Representative Paths Analysis

Nathan R. Tallent
Pacific Northwest National Lab
tallent@pnnl.gov

Darren J. Kerbyson
Pacific Northwest National Lab
Darren.Kerbyson@pnnl.gov

Adolfy Hoisie
Pacific Northwest National Lab
Adolfy.Hoisie@pnnl.gov

ABSTRACT

Representative paths analysis generalizes and improves MPI critical path analysis. To improve diagnostic insight, we sample the distribution of program path costs and retain k representative paths. We describe scalable algorithms to collect representative paths and path profiles. To collect full paths efficiently, we introduce path pruning that reduces permanent space requirements from a trace (proportional to ranks and MPI events) to path length (the minimum). To make space requirements independent of ranks and events — even a small constant in practice — we profile program paths. Avoiding the limitations of prior path profiling approaches, we dynamically discover tasks and attribute costs in high resolution. We evaluate our algorithms on seven applications scaled up to 7000 MPI ranks. Full program paths use as little as 0.01% the permanent space of current methods; profiles require a nearly constant 100–1000 KB. Execution overhead is under 5% when synchronization intervals are sufficiently large (a few milliseconds).

ACM Reference Format:
Nathan R. Tallent, Darren J. Kerbyson, and Adolfy Hoisie. 2017. Representative Paths Analysis. In *Proceedings of SC17*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3126908.3126962

1 INTRODUCTION

Critical path analysis (e.g., [2, 6, 9, 27, 38]) is a well known technique for identifying bottlenecks to application scalability and predicting performance. Compared to standard profiles or traces, paths retain program *dependences*. However, MPI critical path algorithms either require substantial space or attribute path costs in low resolution.

Collecting full MPI critical paths requires substantial space. Although the best algorithms prune many paths, they still retain one candidate path per rank [27, 38]. Suppose a task is the computation between two MPI operations. Then, collecting full paths requires space proportional to ranks and tasks. If a program executes on P process ranks and has T task instances per rank, the permanent space requirement is $O(TP)$, equivalent to a trace. This results in large output files — gigabytes at only 1000 ranks — that challenge both I/O systems and post-mortem analysis. This space requirement also makes exploring multiple paths prohibitive: collecting k paths uses $O(kTP)$ space.

SC17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126962

Critical path profiling [16] reduces the space requirement by sacrificing task sequencing. During an execution, many task instances are repeated. Let $\dot{T}$ represent the number of *distinct* tasks — e.g., program functions — in a program path. A critical path profile requires $O(\dot{T})$ space, which is typically scalable with respect to both ranks and execution time. Unfortunately, current approaches [16] have two distinct drawbacks. They do not perform dynamic task discovery but require as input a set of static tasks such as function names. Moreover, they attribute critical path costs only to tasks, much lower resolution than per-rank profiles (program statements).

This paper introduces representative paths analysis (§3), a generalization of MPI critical path analysis (§2). We describe a family of algorithms for collecting representative program paths as either full paths or profiles. We make the following contributions.

First, to improve diagnostic insight, we sample the entire distribution of program path costs and retain k representative paths (§4). Whereas prior work focuses on one path (most or least critical) or the top k critical paths [2], representative paths shows the *distribution*. One can estimate program efficiency by comparing each path to the longest. For SPMD applications, when load imbalance causes waiting time, a standard profile may show time in an MPI wait loop; but comparing paths shows the exact task that receives disproportionate work. With paths, it is also possible to estimate scaling potential or the effects of a task optimization by modeling changes to the paths' tasks and comparing the new paths.

Second, we introduce pruning (§5) that reduces the permanent space requirement for collecting full paths from $O(TP)$ to the path length, or $O(\tilde{T})$, the minimum. Here, $\tilde{T}$ represents task instances along the critical path. When $\tilde{T}$ and T are similar, space scales with ranks and the savings is large. Collecting k representative paths requires $O(k\tilde{T})$ permanent space, not $O(kTP)$. Moreover, the space savings reduces the cost of post-mortem path analysis simply by reading far less unnecessary data from permanent storage.

Third, we show how to collect representative path profiles (§6). A profile's space requirements are typically independent of both ranks and execution time. Compared to collecting full paths, profiles reduce the permanent space requirements to a small constant in practice: collecting k representative paths requires only $O(k\dot{T})$ space. Improving current techniques [16], we show how to dynamically discover program tasks and attribute costs in high resolution, viz. program statements in full calling context. At the same time, we bound the key temporary (online) space requirements to a constant. Our solution is based on (a) adaptive sampling to dynamically discover tasks in bounded space; and (b) dynamic path summarization to maintain enough attribution information about candidate paths until the representatives are selected.

We evaluate (§7) our work using seven MPI applications executing on up to 7000 ranks. Our applications cover numerical solvers, graphs, and machine learning; encompass several distinct communication/synchronization paradigms; and range from proxy

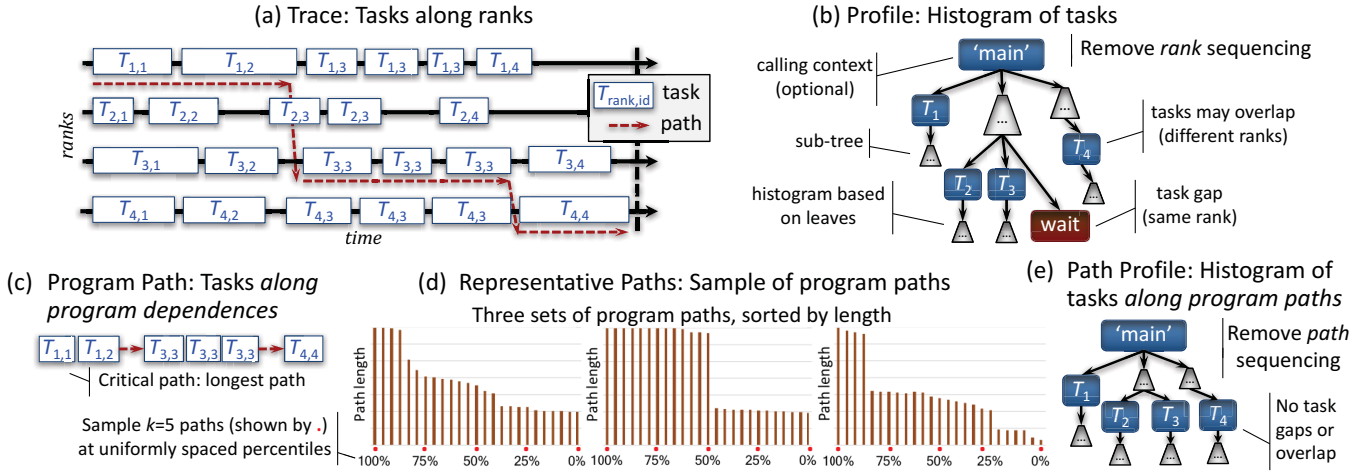

Figure 1: Comparing (a) traces, (b) profiles, and (c) critical paths with (d) representative paths and (e) path profiles.

to production applications. Execution overhead is low (under 5%) when synchronization intervals are sufficiently large (a few milliseconds). We show substantial reductions in storage requirements: paths use as little as 0.01% the permanent space of current methods; and profiles require a nearly constant 100-1000 KB, independent of ranks and execution time. Although most of these applications have a bi-modal distribution of path lengths, we observe exceptions, highlighting the importance of the representative sample.

