Extracting Logical Structure and Identifying Stragglers in Parallel Execution Traces

Katherine E. Isaacs* Todd Gamblin† Abhinav Bhatele† Peer-Timo Bremer†
Martin Schulz† Bernd Hamann*

*Department of Computer Science, University of California, Davis
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
keisaacs@ucdavis.edu, tgamblin,bhatele,ptbremer,schulzm@llnl.gov, hamann@cs.ucdavis.edu

Abstract

We introduce a new approach to automatically extract an idealized *logical structure* from a parallel execution trace. We use this structure to define intuitive metrics such as the *lateness* of a process involved in a parallel execution. By analyzing and illustrating traces in terms of logical steps, we leverage a developer's understanding of the happened-before relations in a parallel program. This technique can uncover dependency chains, elucidate communication patterns, and highlight sources and propagation of delays, all of which may be obscured in a traditional trace visualization.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging – Tracing

Keywords parallel execution trace; logical structure; visualization

1. Introduction

Analyzing the trace of a parallel execution is a powerful tool in identifying performance issues. Before specific problems have been identified, trace analysis typically starts with a visualization to obtain an overview and to form and test hypotheses about causes of observed slowdowns. Most visualizations align trace events by wall-clock time. Such a view rarely preserves the logical structure of the parallel code. Simple questions such as which messages arrived late are difficult to answer without the ability to identify logical phases or steps in a parallel program. As a result, a timeline view can be confusing even for small numbers of processes.

In contrast to existing approaches, we extract the underlying logical structure to show developers parallel traces in the context of their program's communication patterns. We then map timing information onto this structure. In this work, we focus on MPI, a widely used message passing standard in high performance computing. We obtain our traces using VampirTrace [5] and store them in the Open Trace Format (OTF) [2]. Our approach is, however, generally applicable to any programming model based on message passing.

PPoPP '14, Feb 15–19, 2014, Orlando, FL, USA.

ACM 978-1-4503-2656-8/14/02.
http://dx.doi.org/10.1145/2555243.2555288

2. Extracting Logical Structure

The logical structure of a program is the ordering of events implied by that program. Ideally this structure reflects the developers' intended organization but may instead reflect the unintended result of the program. We describe the logical structure by assigning a *logical step* to each event. Events intended to happen simultaneously have the same step.

Our algorithm applies Lamport's *happened-before* relation [3] to derive structure from send and receive operations. Other operations are grouped into aggregate events occurring between the message events. We focus on sends and receives because they impose happened-before relations between processes. We first partition the send and receive events into related communication groups. Then we assign logical steps in each partition. The partitions themselves are partially ordered, imparting global steps on the events.

Partitions represent non-overlapping application phases. The partitioning may be pre-defined, but if not, we derive it from the trace. We first group events related by semantic or ordering constraints. Initially, each event is a partition. We build a graph over the partitions using happened-before relationships as edges. We then merge matching sends and receives as these must be related. Similarly, we merge partitions with events handled by the same `MPI_Waitall` call.

A merged partition maintains relations between its events and those in other partitions. This may introduce cycles into the partition graph, indicating that no absolute linear ordering exists. Therefore, partitions that form a strongly connected component must be related and are merged. Optionally, we attempt to merge partitions that logically belong together despite not having ordering constraints. In bulk synchronous codes, we expect each process to be active at any distance in the partition graph and merge accordingly.

We assign local steps in each partition following two principles. First, happened-before relationships must be maintained: events cannot change order and receives must oc-

cur at least one step after their sends. Second, send events have a greater impact on structure. This is because the order messages are received is not always uniquely defined by the program and some events may contain multiple receives. We first determine simultaneous send groups by applying happened-before relationships while assuming receives are instantaneous. Then each event is assigned the least step possible based on these constraints. Global steps are offset by the maximum step in the preceding partitions. Finally, we insert aggregate events representing work done by a process between each pair of communication events.

3. Mapping Temporal Information

Traditional trace visualizations use the horizontal axis for recorded time. Visualizing the logical structure instead uses the horizontal axis to represent order. We restore the temporal information by coloring events with time-based metrics that take advantage of the logical structure.

We calculate how late an event was relative to its peers. We define the *lateness* of an event as the excess completion time over the earliest related event sharing the step. Figure 1 shows a portion of a 16-process MG [1] trace as visualized in Vampir [4] versus using logical steps and lateness. The latter shows a late aggregate event in the first process propagate lateness to other processes via message dependencies.

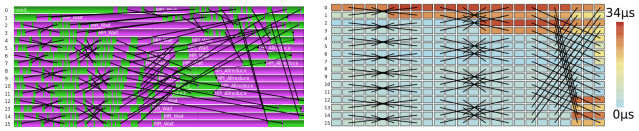

Figure 1: Vampir (left) and lateness-colored logical structure (right) visualization of a 16-process execution of MG.

We classify four situations contributing to event lateness (Figure 2). Two create lateness, through the event itself (2a) or a message for which the event is waiting (2d). The other two propagate lateness. Either the process is late in leaving the preceding event (2b) or a send is already late, causing the event to wait (2c). Figure 1 exhibits cases (a), (b), and (c).

(a) Created in event. (b) Process propagation. (c) Message propagation. (d) Created in message.

Figure 2: Initiation and propagation of lateness

We use this classification to narrow our focus to just the events where lateness originates, subtracting propagated lateness in each event to calculate *differential lateness*. By searching for events with high differential lateness, we can pinpoint areas of interest in the trace automatically.

4. Case Study

We analyze a massively parallel algorithm to compute *merge trees*. The algorithm relies on a global gather-scatter approach. Each process is assigned an equal portion of the data. All processes perform a local computation and send the results to their respective gather processes. These integrate the information and send the results both upwards to the next level merge and back downwards to the leaves. This repeats up the tree ending with the root performing the final gather.

Figure 3: Vampir (top) and lateness-colored logical structure (bottom) visualization of a 16-process, 4-ary merge tree.

Figure 3 shows two visualizations of a complete 16-process, 4-ary merge tree. The logical step view exposes features not obvious in the traditional visualization. The late events reflect data-dependent load imbalance. The logical steps highlight the gather tree structure, revealing that the gather processes send back to the leaves before sending up to the root. This misses an opportunity for a more aggressive pipelining of the computation as no process can finish until the root has been reached and thus the gather should be prioritized over the scatter.

Acknowledgments

This research is supported by the U.S. Department of Energy Office of Science Graduate Fellowship administered by ORISE-ORAU under contract DE-AC05-06OR23100 and by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-ABS-647655).

References

[1] D. H. Bailey et al. The nas parallel benchmarks. *Int. J. Supercomput. Appl.*, 5(3):63–73, 1991.

[2] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the open trace format (OTF). In *Proc. of 6th Int. Conf. on Comp. Sci.*, ICCS'06, pages 526–533. Springer-Verlag, 2006.

[3] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[4] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[5] TU Dresden Center for Information Services and High Performance Computing (ZIH). VampirTrace 5.14.2 user manual. `http://www.tu-dresden.de/zih/vampirtrace`, March 2013.