of load imbalance that traditional metrics cannot provide. These results also prove the scalability of our methodology for up to several thousand processes.
- *Integration with a production-level tool:* We integrate our methodology into the Scalasca performance analysis tool [2], which will make it available to application developers worldwide.

Our paper is organized as follows. After reviewing the theory of critical-path analysis, we describe our general methodology along with the definitions of our performance indicators in Section II. Section III is devoted to their scalable calculation. In Section IV, we evaluate our indicators experimentally. Finally, we discuss related work in Section V, before we conclude the paper and outline future work in Section VI.

II. Critical-path analysis

This section defines our base terminology and describes the basic concept of the critical path. Building upon this foundation, we then introduce our three novel performance indicators and outline how they assist in pinpointing optimizations.

Our analysis combines critical-path data with per-process performance data. The critical path provides an overview of the most time-consuming activities, but does not capture important parallel performance characteristics such as load balance. Alternatively, per-process profiles do not capture dynamic effects that characterize a program's execution. Our work combines critical-path and per-process profiles to characterize load balance and to highlight typical parallelization issues more reliably than per-process profiles alone.

We follow Scalasca's concept of automatically extracting patterns of inefficiency from detailed event traces that are too unwieldy to explore manually in their entirety. Thus, we combine critical-path and per-process profiles to derive a set of compact *performance indicators*. These indicators provide intuitive guidance about load-balance characteristics that quickly draw attention to potentially inefficient code regions.

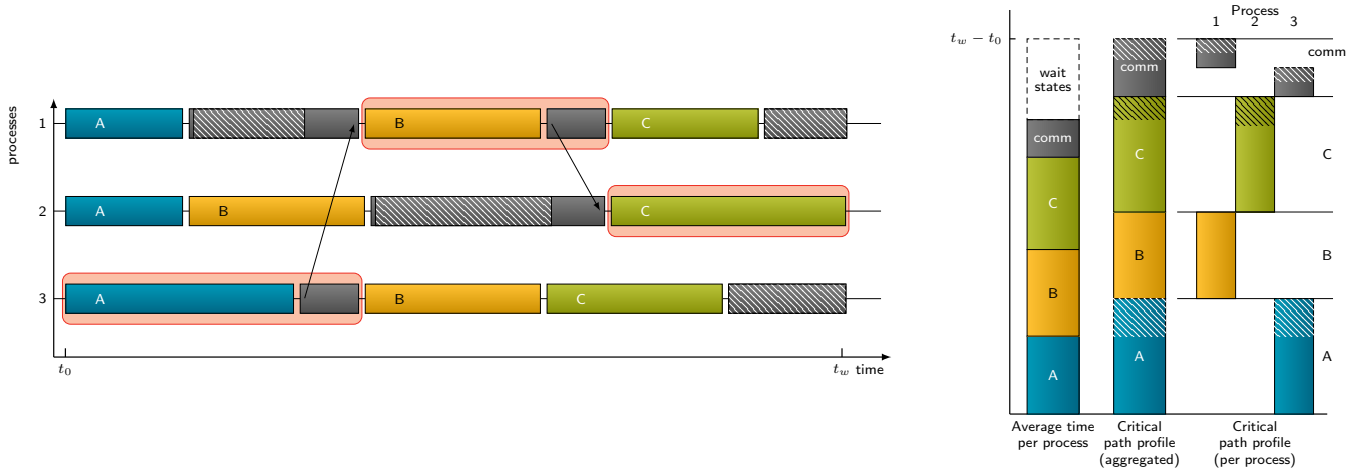
Our critical-path analysis produces two groups of performance data structures. The first group, the *critical-path profile* and the *critical-path imbalance indicator*, describes the impact of program activities on execution time. The critical-path profile represents the time that an activity spends on the critical path. The imbalance indicator captures how much time in a call path is lost due to load imbalance. These two metrics provide an easily accessible overview of promising optimization targets and parallel inefficiencies in SPMD programs.

The second group, which we call performance impact indicators and which consists of the *intra-* and *inter-partition imbalance* indicators, describes how program activities influence resource consumption. These indicators are especially useful for the analysis of MPMD programs. In particular, they classify load imbalance in MPMD programs by origin, and can distinguish if resource waste is a result of an uneven distribution of process partitions, or of imbalance among processes performing the same activity within a single partition. In the following, we explain the concept of the critical path and our performance indicators in more detail.

A. The critical path

Figure 1(a) shows the execution of a parallel program as a time-line diagram with a separate time line for each process. We assume that the entire parallel resource is allocated from program start t_0 until program end t_w. Conceptually, processes that finish earlier than t_w idle in a wait state until the last process finishes (pseudo-synchronization). Thus, the total resource consumption corresponds to the number of processes P multiplied by $t_w - t_0$. Additional wait states may occur at synchronization points during execution due to load or communication imbalance. In the following, we use the term *wall-clock time* when we refer to fractions of the length of execution and the term *allocation time* when we refer to fractions of resource consumption. While the wall-clock time is always a value between zero and t_w, the allocation time can be as large as $P * (t_w - t_0)$.

An *execution path* through the process-time space is a sequence of activities that are connected by time-line sections or communication edges. An *activity* corresponds to a single execution of a particular call path by a specific process excluding inherent wait states. The *call path* identifies not only the code location of an activity, but also includes the sequence of regions/functions entered on the way to that specific point in the execution (e.g., main() $\rightarrow$ foo() $\rightarrow$ bar()). Using the call path instead of a function name alone allows invocations of the same function to be distinguished by the parent function in which they occur. The *critical path* is the longest execution path in the program that does not include wait states. Therefore, it determines the runtime of the program: any increase in the computation time of its activities or its communication times will increase application runtime. An optimization on the critical path may decrease runtime; but the improvement is not guaranteed since an execution may have multiple critical or near-critical paths. In contrast,



(a) SPMD program time-line diagram. Each rectangle represents an activity. Arrows between processes denote communication; the hatched areas inside communication activities represent wait states. Highlighted activities mark the critical path. Processes 1 and 3 end with wait states due to pseudo-synchronization.

(b) Average activity time and critical-path profile (aggregated and per process); hatched areas denote critical-path imbalance.

Fig. 1. The critical path and the associated critical-path profile of an SPMD program.

optimizing an activity not on the critical path only increases waiting time, but does not affect the overall runtime.