Finally, we cover related work (§8) and provide conclusions (§9).

2 CRITICAL PATH ANALYSIS

Critical path analysis views an application's execution as a directed acyclic *program activity graph* (PAG) [38] where vertices represent application events and edges represent dependences between events. Our interest is in MPI synchronization events. We define the computation between two adjacent vertices on the same MPI rank as a task. Within a task, program-order dependences are implicit. Figure 1(a) shows a critical path within an execution trace.

There are two basic approaches for online critical path analysis. The *longest path method*, which we adopt, defines the critical path as the longest path and requires one forward pass through the PAG [38]. The *slack method* prunes paths that have slack [27]. The basic longest path algorithm is as follows. At each branch (multiple outgoing edges), send path information along each edge. At each join (multiple incoming edges), retain the longest path seen so far by the rank. Thus, joins effectively prune many paths by retaining only one path (the longest) per rank.

The space complexity of this algorithm is as follows. Let a program execute on P process ranks and have at most T task instances per rank. Retaining one path per rank requires $O(TP)$ permanent space, equivalent to a trace of MPI operations. This results in large output files (e.g., gigabytes at only 1000 ranks) that challenge post-mortem analysis and can challenge I/O devices or contribute to I/O contention. Collecting more than one path scales the space requirements linearly: to collect k paths requires $O(kTP)$ space. Slack-based algorithms have the same requirements.

Critical path profiling [16] can reduce the space requirement substantially. Observe that during an execution, many task instances are repeated. Let $\dot{T}$ represent the number of distinct tasks along a program path. A path profile requires only $O(\dot{T})$ space. In most cases, $\dot{T}$ is constant with respect to ranks; and is constant or grows very slowly with execution time. Unfortunately, current path profiling methods do not perform dynamic task discovery but require as input a static set of task names (e.g., functions). Moreover, these methods attribute critical path costs only to tasks, which is much lower resolution than standard profiles (program statements) [1, 18].

3 REPRESENTATIVE PATHS ANALYSIS

Figure 1 compares representative paths with traces, profiles, and critical paths. Compared to a standard trace or profile (Fig. 1(a–b)), path-based analysis (Fig. 1(c)) retains information on task *dependences*. Whereas traces attribute costs to tasks along a rank, path-based analysis attributes costs to tasks *along program paths*. As a result, neither traces nor profiles account for task overlap or gaps between tasks. That is, analyzing tasks along any rank ignores blocking time; and analyzing tasks across ranks ignores overlap.

Generalizing critical path analysis, representative paths analysis collects k paths that sample the distribution of program path lengths or costs (Fig. 1(d)). Sampling the 100^{th} and 50^{th} percentiles gives the critical and median paths, respectively. We typically sample at evenly spaced percentiles. Thus, sampling $k = 5$ paths collects paths at percentiles 100%, 75%, 50%, 25%, and 0%. Path profiling (Fig. 1(e)) makes this analysis extremely scalable.

Representative paths provides more diagnostic information than the common alternatives. Most and least critical paths shows only spread of paths; representative paths shows both spread and distribution. Near critical paths [2] identify the top k paths. The top k paths are often nearly indistinguishable in terms of cost and composition. Because the number of program paths doubles with each branch, the top k paths usually represent the 100^{th} percentile. Furthermore, MPI applications often have many phases, separated by all-to-all collectives. Over Q phases, the top k program paths

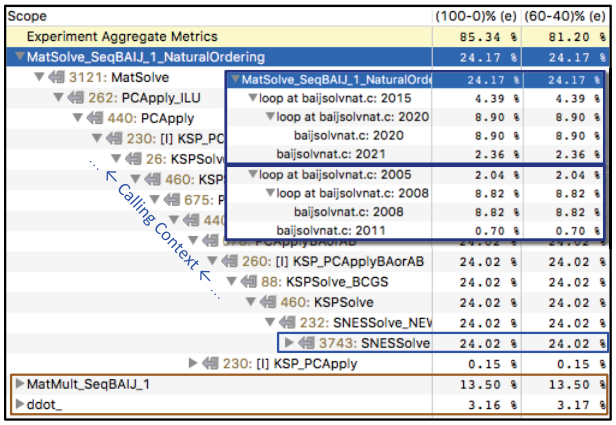

Figure 2: Representative paths diagnosing load imbalance.

often share task instances from the first $Q-1$ phases. In contrast, representative paths samples the range of percentiles.

Representative paths provide the following diagnostic information. First, estimate efficiency (the ratio of computation to total execution time) by subtracting each path from the longest. Second, estimate speedup potential by modeling the effects of a task optimization in each path. Third, estimate scaling potential by examining the concurrency in tasks along each path. Fourth, for SPMD programs where task structure is similar on all ranks, pinpoint and quantify load imbalance by comparing paths' task structure. Whereas a standard profile can show time in an MPI progress engine wait loop — *effects* of load imbalance — comparing paths pinpoints *task instance and statements responsible* for load imbalance. All of these diagnostics require multiple program paths and are therefore unavailable with standard profiles or traces.

Figure 2 shows an example of diagnosing load imbalance with representative paths analysis. We profiled 8 representative paths of PFLOTRAN [14], an SPMD subsurface flow and reactive transport simulator. As a profile, all instances of the same computation on a path are aggregated. (Total profile size was 1000 KB.) The figure's leftmost column shows a variant of the `MatSolve` task (blue highlight) with its calling context underneath. The call to `MatSolve` comes from a call to PETSc's [4] non-linear solver `SNESSolve`.

The figure's right two columns show two metrics computed from the representative paths. The left metric differences paths 100% and 0% (most and least critical); the right differences paths 60% and 40%. The differences are presented as a *percentage of the critical path*. The top values (yellow highlight) refer to the entire path; below is specific task attribution. Thus, the top-left value (85%) means the 0% path spent 85% of the critical path in wasted time. Similarly, on the 0% path the `MatSolve` task (blue highlight) spent 24% of the critical path in wasted time. The inset shows detailed loop and line-level attribution within the `MatSolve` implementation.

Observe that the breakdown of the two difference metrics is nearly identical. This implies a bi-modal distribution of paths, where paths 100%. . .60% and paths 40%. . .0% have nearly identical costs and tasks. Therefore, the top-right 81% (yellow highlight) implies that *the bottom 40% of the program paths waste at least 81% of the execution time*. (Execution time is determined by the critical path.) The inefficiency is due to domain-induced load imbalance. Because of the nature of the physical domain, some 'cells' are inactive. There are enough inactive cells that, at this scale, 40% of program paths have no work for the solver. (Correcting the load imbalance is difficult because the simulator is designed to be configurable.)

This diagnostic insight would be impossible with standard traces, profiles, and even knowledge of the most and least critical paths. However, one profile of 8 representative paths quantified the severity of load imbalance and pinpointed it *to statements within a task*.