B. Critical-path profile

The *critical-path profile* represents the total amount of (wall-clock) time each activity spends on the critical path. Figure 1(b) illustrates the critical-path profile for the example program run that Figure 1(a) shows. Mathematically, the critical path profile is defined as a mapping that assigns to each combination of activity (i.e., call path) and process the time attributable to the critical path. The critical-path profile can provide a simple overview of the most time-consuming subroutines, as shown in the middle bar, but also allows detailed insight into each process's time share on the critical path, as illustrated by the bars on the right of Figure 1(b).

C. Critical-path imbalance indicator

The critical-path imbalance indicator ι is the difference between a call path's contribution to the critical path and the average time spent in the call path across all processes. Precisely, we define it for a critical-path activity i as:

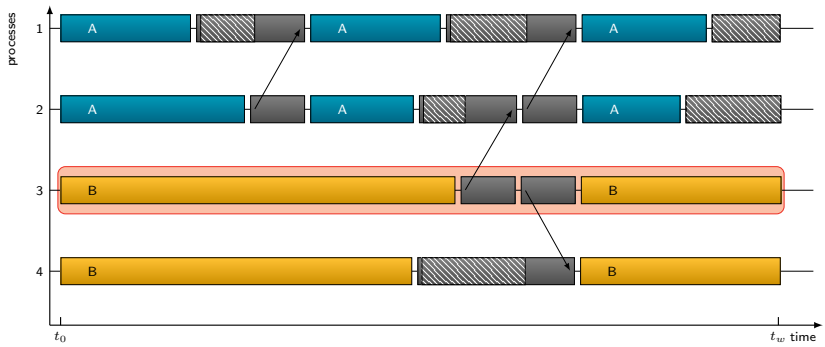
$$\iota(i) = \max\left(d_{cp}(i) - \text{avg}(i),\, 0\right)$$
$$\text{avg}(i) = \frac{1}{P}\sum_{p=1}^{P} d_p(i)$$

where $d_{cp}(i)$ denotes the duration of activity i on the critical path, $d_p(i)$ represents the duration of the activity on process p without wait states, and P is the number of processes. Since an imbalance only affects overall execution time if the time on the critical path is larger than the average, we only include positive values. Figure 1(b) illustrates the concept graphically. The critical-path imbalance is the hatched area of the critical-path profile bars. Activity B exhibits no critical-path imbalance since it is perfectly balanced. Activities A and C as well as the communication activities exhibit critical-path imbalance, indicating some inefficient parallelism in these activities.

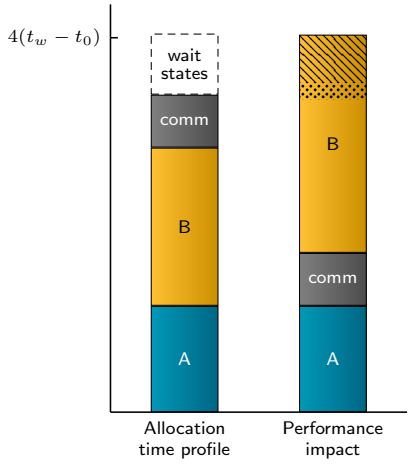
Essentially, the amount of critical-path imbalance in a program corresponds to the wall-clock time that is lost due to inefficient parallelization, compared to a perfectly balanced program. Thus, the critical-path imbalance indicator provides clear guidance in discovering parallelization bottlenecks. Also, the ratio of critical-path imbalance and the total time of an activity on the critical path provides a useful load imbalance metric. Thus, it provides similar guidance as prior profile-based load imbalance metrics (e.g., the load-imbalance percentage metrics defined in CrayPat [3]), but the critical-path imbalance indicator can often draw a more accurate picture. The critical path retains dynamic effects in program execution, such as shifting of imbalance between processes over time, which per-process profiles simply cannot capture. Thus, purely profile-based imbalance metrics regularly underestimate the actual performance impact of a given load imbalance. As an extreme example, consider a program in which a function is serialized across all processes but runs for the same amount of time on each. Purely per-process profile based metrics would not show any load imbalance. Our critical-path imbalance indicator correctly characterizes the function's serialized execution as a performance bottleneck. In Section IV-A, we demonstrate this effect using a synthetic benchmark.

D. Performance-impact indicators

Existing approaches to the characterization of load imbalance in parallel programs mostly target SPMD codes. These approaches often do not work with MPMD programs, in which different groups of processes (*process partitions*) execute entirely different activities. Often, multi-physics MPMD programs combine multiple SPMD codes which run in separate process partitions. The partitioning complicates determining



(a) Time-line diagram of an MPMD program run, showing two partitions that execute different activities; highlighted activities mark the critical path, hatched areas represent wait states.



(b) Allocation time profile and performance impact; the hatched area represents inter-partition imbalance costs, the dotted area intra-partition imbalance costs.

Fig. 2. Parallel profile and performance impact indicators for an MPMD program.

the performance impact of load imbalance in a single code region on the overall runtime. Achieving optimal load balance in MPMD codes typically also involves runtime configuration adjustments, such as finding optimal process partition sizes. Developers currently must use trial-and-error methods to find suitable configurations due to the lack of proper tool support. Our performance impact indicators provide improved insight into load balance issues in MPMD programs, which will simplify the search for optimal configurations.

1) Intra- and inter-partition imbalance: To characterize complex load-balance issues that arise in MPMD programs, we define an extended load imbalance classification scheme. We distinguish between *intra-* and *inter-partition* imbalance. Intra-partition imbalance describes load imbalance within an (SPMD) process partition, while inter-partition imbalance describes load imbalance across process partitions.

2) Performance impact: To obtain an accurate picture of parallel efficiency in MPMD programs in which different code paths execute in parallel, we cannot simply consider activities independently. Instead, we must identify those activities that determine the program runtime (i.e., those on the critical path) and compare their resource consumption with the resource consumption of remaining activities. A difference in resource consumption of critical and non-critical activities indicates load imbalance in the program.

Our performance-impact indicators determine the contribution of specific activities to the overall resource consumption of MPMD programs (in particular) and identify where intra- or inter-partition imbalance wastes resources. We base the concept on the notion that the activities on the critical path determine the runtime of the program and, therefore, its overall resource consumption. Further, we distinguish between the amount of resources that activities consume directly and the amount that wait states occupy. Then, we map the resources that wait states occupy as *imbalance costs* onto the activities on the critical path. The sum of the time consumed by the activity itself across all processes and its imbalance costs describe the total performance impact of the activity. We distinguish imbalance costs based on their impact on intra- and inter-partition imbalance to provide additional insights for MPMD programs. High imbalance costs suggest a performance bottleneck, and the ratio of imbalance costs and overall performance impact of an activity provides a metric for load balance. In a well-balanced program, the aggregated imbalance costs are low.