4 SELECTING PATHS

The problem of selecting representative paths is similar to reservoir sampling [35]. That is, we desire a sample of size k from N program paths where we initially know neither N nor the path distribution. A difference is that the path tails (giving total path cost) are distributed over P program ranks. Another difference is that we are interested in representatives at uniformly spaced percentiles, or k order statistics [22, §14.4] for N distributed paths. Our solution, building on the longest path method (§2), is to incrementally maintain a sample set of k path order statistics per rank. The key is that the set *represents all paths encountered by the rank so far*. At each program branch, the set is sent along each edge. Each two-way join retains, from the $2k$ paths per incoming edge, k paths at the selected percentiles. At all-to-all synchronization, a path allreduce operation retains k representatives of the kP candidates over all ranks. The algorithm's correctness follows from path selection at joins. That is, if each incoming edge has k representative paths, the join selects k paths representative of all incoming edges, and by induction, all paths encountered by the rank so far.

4.1 Algorithm Overview

Figure 3 shows our algorithm template for collecting k representative MPI program paths as either full paths or as a profile. The algorithm has two template parameters. The *space* parameter selects between two types of permanent storage, either full paths (`path`) or profiles (`prof`). Figure 4 shows specialized definitions. The *stat* parameter selects between different kinds of path statistics. Figure 5 shows two possible definitions. The default (`repr`) collects paths at uniformly spaced percentiles based on k.

Each rank maintains a Path-set of k paths. The definition of the Path-set structure depends on whether the algorithm is collecting full paths or profiles (Fig. 4(a) and (b)). In both cases, a path structure has an accumulated cost and summary. When collecting full paths, each path is a distributed list of tasks; hence, each task has an id and tail link (Fig. 4(a)).

The algorithm in Figure 3 follows the two rules described above. First, paths branch at sends; therefore a send, whether blocking (line 1) or non-blocking (line 5), updates its rank's Path-set and then sends the resulting set (lines 4, 8). (Code for path branches is shown in orange.) Second, when paths join at recvs, k representative paths are retained from the two incoming Path-sets. (Code for path joins is shown in blue.) With blocking recv (line 9), path selection occurs immediately (line 13). With non-blocking recv (line 14), path selection must wait to ensure paths have arrived (line 24). The resulting Path-set represents all paths encountered by the receiving rank so far.

```
// Template parameter space: path (k full paths) or prof (profile of k paths)
// Template parameter stat: repr (representative paths) or topk (most/least paths)
1  rp_send(payload, y)                                          // rank x → y
2    rp_makeTask⟨space⟩(my path-set P)
3    send(payload, y)
4    bsend(P, y, comm′)
5  rp_isend(payload, y, request)                                // rank x → y
6    rp_makeTask⟨space⟩(my path-set P)
7    isend(payload, y, request)
8    bsend(P, y, comm′)
9  rp_recv(payload, x)                                          // rank x|* → y
10   rp_makeTask⟨space⟩(my path-set P)
11   recv(payload, x)                                           // resolves x
12   recv(path-set P_x, x, comm′)
13   rp_select⟨stat⟩(P, copy(P), P_x)
14 rp_irecv(payload, x, request)                                // rank x|* → y
15   irecv(payload, x, request)
16   if x ≠ * then irecv(path-set P_x, x, request′)
17 rp_wait(request)                                             // rank x → y
18   rp_makeTask⟨space⟩(my path-set P)
19   wait(request)
20   info ← find_arg_info(request)
21   if info.recv then                                          // x resolved by line 19
22     if info.x ≠ * then wait(info.request′)
23     else recv(path-set P_x, x, comm′)
24     rp_select⟨stat⟩(P, copy(P), P_x)
25 rp_barrier()
26   rp_makeTask⟨space⟩(my path-set P)
27   power2 ← 0x1
28   while power2 < P do                                        // assume P is a power of 2
29     dst, src ← (me + power2)%P, (me +P− power2)%P
30     sendrecv(P, dst, path-set P_src, src)
31     rp_select⟨stat⟩(P, copy(P), P_src)
32     power2 ← power2 ≪ 1
33   if stat=repr then bcast(P, root rank)                      // when not associative
34   foreach path p in P do save my rank's samples/tasks (path-summary)
35   reset phase state: paths, path/sample-buffer, sampling period
```

Figure 3: Algorithm template for collecting k representative program paths as either full paths or a profile.

```
// Path-set⟨space=path⟩: k paths; each is ⟨cost, path-summary, tail-task⟩
// Path-buffer: uncommitted tasks; each is ⟨cct-id, path-rest-id, cost⟩
1 rp_makeTask⟨space=path⟩(P)                                    // Path-set P
2   cct-id ← id of backtrace in calling context tree
3   foreach path p in P do
4     task ← new task with cct-id, link to previous task, and cost
5     i ← onto path-buffer, push task at index i
6     rp_updatePathSum(p.path-summary, i)
```

```
// Path-set⟨space=prof⟩: k paths; each is ⟨cost, path-summary⟩
// Sample-buffer: uncommitted task samples; each is ⟨cct-id, time⟩
1 rp_makeTask⟨space=prof⟩(P)                                    // Path-set P
2   foreach path p in P do update cost                          // for validation/scaling
3 rp_sample⟨space=prof⟩(t)                                      // sample interval t
4   cct-id ← id of backtrace in calling context tree
5   i ← onto sample-buffer, push cct-id and t at index i
6   foreach path p in P do rp_updatePathSum(p.path-summary,
      i)
7   if sample-buffer size > next threshold then  double sampling
      period
```

```
// Path-summary: array of path intervals where interval is rank:[low, high]
1 rp_updatePathSum(s, i)          // Path-summary s, index i; my rank r
2   if can merge r:[i, i] with s's last interval then merge
3   else if available space in s then create new interval r:[i, i]
4   else if space=path then set overflow flag
5   else if space=prof then randomly select interval to merge with
      'unknown-location'; create new interval r:[i, i]
```

Figure 4: The *space* parameter: (a) `path`, (b) `prof`; (c) common.

Path selection is performed by `rp_select⟨stat⟩` (Fig. 5). There are several possible definitions. The default, `rp_select⟨stat=repr⟩`, selects paths at uniformly spaced percentiles based on k, where the first and last percentiles are 100% and 0% (most and least critical). We copy the two incoming Path-sets ($\mathcal{Q}$ and $\mathcal{R}$) to a temporary array, sort it, and then select paths based on pre-computed indices. In contrast, `rp_select⟨stat=topk⟩` selects the top $k/2$ most and least critical paths. It is also possible to simply collect a uniform

```
1 rp_select⟨stat=repr⟩(P, Q, R)                                 // Path-sets P, Q, R
2   P ← { k representative paths from Q ∪ R }
```

```
1 rp_select⟨stat=topk⟩(P, Q, R)                                 // Path-sets P, Q, R
2   P ← { max k/2 and min k/2 paths from Q ∪ R }
```

Figure 5: The *stat* parameter: (a) `repr` or (b) `topk`.

random sample of paths. This could avoid the sort required by order statistics. However k is usually small so sort is not a bottleneck.

When paths join at all-to-all synchronization, a path allreduce operation ensures that each rank completes with an identical Path-set. The algorithm is shown in `rp_barrier` (Fig. 3, line 25). The core of the allreduce (lines 27–32) is based on the dissemination barrier [15]. For convenience, the algorithm assumes the number of ranks P is power of 2. Unlike a typical dissemination allreduce, an additional step may be required because path selection (line 31) based on sampling is *not* associative. When path selection is non-associative, after $\log P$ iterations, each rank has a valid but *not* identical set of representative paths. The additional step broadcasts one Path-set (line 33). After this step, each rank has an identical Path-set that represents all paths in the phase. The code can be adapted to handle other all-to-all collectives.

4.2 Implementation Details

Our implementation supports MPI-1 [21], which covers most applications. Basic support for MPI-2 (one-sided) is easy because one-sided operations do not synchronize. Supporting MPI-3 (non-blocking collectives) is straightforward, but requires rewriting collectives as point-to-point operations; the shortcut described below for single-root collectives is invalid.

The algorithm preserves the invariant that all non-blocking calls remain non-blocking and thus cannot deadlock. For performance, the recv algorithms pre-post recvs for the incoming Path-set whenever the source is not a wildcard. To support wildcards (designated by '*'), recvs wait until the source is resolved before receiving the incoming Path-set (Fig. 3, lines 11, 16, 22.)

There are two basic options for sending Path-sets: (a) send sets with the message payload (piggybacking), either using padding or MPI datatypes; or (b) send sets in a separate message. In practice, a two-message scheme typically has better performance (cf. [28]). The reason is that piggybacking typically introduces a memory copy for the payload. The two-message scheme preserves zero-copy message transfer. Another advantage of the two message scheme is that it can easily be implemented using the PMPI override interface. A disadvantage is that one must use an extra communicator or unused tag when sending path sets because we cannot guarantee the order of posted non-blocking recvs. We adopt the two-message scheme (Fig. 3, lines 4 and 8).

Grouped Point-to-point Operations. MPI provides convenience operations such as sendrecv and waitall that group several primitive operations. To capture paths according to program semantics, we re-implement grouped operations.

Single-root Collectives. Unlike all-to-all collectives that synchronize with all ranks, single-root collectives (e.g., `MPI_Bcast`) synchronize only with a root rank. One option is to re-implement with point-to-point operations. Another option with blocking semantics is to use our all-to-all algorithm. Because performance is frequently unchanged, we adopt the second approach.

Incremental Waiting and Busy Waiting. Implementing waitany is a straightforward extension to `rp_wait`. The implementation for test is also similar to `rp_wait`.

Rank Groups. To support paths across multiple communicators, we record rank labels relative to the 'world' communicator.

5 PRUNING PATHS

Recall that a program executes on P process ranks and has at most T task instances per rank. Let $\tilde{T}$ be the task instances along the critical path. We introduce new pruning that reduces the permanent space required to gather k paths from $O(kTP)$ to $O(k\tilde{T})$, the minimum. When program paths are not proportional to ranks — the typical case for MPI applications — we have $\tilde{T} = O(T)$ and space is $O(kT)$, a substantial reduction over $O(kTP)$. In addition, the space savings can greatly reduce the cost of post-mortem path analysis by reading far less data from permanent storage.

To reduce permanent space requirements to the minimum, we use some temporary storage. However, we bound this temporary storage to $O(kT'P)$, where T' is the maximum number of tasks on one rank for any *phase*. Because most executions have many phases, the pruning results in drastic reductions in permanent storage for only a modest amount of additional memory.

The temporary storage is needed because a phase's set of representative paths cannot be known until the phase-end path allreduce. Recall that the algorithm maintains k representative paths per rank, resulting in kP candidate paths at a phase end. Because any of the candidates could be selected, each rank must maintain its k paths. Recall that a path is a distributed list of tasks. Storing a phase's tasks requires $O(kT'P)$ space. The standard critical path algorithm stores these tasks permanently — an eager commit — wasting space. We store them in memory and call them *uncommitted*. After the path allreduce, we permanently store only tasks from the selected paths (Fig. 3, line 34) — a lazy commit — and reclaim the space.

One way to implement pruning is to walk the task back-links of selected paths, requiring additional communication. To avoid this, we developed path summarization so that each rank *locally* knows exactly which tasks to retain. We use two structures, the Path-buffer and Path-summary. The Path-buffer (Fig. 4(a)) is a buffer of uncommitted tasks on a *rank*. The Path-summary (Fig. 4(c)) summarizes uncommitted tasks on a *path*.

There are several requirements for the Path-summary structure. First, it should be small, fixed-size, and contiguous so that it can be sent at each program branch (Fig. 3, lines 4 and 8). Second, it should summarize the *entire* distributed path so that *all* tasks can be retained. Third, it should summarize in enough *resolution* so that each rank can retain *only* tasks on a path. Finally, there should be a rapid mechanism for adding a new task.

We implement the Path-summary (Fig. 4(c)) as an array of path intervals. When collecting full paths (*space*=`path`), each interval r:[*low*, *high*] represents an inclusive sequence of path tasks on rank r. The interval bounds *low* and *high* are indices in the Path-buffer. As shown in `rp_updatePathSum`, we push new tasks on the end, merging the task with the last interval when possible. This simple compression scheme captures the common case. Because each interval bound is only 2 bytes (with 2-byte buffer indices), each interval is only 8 bytes. Thus, 128 intervals requires 1 KB of space. As discussed below (§7.5), most paths can be captured in full resolution using fewer than 32 intervals.

This approach works well in practice. At each branch, the algorithm adds a task to the path, updating its Path-summary. This path update is fast ($O(1)$). Furthermore, the rank's Path-set is small and can be sent using MPI's eager protocol (cf. Fig. 3, lines 4 and 8). Using the eager protocol is important because the biggest potential performance problem is not extra network time but sending a Path-set using MPI's rendezvous protocol when the payload uses eager. A Path-set for 8 paths, each with a 128-interval Path-summary (a total of 8 KB), can be sent without adjusting the default eager threshold on common MPI implementations.

The final reason the approach works well is that there are solutions to buffer overflows. If the Path-summary overflows there are two solutions. First, depending on the application and interconnect, it may be possible to increase the Path-summary size without affecting application behavior. As a final resort, we set a flag that implies that all tasks subsequent to the last interval should be committed. Although, pruning is partial for this phase, all k paths are collected.

If the Path-buffer fills up, which is very rare, there are two alternatives. First, dynamically add buffers until the (2-byte) index space fills. Second, disable pruning and eagerly commit all tasks.

6 PROFILING PATHS

A path profile is a histogram of the *distinct* tasks along a program path (Fig. 1(e)). It associates tasks with their cumulative costs, ignoring task sequencing. Compared to collecting full paths, profiling k representative paths reduces permanent space requirements from $O(k\tilde{T})$ to $O(k\dot{T})$, where $\dot{T}$ represents the number of distinct tasks. Whereas $\tilde{T}$ is proportional to execution time and possibly ranks, $\dot{T}$ is typically independent of both. For applications with Q phases,

usually $\dot{T} = O(\bar{\tau}/Q)$, a substantial reduction. In our results, profiles are 100–1000 KB and nearly constant.

We improve path profiling not only by collecting representative paths, but also by enabling dynamic task discovery and high-resolution attribution. Current path profiling [16] relies on a set of static tasks names as input. Moreover, task costs are attributed at a very coarse (function) resolution. We show how to attribute costs to program statements in full calling context, the same resolution as standard profilers [1, 18].