The time-line diagram in Figure 2(a) shows an MPMD program run in which one partition of the processes executes activity A, and another partition executes activity B, with the critical path running entirely on process 3. In this example, we classify the underload in activity B on process 4 as intra-partition imbalance costs; and we classify the resource consumption due to the wait states on processes 1 and 2 as inter-partition imbalance costs. Figure 2(b) shows the parallel allocation time profile of the program on the left and the performance impact on the right. Since activity B is the only activity on the critical path, it accumulates the entire critical-path imbalance costs; hence, these costs are added as imbalance costs onto the performance impact of activity B. In this example, inter-partition imbalance costs (hatched area) are roughly three times as large as the intra-partition imbalance costs (dotted area). As a result, serial optimizations to activity B or rebalancing the partitions to assign more processes to activity B (or fewer processes to activity A) promise greater benefits with respect to resource consumption than load-balancing activity B within its partition.

The mapping of imbalance costs onto critical activities does not necessarily reflect the direct causes of wait states. Instead, the imbalance cost indicators are a heuristic to highlight optimization possibilities that are most likely to improve

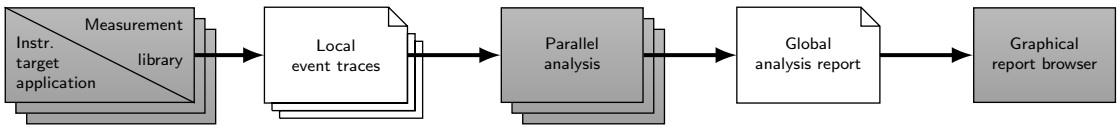


Fig. 3. Scalasca's parallel trace-analysis workflow; gray rectangles denote programs and white rectangles denote files. Stacked symbols denote multiple instances of programs or files, run or processed in parallel.

performance. The imbalance costs represent an upper bound of the achievable performance improvement.

3) Calculating the imbalance costs: We can easily compute the imbalance costs in parallel based on knowledge of the critical-path profile. We calculate the local imbalance costs on each process and then aggregate them using a global reduction operation. Basically, on each process, we identify activities that take more time on the critical path than on the process and assign imbalance costs to these activities.

In detail, we first compute the critical-path headroom h_p for each process p. This auxiliary construct is the difference between the duration of the critical path and the duration of the activities on p, which equals the overall waiting time on the process. We then proportionally assign this headroom to the call paths of those activities that spend less time on the process than on the critical path. Thus, for each critical-path activity i we calculate its critical-path excess time $\delta(i)$:

$$\delta(i) = \max\left(d_{cp}(i) - d_p(i),\, 0\right)$$

We then apply a scaling factor α:

$$\alpha = \frac{h_p}{\sum_j \delta(j)},$$

that matches the sum of the excess times to the headroom, to obtain the local imbalance costs $\alpha\delta(i)$ for activity i. If the critical-path activity i does not occur on the local process at all, its imbalance costs are inter-partition imbalance costs; otherwise they are intra-partition imbalance costs.

III. Implementation

We implement our critical-path analysis as an extension to the automatic wait-state detection of the Scalasca performance-analysis tool, leveraging its scalable trace-analysis approach. In the following, we describe how we extract the critical path from event traces in a scalable way, and how we use it to calculate the performance indicators that we introduced in Section II. We start with a brief review of Scalasca's event-tracing methodology [2].

A. Scalable trace analysis in Scalasca

Scalasca is a performance-analysis toolset specifically designed for large-scale systems, such as IBM Blue Gene and Cray XT/XE. It provides an integrated performance-analysis approach and includes support for call-path profiling. It features a scalable, automated, post-mortem event trace analysis that can identify wait states that occur in program execution, for example, as the result of unevenly distributed workloads.

Figure 3 illustrates Scalasca's trace-analysis workflow. To collect event traces, we first instrument the target application, that is, we insert extra code that intercepts relevant events at runtime at specific program points. Such events include entering and leaving of source-code regions and sending and receiving of messages. These instrumentation hooks generate corresponding event records, which are subsequently stored in a memory buffer before they are flushed to disk, typically at the end of measurement. Limiting tracing to selected intervals supports focused analyses of long-running codes.

After the target application finishes execution and writes its trace data, we launch the trace analyzer with one analysis process per (target) process. This approach exploits the distributed memory and processing capabilities of the underlying parallel system, which is the key to achieving good scalability. During *forward replay*, the analyzer traverses the traces in parallel, iterating over each process-local trace from beginning to end. The analyzer re-enacts the original communication, exchanging the data required for the wait-state search at each recorded synchronization point using communication similar to that originally used by the program. The analysis identifies wait states in synchronization and communication operations by measuring temporal differences between local and remote events. We categorize every wait-state instance detected by type, and accumulate the associated waiting time in a process-local [type, call path] matrix. At the end of the trace traversal, Scalasca gathers the distributed analysis results into a three-dimensional [metric, call path, process] structure that characterizes the entire experiment. Scalasca then stores this global analysis result on disk for subsequent interactive examination with a graphical report explorer.

B. Critical-path detection

We now present our technique to obtain a critical-path profile to calculate our performance indicators. We build directly on Scalasca's parallel trace replay to extract the critical path from an event trace in a scalable way. The critical-path extraction algorithm needs information on which MPI operation instances incur wait states. Thus, we need the results of the wait-state analysis outlined in Section III-A to start the critical-path search. We extend the wait-state search algorithm to annotate communication/synchronization events at which wait states occur. The critical-path search then again replays the trace. Since we must know the end of the execution to determine the critical path, we perform this pass in backward

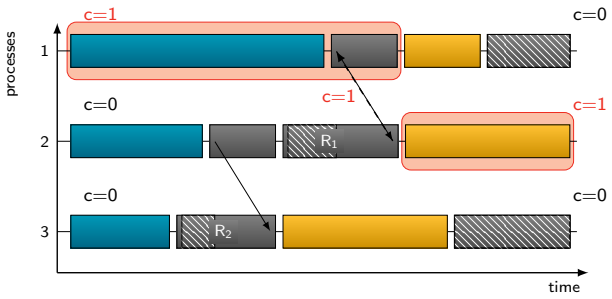
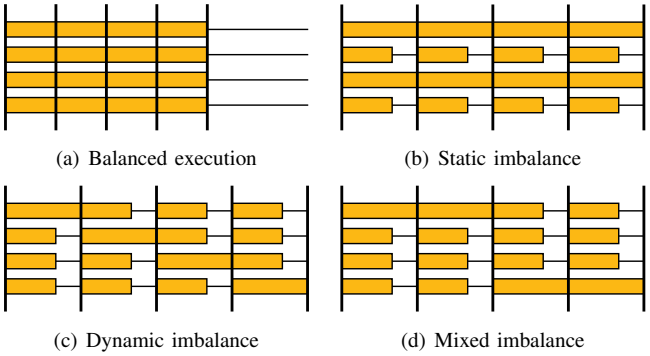


Fig. 4. Backward replay critical-path detection. The red critical path ends on process 2; the critical-path token c moves to process 1 at the synchronization.



(a) Balanced execution (b) Static imbalance

(c) Dynamic imbalance (d) Mixed imbalance

Fig. 5. Time-line diagrams illustrating the time distribution of the work() function in the synthetic load balance scenarios.

direction, replaying the trace in reverse order. During the backward replay, we follow the critical path from the end to the beginning. In the unlikely case of more than one critical path, our algorithm arbitrarily selects one of them.

For MPI programs, the critical path runs between MPI_Init and MPI_Finalize. As the backward replay starts at the end of the trace, we first determine the MPI rank that entered MPI_Finalize last. This point marks the endpoint of the critical path. Since the critical path only runs on a single process at any given time, we set a flag on the corresponding analysis process to signal its "ownership" of the critical path.

The rest of the critical-path search exploits the lack of wait states on the critical path. As the backward replay progresses, the critical path remains on the flagged process until it reaches a communication event in the trace that the wait-state analysis pass marked as incurring waiting time. In this case, the critical path continues at the corresponding communication event on the process that is responsible for the wait state (i.e., the communication peer). We transfer critical-path ownership to this process via backward communication replay. In the case of a point-to-point communication event, the replay uses a point-to-point message for the ownership transfer; in the case of a collective communication event, it uses a reduction operation.

Figure 4 illustrates the backward replay, which starts from the end of the trace shown on the right. Since process 2 finished execution last, it owns the final critical path segment. Moving backwards through the trace, we find a wait state at communication event R_1. Now, the critical-path flag moves to the wait state's origin on process 1 using a point-to-point message transfer from the original receiver to the original sender. During the course of the replay, the processes that own the critical-path flag accumulate their individual contributions to the critical-path profile locally. While in principle our approach can capture the entire dynamic structure of the critical path, Scalasca currently collects and reports only the critical-path profile. However, future extensions that make use of the dynamic structure are conceivable, such as a trace time-line display with critical-path highlighting.

After Scalasca completes the critical-path extraction, we derive our performance indicators. The nature of this task lends itself to parallel computation. We therefore accumulate the global critical-path profile using a global reduction operation and distribute it to all processes, which then calculate their individual contributions to each indicator.

IV. Evaluation

To evaluate the critical-path search and our critical-path based performance indicators, we apply our analyses to measurements of several real-world simulation codes. However, we first demonstrate the benefit of critical-path based load imbalance analysis using a small synthetic benchmark. We then present our experiences from analyzing the SPECMPI 2007 benchmark suite to demonstrate the applicability of our methods to a broad set of HPC codes; afterwards we discuss the application of our performance indicators using an SPMD example (PEPC) and an MPMD example (ddcMD). We perform the SPECMPI experiments on the Intel/Infiniband cluster system Juropa and the PEPC and ddcMD experiments on the 72-rack IBM Blue Gene/P supercomputer Jugene at the Jülich Supercomputing Centre.

A. Synthetic load imbalance benchmark

To validate the benefit of our critical-path based load imbalance characterization in comparison to traditional profile-based solutions, we apply it to a small benchmark application that can simulate different load-imbalance scenarios. The application executes a loop in which it runs a work() function for a configurable amount of time and synchronizes all processes with a barrier at the end of each loop iteration.

We run four different scenarios using 32 processes on the Juropa cluster. The overall workload (i.e., the total CPU allocation time spent in the work function) is constant in the scenarios, but the workload distribution across the processes varies. In the first scenario, the simulated workload is perfectly balanced across all processes. We run three imbalanced scenarios (*static*, *dynamic*, and *mixed*) in which we increase the program runtime by 25% compared to the balanced scenario with three different ways of adding artificial imbalance to the work function. In the static imbalance scenario, each process retains a constant (positive or negative) deviation from the average workload in every iteration. In the dynamic imbalance scenario, a different process in every iteration spends more time in work() than the others; however, the overall aggregate

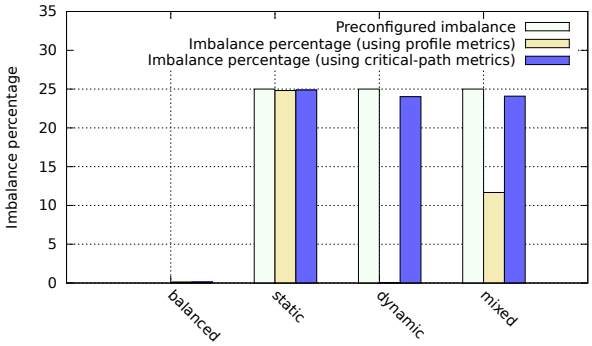


Fig. 6. Imbalance detected in artificial load imbalance example using critical-path and per-process profile metrics.