Figure 6 shows a task histogram as a calling context tree [3] (CCT), a compact way of representing common ancestors in calling contexts. Histogram entries correspond to leaves in the CCT; a leaf is a program statement. Several leaves represent one task, providing good attribution detail. Instead of maintaining k CCTs, we represent all paths' contexts with one tree. Each entry has k cost metrics for its contribution to each of the k paths. During profiling, each rank has a CCT; post-processing computes the union.

Although $\dot{T}$ refers only to histogram entries (leaves), the total CCT size is typically bounded. First, the number of distinct calling contexts often reaches a maximum. Second, call path lengths are often bounded. When they are not, recursion is often in play; and recursive frames can be detected and elided.

It is easy to summarize a rank's *committed* tasks using a per-rank histogram. The basic challenge of high-resolution path profiling is managing *uncommitted* tasks on candidate paths. Recall that a candidate task is uncommitted until a path allreduce selects and commits it. Because any candidate path could be selected, each rank must maintain k candidates until a phase ends. Prior path profiling methods maintained k candidate histograms, one histogram per path. That is, the Path-summary was a histogram; to be sent from rank to rank, its size had to be small and fixed. As a result, path costs were attributed to a small, fixed set of coarse tasks.

Another challenge is that, to ensure maximum scalability, we introduce more stringent space requirements than with collecting full paths. To ensure that permanent space is always $O(k\dot{T})$, we forbid spilling uncommitted tasks. (With full paths spilling can be small relative to path length; with a profile, spills are large relative to the histogram.) Furthermore, to avoid memory pressure during long phases, we restrict the space for holding uncommitted tasks to $O(k\dot{T})$. To make these stricter guarantees, we allow for the possibility of decreasing the resolution of task attribution. However, in practice we rarely have make the sacrifice.

Our solution combines three techniques. *First*, we use asynchronous sampling for high-resolution attribution. We sample based on time. With enough time, paths accumulate samples in the most important parts of the most important tasks. On each sample, we unwind the call stack for calling context [32]. Tasks have function calls and many lines of code. Sampling identifies the statements that are executed most frequently — without low-level instrumentation. Because the sampling trigger is a user-defined time interval, it can be adjusted to increase or decrease resolution. We typically use periods on the order of 1000–5000 μs, resulting in sampling frequencies of 1000–200 samples/second. A disadvantage of sampling is that it captures task costs but not instances.

The *second* technique packs a high-resolution uncommitted path histogram into a small Path-summary. Figure 6 shows how we

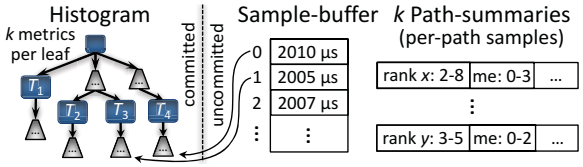

Figure 6: Profiling data structures, replicated per rank.

manage uncommitted samples with two structures, the Sample-buffer and Path-summary. The Sample-buffer (cf. Fig. 4(b)) holds pointers to uncommitted samples on a *rank*. The Path-summary (cf. Fig. 4(c)) contains Sample-buffer intervals for the uncommitted samples on a *path*. On a sample (Fig. 4, line 3) we first eagerly create a histogram entry. (Eagerly creating a CCT entry not only makes commits fast, it saves space and time by reusing existing context.) Then, we push the sample's commit data on the Sample-buffer; and update each Path-summary, using the same interval compression technique described earlier (§5). When a phase ends, the algorithm commits the Sample-buffer entries on the selected paths (Fig. 3, line 34) by updating CCT metrics.

With this technique, sampling attributes in *higher* resolution than with full paths but places *less* pressure on the Path-summary's size. Given a contiguous series of task instances, only one Path-summary interval is needed for many samples. Thus, sampling requires *at most* the same number of intervals as full paths; but it may require *fewer* intervals when small tasks are not sampled.

The *third* technique bounds the sizes of the Sample-buffer and Path-summary by relaxing the resolution of attribution. The Sample-buffer must hold all the samples on a rank during a phase. To avoid a size proportional to samples, we use adaptive sampling to bound its size. As the Sample-buffer begins to fill, we increase the sampling period using exponential decay. We precompute several thresholds corresponding to increasing Sample-buffer load. Each time the load passes the next threshold, we double the sampling period. We reset the sampling period at the end of each phase.

In practice, the need for adaptive sampling is rare. In most cases, it is easy to allocate a Sample-buffer that comfortably holds all samples; the size can be estimated given the sampling period and a typical phase time. For instance, with a 4000 μs sampling period, 16K entries accounts for (long) 65 second phases without any adaptivity. Our results were gathered with a per-rank Sample-buffer of 16K entries; only rarely did the load pass the first threshold. The most common task that causes our profiler to use adaptive sampling is sequential I/O on large files.

To bound the Path-summary, we reserve a special slot for the 'unknown location.' This slot represents only sample time; no location. When there is no longer space to create a new interval, `rp_updatePathSum` randomly selects an interval, merges it with this reserved slot and then creates a new interval (Fig. 4(c), line 5). Alternatively, the user can increase the Path-summary size. Usually, a Path-summary with 32 intervals retains full resolution (§7.5).

LULESH: Percentile's Path Length (v. Base)

Ranks	100	89%	78%	68%	57%	42%	31%	21%	10%	0%
64	1.00	1.00	1.00	1.00	1.00	0.79	0.79	0.79	0.79	0.79
125	1.01	1.01	1.01	1.01	1.01	0.76	0.76	0.76	0.76	0.76
216	0.99	0.99	0.99	0.99	0.99	0.78	0.78	0.78	0.78	0.78
512	0.99	0.99	0.99	0.99	0.99	0.75	0.75	0.75	0.75	0.75
1000	0.99	0.99	0.98	0.98	0.98	0.73	0.73	0.73	0.73	0.73
2197	0.99	0.98	0.98	0.98	0.98	0.66	0.66	0.66	0.66	0.66
4096	0.98	0.98	0.98	0.98	0.98	0.64	0.64	0.64	0.64	0.64
6859	0.97	0.96	0.96	0.96	0.96	0.60	0.60	0.60	0.60	0.60

GTC: Percentile's Path Length (v. Base)

Ranks	100	86%	73%	60%	40%	26%	13%	0%
32	0.99	0.99	0.99	0.99	0.79	0.79	0.79	0.79
64	0.99	0.99	0.99	0.99	0.87	0.87	0.86	0.86
128	0.99	0.99	0.99	0.98	0.82	0.82	0.82	0.82
256	0.98	0.98	0.97	0.97	0.81	0.81	0.81	0.81
512	0.97	0.96	0.96	0.96	0.79	0.79	0.79	0.79
1024	0.92	0.92	0.92	0.91	0.75	0.75	0.75	0.74
2048	0.90	0.90	0.90	0.90	0.68	0.68	0.68	0.68
4096	0.85	0.85	0.85	0.85	0.60	0.60	0.60	0.60

HPCG: Percentile's Path Length (v. Base)

Ranks	100	86%	73%	60%	40%	26%	13%	0%
48	1.00	1.00	1.00	1.00	0.97	0.97	0.97	0.97
96	1.00	1.00	0.99	0.99	0.95	0.95	0.95	0.95
192	1.00	1.00	1.00	1.00	0.94	0.94	0.94	0.94
384	1.00	1.00	1.00	1.00	0.92	0.92	0.92	0.92
768	0.99	0.99	0.98	0.98	0.89	0.89	0.89	0.88
1536	1.01	1.01	1.00	1.00	0.84	0.84	0.84	0.84
3072	1.01	1.01	1.01	1.01	0.81	0.81	0.81	0.81
6144	0.99	0.99	0.99	0.99	0.79	0.79	0.78	0.78

PFLOTRAN: Percentile's Path Length (v. Base)

Ranks	100	86%	73%	60%	40%	26%	13%	0%
20	1.01	1.01	1.01	1.01	0.54	0.54	0.54	0.54
40	1.01	1.01	1.01	1.01	0.32	0.32	0.31	0.31
60	0.97	0.97	0.96	0.96	0.29	0.29	0.28	0.28
80	1.03	1.03	1.02	1.02	0.29	0.28	0.28	0.28
100	1.09	1.09	1.09	1.09	0.29	0.28	0.28	0.28

Base: Unmonitored execution
x%: Representative path at x^{th} percentile
100%: Most critical path
0%: Least critical path

Sweep: Percentile's Path Length (v. Base)

Ranks	100	86%	73%	60%	40%	26%	13%	0%
32	0.99	0.99	0.99	0.99	0.66	0.66	0.66	0.66
64	1.00	0.99	0.99	0.99	0.59	0.59	0.59	0.59
128	1.00	1.00	0.99	0.99	0.51	0.51	0.51	0.51
256	1.00	1.00	0.99	0.99	0.41	0.41	0.41	0.41
512	1.00	1.00	0.99	0.99	0.32	0.32	0.32	0.32
1024	1.00	0.99	0.99	0.99	0.24	0.24	0.24	0.24
2048	1.01	1.01	1.00	1.00	0.19	0.19	0.19	0.19
4096	1.00	1.00	1.00	0.99	0.14	0.14	0.14	0.14

SVM: Percentile's Path Length (v. Base)

Ranks	100	86%	73%	60%	40%	26%	13%	0%
20	1.00	0.97	0.97	0.96	0.95	0.95	0.94	0.93
40	0.99	0.94	0.93	0.93	0.90	0.90	0.90	0.86
60	0.99	0.91	0.91	0.90	0.87	0.86	0.86	0.81
80	0.98	0.88	0.88	0.87	0.83	0.83	0.82	0.76
100	0.98	0.87	0.86	0.85	0.81	0.80	0.79	0.72

PgRank: Percentile's Path Length (v. Base)

Ranks	100	86%	73%	60%	40%	26%	13%	0%
8	1.00	0.25	0.24	0.22	0.04	0.02	0.02	0.00
16	1.00	0.22	0.21	0.19	0.02	0.01	0.01	0.00
24	1.00	0.19	0.18	0.17	0.01	0.01	0.00	0.00
32	1.00	0.15	0.14	0.13	0.01	0.01	0.00	0.00

Figure 7: Distribution of lengths for representative path profiles.

7 EVALUATION

We evaluate representative paths analysis on seven MPI applications executing on up to 7000 ranks. We use parallel post-mortem analysis to process the results [30]. Our applications cover numerical solvers, graphs, and machine learning; encompass several distinct communication/synchronization paradigms; and range from proxy to production applications. We highlight the major communication patterns besides collectives such as allreduce and barrier.

The first two applications are well-known proxies. LULESH [19] is a Lagrangian hydrodynamics benchmark that simulates a hydrodynamic shock using a 3-dimensional unstructured hex mesh. HPCG [10] is conjugate gradient solver using sparse matrices. Both use isend, irecv, and waitall. Sweep3D [37] (denoted 'Sweep'), a neutron transport benchmark, features pipeline communication using blocking recv. GTC [20], a particle-in-cell code for turbulence simulation in a toroid, features sendrecv and gather. We weak scale these four applications on a cluster with an FDR InfiniBand interconnect and nodes that have dual 12-core Intel Haswell CPUs (E5-2670v3 2.3 GHz) and 64 GB RAM (DDR4-2133).

For the next three applications, we select realistic workloads where it is desirable to improve response time with strong scaling. Results are collected on a cluster with FDR InfiniBand and nodes that have dual 10-core Intel IvyBridge CPUs (E5-2680v2, 2.8 GHz) and 128 GB RAM (DDR3-1866).

PFLOTRAN is a production subsurface flow and reactive transport simulator [14]. PFLOTRAN invokes linear and non-linear solvers from PETSc [4] (Portable Extensible Toolkit for Scientific Computation). PETSc provides extensive coverage of two-sided MPI functionality including persistent non-blocking communication.

MaTEx's SVM with shrinking [34] (denoted 'SVM') is a distributed parallel variant of the popular data analytics algorithm, Support Vector Machine. To decrease training time, this version adaptively eliminates samples ('shrinking') that are unlikely to contribute to the final classifier.

PageRank [23] (denoted 'PgRank') is a distributed-memory implementation of the Page Rank [7] algorithm. We use a relatively large power-law graph with 11 million vertices and 1.3 billion directed edges. Communication is personalized all-to-alls using isend and waitall.

Below we show several sets of results. First, we show the distributions of representative path lengths as each application scales. Next we show several pieces of information about collecting the paths. To show our techniques can be used online, we show execution overhead. To show the benefits of our path pruning, we show the permanent space reduction and space scalability of our algorithms. To show accuracy, we validate path lengths. To show that our approach can compactly represent paths, we show the maximum size Path-summary structure needed to collect full resolution paths.

7.1 Distribution of Path Lengths

Collecting k representative paths produces a distribution of path lengths similar to Figure 1(d)). In general, we expect the distribution to be dependent on both the application and its inputs. However, we can make some general observations. In MPI data-parallel SPMD programs, data and MPI ranks are closely related. Therefore, we expect the path length distribution to reflect the data distribution. If the data distribution is uniform, we would expect many paths to be about the same length; if there are active and inactive data cells, we would expect a bi-modal distribution. Furthermore, combinatorics suggests many distinct paths will have similar lengths. Given P ranks, for every 2-way branch there are 2 times more paths (1 path becomes two); and for every all-to-all synchronization, there are P^2 more paths (each rank has a P-way branch). With so many paths, there are likely many combinations of tasks that produce paths of similar length.