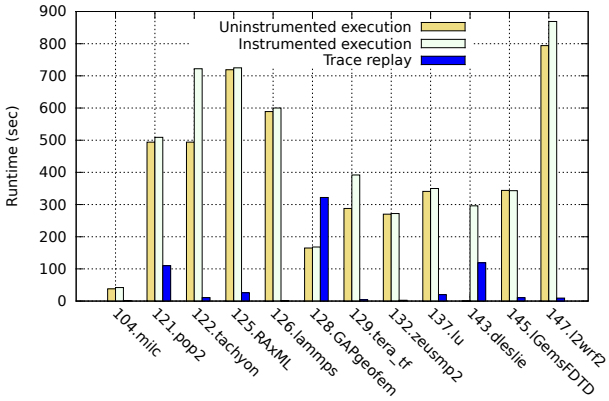


Fig. 7. Uninstrumented and instrumented runtimes and trace replay time for SPECMPI benchmarks. (Juropa cluster, 256 processes)

workload is equal on every process (i.e., the workload is statically balanced). Finally, in the mixed imbalance scenario, one process is overloaded for the first half of the iterations, and a different one for the remaining iterations. Figure 5 shows time-line diagrams that illustrate the scenarios.

We let each scenario run 320 loop iterations. In total, the balanced scenario takes 16.15 seconds (wall-clock time), and the scenarios with artificial imbalance take between 19.98 and 20.04 seconds. The critical-path analysis uncovers between 3.87 and 3.99 seconds of critical-path imbalance in the work function for each of the imbalanced scenarios, closely matching the preconfigured 25% runtime increase (which amounts to 4 seconds) inflicted by the artificial imbalance. The critical-path imbalance indicates to a user that improving the load balance of the work function can save up to 3.99 seconds of runtime in each of the imbalanced scenarios. We can draw the same conclusion based on per-process metrics, such as the maximum and average time spent in work(), for the static imbalance scenario. However, those metrics are insufficient for the dynamic imbalance scenario, in which the aggregate time spent in the work function is the same on each process. Here, the profile-based metrics show that the program spends a significant amount of time in barrier synchronization, but we cannot relate this time to a specific load imbalance.

Figure 6 compares the amount of load imbalance that the critical-path imbalance indicator and per-process profile metrics (using the difference of maximum and average workload) find. The figure shows the imbalance percentage, that is, the ratio of the detected imbalance and the average workload. Unsurprisingly, none of the methods finds any imbalance in the balanced scenario. For the static imbalance scenario, both methods correctly determine approximately 25% load imbalance in the work function, which corresponds to the preconfigured amount. However, the profile-based method fails to detect any load imbalance in the dynamic scenario and detects only half of the actual imbalance in the mixed scenario. Hence, relying on per-process profiles alone can severely underestimate dynamically changing load imbalances while our critical-path imbalance indicator accurately determines the impact of load imbalance under all circumstances.

B. SPECMPI 2007

The SPECMPI benchmark suite [4] consists of various real-world MPI codes that represent a broad range of application domains and parallelization schemes. We use 256 MPI process runs of each of the eleven SPECMPI applications for which a large reference input data set ("lref") is available. We also include the 104.milc benchmark, for which only a medium-size ("mref") configuration is provided. The "lref" benchmarks take between 5 and 15 minutes each on our cluster. For our event tracing experiments, we create custom filters for each application to omit frequently executed but otherwise unimportant user functions from the trace. With the exception of 128.GAPgeofem, for which we had to decrease the number of iterations from 10,000 to 1,500 due to a massive amount of MPI point-to-point events, we can perform trace measurements for all of the original reference benchmark configurations, producing between 170MB and 9GB of compressed trace data per application. Figure 7 compares the runtime of the uninstrumented benchmark executables[1] with their runtimes while recording the trace (excluding the additional overhead of writing trace data to disk at the end of the execution, which does not perturb measurement itself). The comparison shows that the measurement perturbation is acceptably low in most cases, only 122.tachyon, 129.tera_tf and 147.l2wrf2 show more than 10% dilation. With more advanced methods such as direct binary instrumentation combined with prior static analysis [5], we expect to reduce instrumentation overhead further to less than 10% in all cases.

Figure 7 also shows the time for parallel trace replay, including wait-state search and critical-path analysis. In all cases other than the communication-intensive 128.GAPgeofem, analysis time is negligible compared to the runtime.

Since our study does not intend to survey the performance characteristics of all SPECMPI benchmarks, we do not discuss each benchmark in detail. Instead, we consider only selected cases for which the critical-path based metrics provide insight. Figure 8 shows the imbalance percentage, a measure of the

[1]The uninstrumented 143.dleslie executable segfaulted on our system.

overall load imbalance in the execution, derived from profile-based metrics (difference of maximum and average non-waiting execution time per process) and from critical-path metrics (ratio of critical imbalance and critical-path length). As we explained in Section II-C and demonstrated in the artificial imbalance example, the critical-path based approach incorporates dynamic execution effects, such as shifting of load between processes over time, that profile-based metrics miss. We would therefore expect the imbalance determined by the critical-path approach to be higher than the imbalance calculated from per-process profiles. Indeed, in 9 out of 12 cases, the profile-based metrics underestimate the overall imbalance, which indicates that the dynamic effects that the critical-path based metrics address are a common phenomenon.

The master-worker code, 125.RAxML, is interesting. The critical path and our performance impact indicators are ideally suited for the analysis of this parallelization scheme; in particular, we can easily assess the load balance, and determine whether the master process is a bottleneck. On an abstract level, we can regard a master-worker parallelization as MPMD, with the master forming one (trivial) partition and all workers forming another. In the case of 125.RAxML, the analysis shows that the program's main loop is well-balanced and the master process is not a bottleneck, but we identify a serial initialization procedure on the master process that is on the critical path as a source of inter-partition imbalance. Related problems also occur in 145.lGemsFDTD and 147.l2wrf2, in which serial initialization routines, executed only on the root process, become performance bottlenecks. In all cases, the inter-partition imbalance indicator quickly identifies the responsible function. In 104.milc, 11 out of the 256 allocated cores do not participate in the computation at all, resulting in 4% inter-partition imbalance cost. Finally, the critical-path imbalance indicators also detect imbalanced functions in SPMD codes. For example, in 145.lGemsFDTD, the function `nft_class.nft_apply` uses 11.4% of the accumulated allocation time, but is responsible for 41% of the load imbalance costs in the program. Including the imbalance costs, the overall performance impact of this function is actually 18.2% of the total allocation time. The load imbalance alone wastes more than 18 seconds of wall-clock time. In conclusion, the SPECMPI examples show that we can straightforwardly apply our critical-path analysis to a broad range of real-world HPC codes to identify performance bottlenecks.