Figure 7 shows distributions of representative path lengths as each application scales. We collected representative path profiles for 8 and 10 paths. The chart's percentiles are determined by the number of paths. Percentiles 100% and 0% are the most and least critical path. For each percentile, the charts show path length as a percentage of the base execution time. Note that whereas base time

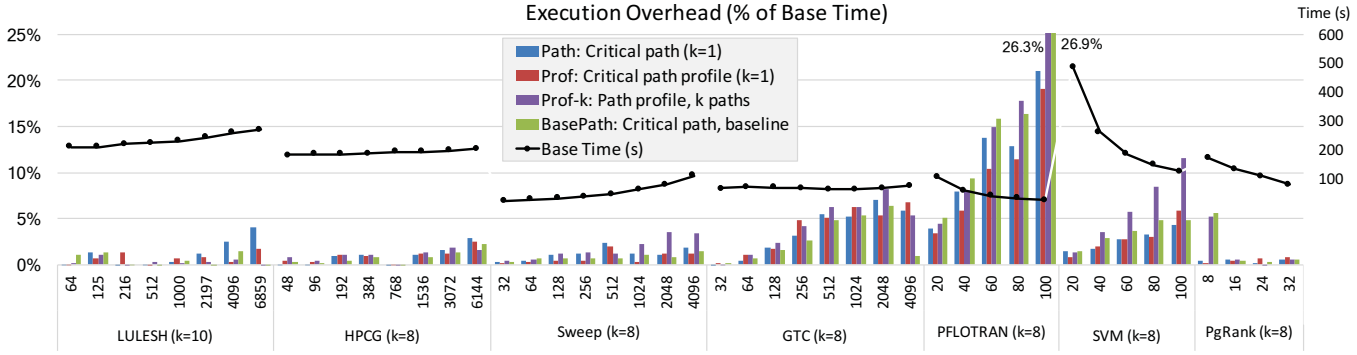

Figure 8: Execution overhead for collecting representative paths and path profiles.

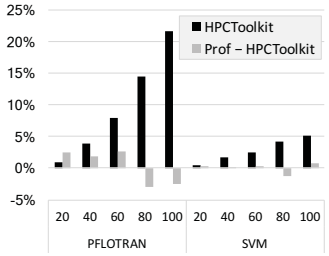

Figure 9: Execution overhead for baseline call path profiler.

includes all communication and synchronization time, our paths *exclude* it; they only include task compute time.

The figure's charts are colored as heat maps, from blue (low) to red (high). We make four observations.

First, the relative distance from slowest to fastest paths is quite different across applications. Whereas the most critical path is 100% of the base execution (except when network time is notable), the least critical path ranges from a substantial (HPCG) to miniscule (PgRank) fraction of the base execution. As a result for each heat map, the smallest value (darkest blue) is relative to each application. The reason that some of GTC's critical path lengths are under 100% at larger scales is that the paths actually begin to include network time. PFLOTRAN's critical path lengths are over 100% because of measurement error. Both are discussed further in §7.4.

Second, the distributions of path lengths can be quite different, showing the need for representative paths. SVM has an approximately uniform distribution. PgRank's paths follow a skewed distribution. Its critical path is skewed by serial I/O; other paths are skewed by the power-law input graph. Distributions for other applications are bi-modal (LULESH, HPCG, Sweep, GTC, PFLOTRAN). In these cases, most program paths are similar to either the 100% or 0% paths. The reason is that many data-parallel applications have an imbalance in data partitioning. Suppose the unit of data partitioning is a 'cell'. Although there may be differences between various 'active' cells, the differences are negligible relative to the 'inactive' cells.

Third, the character of the length distributions does not change with ranks. This accords with the fact that the applications perform the same computation with scale.

Fourth, the distribution of path lengths quantifies the change in parallel efficiency as the application scales to more ranks. Observe that the spread of bimodal distributions typically expand with scale, corresponding to decreased efficiency. For LULESH at 6859 ranks, about 50% of the paths are executing at high parallel efficiency (about 97% of base run time). In contrast, the other 50% of paths only account for 60% of base run time. An exception is GTC, where the efficiency of the shorter paths *increases* before decreasing.

An interesting question is how to select good values of k. Larger values of k yield smaller bounds on percentiles but increase overhead: communicating Path-sets at branches; selecting representative paths at joins. A first-order analysis focuses on the network bandwidth constraint. As noted in §5, the Path-set size should be within MPI's eager threshold limit t, a value that is related to network latency and MPI communication buffer sizes. This yields the inequality $k(8l) \leq t$, where l is the Path-summary length and each Path-summary interval is 8 bytes (§5). If $t = 12$ KB (MVAPICH's default for Mellanox ConnectX HCAs [33]) and $l = 128$ (cf. §7.5), then $k \leq 12$, similar to our chosen values.

7.2 Overhead

Figure 8 shows the execution overhead for collecting representative paths and path profiles. For each application, the figure shows the base execution time (Base, in seconds, right axis) and the overhead for four representative path variants. Path is a full critical path using our pruning; Prof and Prof-k are path profiles with 1 path (critical) and k paths, respectively; and BasePath is a full critical path using standard pruning (space equivalent to a trace). The overhead is relative to all application activities but excludes any post-application data storage and processing.

For LULESH, HPCG, Sweep, and PgRank, overhead is 5% or less, which is considered very good. Overhead is higher on PFLOTRAN (up to 27%); and to a lesser extent SVM (up to 12%), and GTC (5–7%). The reason for overhead on PFLOTRAN and SVM is that strong scaling makes synchronization very frequent. At 100 ranks, SVM performs a collective every 1200 μs; and PFLOTRAN issues an MPI operation every 150 μs. At these frequencies, collecting calling contexts (stack unwinding) and selecting paths at joins and collectives introduces noticeable overhead. Despite the overhead, path lengths are still accurate (§7.4). The reason for overhead on

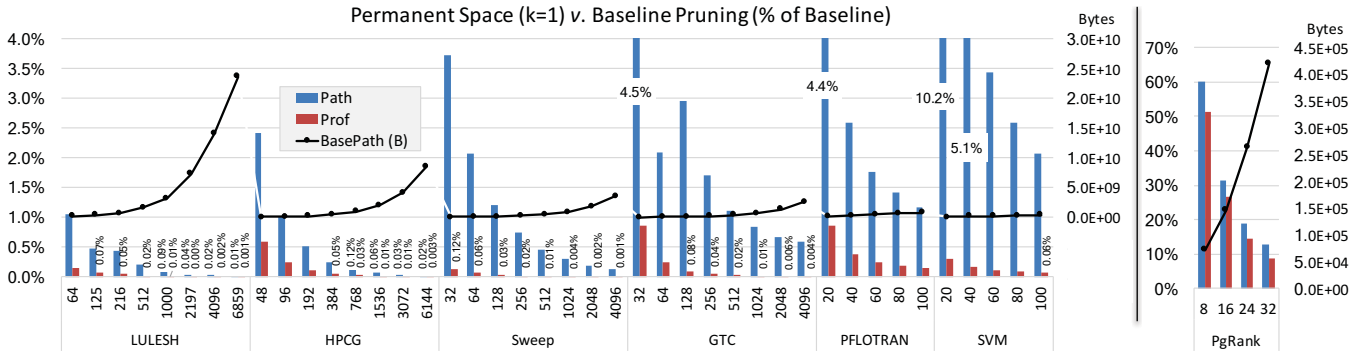

Figure 10: Permanent space benefits of full pruning and profiling: space relative to baseline pruning.

GTC is that, at larger scales, its sendrecvs are very sensitive to imbalance; see §7.4.

Usually the overhead between the variants is similar. Although one might expect BasePath to have more overhead, I/O buffering makes its performance similar to profiles. With PFLOTRAN, BasePath overhead is higher because of the operation frequency. With PFLOTRAN and SVM, Prof-k is higher: because of operation frequency, the additional time needed to select paths is noticeable.