C. PEPC

PEPC ("Pretty Efficient Parallel Coulomb-solver") is an SPMD N-body simulation code developed by the Simulation Laboratory Plasma Physics at the Jülich Supercomputing Centre [6]. We use the 1.4 benchmark version of the code for our experiments and simulate 1.6 million particles over 20 time steps on 512 cores. We conduct trace analysis to obtain the critical-path profile, our performance indicators, and a per-process performance profile.

The measured wall-clock execution time for the example configuration is 70.6 seconds. The major fraction of this

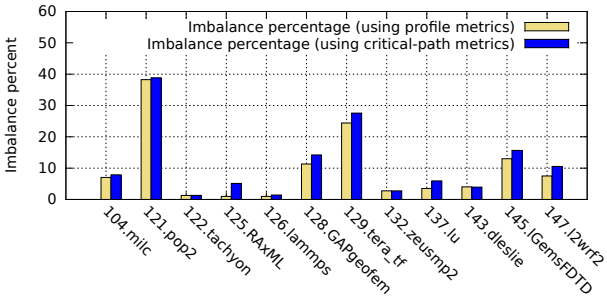


Fig. 8. Load imbalance metrics for SPECMPI.

	tree_walk	sum_force
Runtime per process		
Minimum	14.44 s	20.15 s
Average	15.07 s	20.63 s
Maximum	16.68 s	21.17 s
Imbalance time (Maximum-Average)	1.61 s	0.54 s
Critical-path time	20.78 s	21.29 s
Critical-path imbalance (Crit.time-Average)	5.71 s	0.66 s

TABLE I
PEPC PROFILE STATISTICS FOR "TREE_WALK" AND "SUM_FORCE"

time is spent in two code regions: a computational tree-balancing loop ("tree_walk") and the calculation of particle forces ("sum_force"). Table I summarizes runtime and critical-path profile data for these two code regions and the associated load imbalance metrics. The per-process profiles suggest that the "sum_force" calculation takes around 5 seconds longer than the "tree_walk" loop. However, even though no single process spends more than 16.7 seconds in the "tree_walk" loop, the critical-path profile reveals that both code regions are responsible for around 21 seconds of the total wall-clock time. The iterative nature of the program and load imbalance in the "tree_walk" loop explains this discrepancy. Since PEPC employs a dynamic load-balancing scheme, the load imbalance is not static. Instead, load maxima appear on different processes in different iterations. Due to global synchronizations within each iteration, the per-iteration load maxima accumulate on the critical path, so that the total impact of the "tree-walk" loop on program runtime is actually larger than the time it consumes on any process. As shown in the artificial example earlier, runtime profiles cannot capture this effect and underestimate the loop's runtime influence. The critical-path based metrics clearly provide more insight; in particular, the critical-path imbalance indicator shows that the load imbalance in "tree_walk" wastes 5.7 seconds of wall-clock time. These results demonstrate that the critical-path imbalance indicator estimates the influence of load imbalance on program runtime with higher accuracy than indicators based on per-process profiles alone do, making it a highly valuable tool to assess load balance and to detect parallelization bottlenecks.

D. ddcMD

To demonstrate the functionality of our critical-path analysis with an MPMD code, we perform experiments with the molecular dynamics simulation code ddcMD [7]. This code partitions simulation components in a heterogeneous decomposition according to their scaling properties to circumvent the scaling problems of long-range interaction force calculations [8]. ddcMD uses a particle-particle/particle-mesh algorithm that divides the force calculation into an explicit short-range pair-interaction piece, and a long-range piece that is solved in Fourier space. While the short-range interaction scales well, the long-range force calculation does not. In ddcMD, this problem is solved by calculating the long-range forces (mesh calculation) on a small partition of around 5-10% of the available processes, and the short-range pair interactions (particle calculation) on a partition of the remaining processes. Both tasks can run in parallel, but must be carefully load-balanced in order to achieve high efficiency. In particular, the mesh calculation should run slightly faster than the particle calculation, so that the mesh processes do not block the large group of particle processes.

Load balance between the particle and mesh tasks can be tuned in various ways. Obviously, the partition sizes themselves are crucial parameters. However, on systems with a highly regular network interconnect, such as the Blue Gene/P, the placement of processes on the system is also an important factor for achieving optimal efficiency in ddcMD, which leaves only few reasonable options for the partition sizes. Thus, a useful strategy is to first choose a fixed partitioning scheme and process mapping that optimally fits the Blue Gene's network topology, and then fine-tune the load balance between particle and mesh tasks using two other parameters that significantly impact efficiency: the inverse screening length α and the mesh size δ. By changing α and δ one can shift workload from mesh tasks to the particle tasks and vice-versa without loss of accuracy. Increasing the mesh size increases the workload of the mesh processes, and the accompanying decrease in α reduces the workload of the particle processes.

Our experiments study the influence on program performance of various combinations of α and δ (with $\alpha\delta$ kept constant). We run simulations of 38 million particles on 4,096 cores of Blue Gene/P, with fixed partition sizes of 3,840 cores for the particle tasks and 256 cores for the mesh tasks. The application itself runs for 374 seconds in the worst case. Our parallel trace analysis (including trace I/O and report collation) on the 4,096 processes runs another 285 seconds, 113 seconds of which is for the parallel replay.

Figure 9 summarizes the analysis results. The bars in the background show the duration of particle and mesh calculations on the critical path. The line at the top (shown in red) represents the overall resource consumption (i.e., critical-path length multiplied by the number of processes), the remaining lines visualize the attribution of resource consumption to the two activity classes and the inter-partition imbalance costs (this graph does not show intra-partition imbalance costs). The

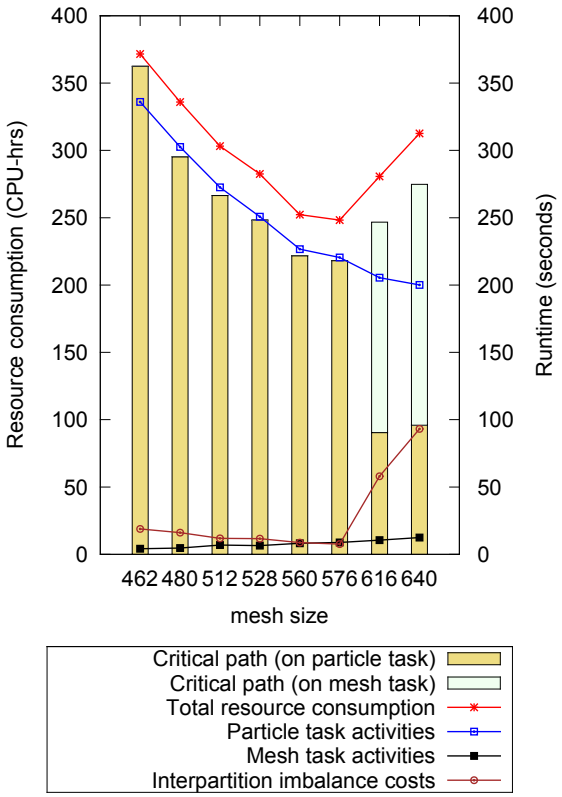


Fig. 9. Influence of mesh size δ and inverse screening length α on ddcMD performance.

spectrum of configurations ranges from small mesh sizes that place little workload on the mesh tasks and a large workload on the particle tasks on the left to large mesh sizes that shift workload from particle to mesh tasks on the right. On the left of the spectrum, the critical path visits only particle tasks while it shifts some to the mesh tasks on the right.

As expected, the workload of mesh tasks increases with larger mesh sizes, while the workload of the particle tasks decreases due to the complementary change in α. Since the overall resource demand of the particle calculations is much higher than the demand of the mesh tasks, the shift in $\alpha\delta$ combinations in favor of lower α values also leads to an overall decrease of resource consumption, and, consequently, runtime. The runtime difference between particle and mesh tasks also shrinks, further decreasing resource consumption and improving resource utilization. The inter-partition imbalance costs clearly illustrate this effect. For small mesh sizes, the inter-partition imbalance costs represent the resource loss due to the excess time on the particle tasks, which leads to wait states among the mesh tasks. The inter-partition imbalance costs and the resources employed for mesh computations exhibit a noticeable gap that indicates that the resources wasted by wait states are larger than those used by mesh computations. At a mesh size of 576, we reach an optimal configuration.

Once the mesh size exceeds that threshold, the performance characteristics change completely. The mesh calculation takes

longer than the particle calculation, so the critical path now visits the mesh tasks. While the resources consumed by the particle computations continue to decrease, the particle tasks must now wait for the mesh calculations to finish, which leads to an overall increase in runtime and total resource consumption. Considerable resources are wasted since the large number of processes in the particle partition must wait for the mesh computations, as indicated by large inter-partition imbalance costs that originate from the mesh tasks. With further increases of the mesh size, the resource waste due to the poor workload distribution between the two partitions grows rapidly.

The figure shows that the load balance between the particle and mesh partitions is key to efficient execution of ddcMD. In contrast, intra-partition imbalance costs, which Figure 9 does not include, only vary between 3 and 5% of the overall resource consumption, which indicates that the particle- and mesh-workloads themselves are well balanced.

In conclusion, the ddcMD example clearly demonstrates the usefulness of our performance indicators for characterizing load balance in MPMD programs. They provide appropriate guidance for choosing an optimization strategy. For example, the mesh sizes of 528 and 616 points lead to roughly the same overall runtime, but our performance indicators reveal radically different performance characteristics. With a 528 point mesh size, some remaining inter-partition imbalance costs of the particle computations suggests optimization by shifting workload from the particle to the mesh tasks. In the other case, the occurrence of mesh tasks on the critical path and the high inter-partition imbalance costs of the mesh computations indicate a poor workload distribution that leads to wait states among the particle processes.

V. Related work

Several researchers have explored critical-path analysis as a means to identify performance problems and to steer optimizations. For example, Barford and Crovella [9] use it to study the performance of TCP in the context of HTTP transactions, while Tullsen and Calder [10] reduce dependencies in binary codes based on critical-path information. In the context of parallel message-passing programs, the critical path is used in several prior tools to describe the actual computational demands of an application. ParaGraph [11] is an early tool that could visualize the critical path of a parallel program in a space-time diagram. Miller et al. [12] describe a distributed algorithm to capture the critical path. Schulz [13] describes techniques to extract the critical path from a message-passing program in the form of a graph, which can then be used for further post-mortem analysis. Hollingsworth [14] employs a sophisticated on-line mechanism built on top of dynamic instrumentation to collect the necessary input. This information is then aggregated per function to reduce the memory required to store the results.

However, the critical path itself is not overly expressive. We must understand the impact of changes to the critical path and execution time. To address this problem Alexander et al. [15] compute *near-critical paths* through search algorithms on execution graphs. They weigh each program edge with the computational complexity of the corresponding program section. Their algorithm requires global searches across the entire graph, which is not scalable. Further, the algorithm results in a large number of paths that are considered near-critical. Each near-critical path typically varies little from the primary critical path, which reduces its usefulness.

In general, critical-path techniques play only a minor role in current performance-analysis tools—despite the power and expressiveness of the abstraction. This minor role arises partly from the difficulty of isolating the critical path. Since it is a global *a posteriori* property, it is hard to obtain online. Also, the sheer size of the critical path makes its scalable extraction challenging. Here, our parallel post-mortem event-trace analysis is ideally suited to extract the critical path reliably and in a scalable manner. More importantly, prior work has not fully exploited the information provided by critical-path analysis. In particular, the structure of the graph is either exposed in its entirety or lost in aggregated metrics. Our work targets this gap through compact performance indicators that retain some of the dynamic information that the critical path provides.

Apart from the generation and analysis of critical paths, several tools explore the communication characteristics of parallel programs to find potential optimization targets. Two traditional approaches exist. Profiling aggregates communication behavior over time. mpiP [16] is an example of this class of tools. Many tools, including TAU [17], Open|SpeedShop [18], Vampir [19], and Paraver [20], generate trace files to be analyzed postmortem, often manually with the help of visualization. The PARADIS project [21] uses event graphs, which are similar to the execution graphs we show in this paper, to detect performance anomalies and bottlenecks. Similarly, Kranzlmüller et al. [22] use event graphs for debugging purposes.

Several performance tools analyze load imbalance. CrayPat [3] calculates imbalance metrics from profiles that provide measures of absolute and relative load imbalance. HPCTOOLKIT [23] attributes the costs of idleness at global synchronization points to overloaded call paths, highlighting imbalances in call-path profiles. However, profiling-based approaches, which aggregate performance data along the time dimension, can generally express only static imbalances reliably and do not capture dynamic load shifts between processes over time. As an alternative, both Huck et al. [24] and Gamblin et al. [25] monitor dynamic load changes by instrumenting individual iterations of the main loop in parallel programs. However, these approaches are limited to the analysis of SPMD programs; to our knowledge, none of the currently available performance tools addresses the more complicated load-balance issues in MPMD programs. As many applications are adopting a more heterogenous or MPMD model, we therefore decided to revisit the critical path as a key structure.

VI. Conclusion

Using the critical path to determine those activities that are responsible for the execution length of a parallel program is a well-known concept of characterizing application performance. However, prior work failed to strike a good balance between the unmanageable size of the full critical path on the one hand and radical aggregations that obscure important dynamic aspects on the other. In this paper, we have shown how to generate compact performance indicators that can be easily analyzed but nevertheless retain critical dynamic elements well-suited for the analysis of load imbalance.

The critical-path imbalance provides an easy-to-use tool for estimating the parallel efficiency of a message-passing code. Targeting SPMD applications, it relies on the assumption that all processes execute more or less the same mix of activities. Most suitable for MPMD codes in contrast, our imbalance cost indicators provide instruments to assess the overall resource consumption as a result of load imbalance, and to decide whether to attribute it to load imbalance within or across partitions. By applying our novel indicators to real-world simulation codes, we have demonstrated their usefulness (i) in identifying the load imbalance with the most severe effects on application performance in an SPMD code and (ii) in finding the most efficient configuration of an MPMD molecular dynamics application.

As part of our future work, we will exploit knowledge of the dynamic structure of the critical path to improve our analyses further. Moreover, due to the increasing number of cores per CPU and the growing trend to employ OpenMP for node-internal work sharing while using MPI for inter-node communication, we will extend our analysis approach to support hybrid OpenMP/MPI applications.

Acknowledgements

Financial support from the German Research Foundation through Grant GSC 111, the German Federal Ministry of Education and Research through Grant 01-IH-11006, and from the Helmholtz Association of German Research Centers through Grant VH-NG-118 is gratefully acknowledged. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-522431). The authors also would like to thank Dave Richards from Lawrence Livermore National Laboratory for his invaluable help in setting up the ddcMD code.

References

[1] J. E. Kelley, Jr and M. R. Walker, "Critical-Path Planning and Scheduling," in *Proc. of the Eastern Joint IRE-AIEE-ACM Computer Conference, Boston, Massachusetts*. ACM, December 1959, pp. 160–173.

[2] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications," *Parallel Computing*, vol. 35, no. 7, pp. 375–388, Jul. 2009.

[3] L. DeRose, B. Homer, and D. Johnson, "Detecting Application Load Imbalance on High End Massively Parallel Systems," in *Euro-Par 2007 Parallel Processing*, ser. LNCS, vol. 4641. Springer, 2007, pp. 150–159.

[4] Standard Performance Evaluation Corporation. (2007) SPEC MPI2007 benchmark suite. [Online]. Available: http://www.spec.org/mpi2007/

[5] J. Mußler, D. Lorenz, and F. Wolf, "Reducing the overhead of direct application instrumentation using prior static analysis," in *Proceedings of Europar 2011, Bordeaux, France*, ser. LNCS, vol. 6852. Springer, Sep. 2011, pp. 65–76.

[6] S. Pfalzner and P. Gibbon, *Many-Body Tree Methods in Physics*. New York: Cambridge University Press, Sep. 2005.

[7] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz, "Extending Stability Beyond CPU Millennium: A Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability," in *Proceedings of IEEE/ACM Supercomputing '07*, Nov. 2007.

[8] D. F. Richards, J. N. Glosli, B. Chan, M. R. Dorr, E. W. Draeger, J.-L. Fattebert, W. D. Krauss, T. Spelce, F. H. Streitz, M. P. Surh, and J. A. Gunnels, "Beyond Homogeneous Decomposition: Scaling Long-Range Forces on Massively Parallel Systems," in *Proceedings of IEEE/ACM Supercomputing '09*, Nov. 2009.

[9] P. Barford and M. Crovella, "Critical Path Analysis of TCP Transactions," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 2, pp. 80–102, 2001.

[10] D. Tullsen and B. Calder, "Computing Along the Critical Path," UC San Diego, Tech. Rep., 1998.

[11] M. T. Heath, A. D. Malony, and D. T. Rover, "The Visual Display of Parallel Performance Data," *IEEE Computer*, vol. 28, no. 11, pp. 21–28, November 1995.

[12] B. P. Miller, M. Clark, J. K. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 206–217, Apr. 1990.

[13] M. Schulz, "Extracting Critical Path Graphs from MPI Applications," in *Proceedings of the 7th IEEE International Conference on Cluster Computing*, September 2005.

[14] J. K. Hollingsworth, "An Online Computation of Critical Path Profiling," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '96)*, 1996.

[15] C. A. Alexander, D. S. Reese, J. C. Harden, and R. B. Brightwell, "Near-Critical Path Analysis: A Tool for Parallel Program Optimization," in *Proceedings of the First Southern Symposium on Computing*, 1998.

[16] "mpiP," http://mpip.sourceforge.net/.

[17] R. Bell, A. Malony, and S. Shende, "ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis," in *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, Aug. 2003, pp. 17–26.

[18] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008.

[19] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[20] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A Tool to Visualise and Analyze Parallel Code," in *Proceedings of WoTUG-18: Transputer and Occam Developments*, vol. 44. Amsterdam: IOS Press, 1995, pp. 17–31.

[21] C. Glasner, E. Spiegl, and J. Volkert, "PARADIS: Analysis of Transaction-Based Applications in Distributed Environments," in *Proceedings of the 5th International Conference in Computational Science (ICCS), Part II, LNCS vol. 3515*, May 2005, pp. 124–131.

[22] D. Kranzlmüller, M. Löberbauer, M. Maurer, C. Schaubschläger, and J. Volkert, "Automatic Testing of Nondeterministic Parallel Programs," in *Proceedings of PDPTA '02*, 2002, pp. 538–544.

[23] N. R. Tallent, L. Adhianto, and J. Mellor-Crummey, "Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles," in *Supercomputing 2010*, New Orleans, LA, USA, Nov. 2010.

[24] K. A. Huck and J. Labarta, "Detailed Load Balance Analysis of Large Scale Parallel Applications," in *Proceedings of the 39th International Conference on Parallel Processing*, Sep. 2010, pp. 535–544.

[25] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed, "Scalable Load-Balance Measurement for SPMD Codes," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.