To show that overheads on PFLOTRAN and SVM are due more to general monitoring than specifically collection of representative paths, Figure 9 compares our results with the execution overhead for a baseline per-process profiler, HPCToolkit [1]. HPCToolkit, part of the Exascale Computing Project, uses sampling and call path unwinding to collect *per-process* call path profiles. We configure HPCToolkit to sample on time (`REALTIME`) at the same frequency as our representative path profiler. On each sample, both tools unwind the call stack using HPCToolkit's unwinder. The figure shows absolute overhead for HPCToolkit as well as the difference between HPCToolkit and Prof ($k = 1$). The figure shows that sampling and unwinding account for most of the overhead.

7.3 Space Reduction and Space Scalability

Space reduction. Figure 10 highlights space reduction by showing, for each application, the permanent space required for the standard full paths algorithm (BasePath, in bytes, right axis) and then the relative space required by our full paths (Path) and path profiles (Prof). Lower is better. Because the standard algorithm (BasePath) has space equivalent to an MPI trace, space quickly grows to gigabytes and is clearly not scalable. (PgRank is an outlier because it has so few MPI operations.) In contrast, with Path, pruning reduces storage to space proportional to path length. The savings increase with scale because typically a factor of P ranks is removed. At the largest scale, Path requires 0.01%, 0.02%, 0.13% and 0.6% (LULESH, HPCG, Sweep, GTC) the size of BasePath. Below, we explain why the reduction is less for Sweep and GTC.

Path still requires space proportional to execution time. Prof solves this problem, yielding nearly constant space with respect to ranks and execution time (see below). At the largest scale, Prof requires space that is 0.001–0.004% (LULESH, HPCG, Sweep, GTC) of BasePath. The profile sizes are 100–1000 KB.

Space scalability. Figure 11 shows space scalability with respect to ranks, execution time, and number of paths. All results are normalized so that 1 represents perfect scaling. We especially note that path profiles scale with respect to ranks, time, and paths.

To show how space scales with *ranks*, Figure 11(a) shows the size of full paths (Path) and profiles (Prof) relative to *the smallest rank*. When space is independent of ranks, this value should be 1. Path scales well; usually results are less than 2. The exceptions are Sweep and GTC: with Sweep, paths are related to $\sqrt{P}$; with GTC, communication increases with ranks. Prof shows ideal or better-than-ideal scaling. The reason for values smaller than 1 is that the calling context tree is stored in a text-based XML format. With different samples, long call paths and function names yield unexpected size differences.

To show how space scales with *execution time*, Figure 11(b) shows the size of paths (Path) and path profiles (Prof) for a time-extended execution relative to the base. To extend time, we selected an application-specific factor and increased iterations by that factor; we used 2× or 10×. Because paths are proportional to execution time, given a factor $\alpha\times$ we expect space to increase by that factor. Because profiles should *not* be proportional to execution time, we expect space to be identical to the base execution, i.e., it should *not* increase. A result of 1 means that space conforms to the above expectations. The results for Path are very close to ideal. The exceptions (HPCG, PgRank) are when there is a substantial amount of initialization relative to the iteration count. (For this GTC input, increasing iterations resulted in numerical instability.) The results for Prof are at or better than ideal. Values can be smaller than 1 because of the XML format.

To show that *profiles* scale well with respect to *number of paths*, Figure 11(c) shows the size of a profile with k paths relative to one path. We set k to either 8 or 10. Because a profile contains a set of metrics for each path, we expect space to scale linearly with k. A result of 1 means that space conforms to the above expectations. The results are typically well below 1. The reason is that although task metrics do scale linearly with k, most of the calling context for each task can be shared via the calling context tree. (We do not show results for Path because savings from a shared CCT are inconsequential relative to path lengths.)

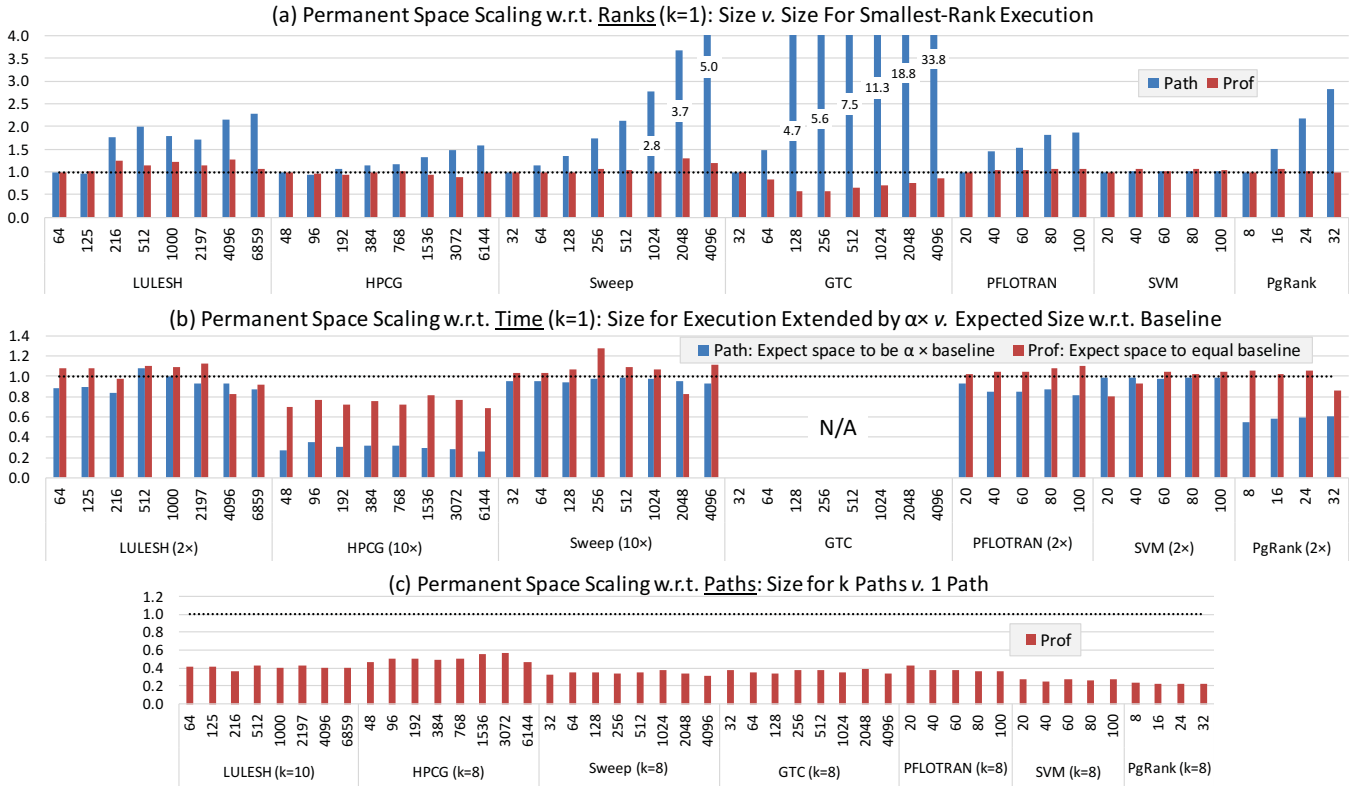

Figure 11: Permanent space benefits of full pruning and profiling: space scalability with (a) ranks, (b) time, and (c) paths.

7.4 Validation

Figure 12 shows validation results. We validated full (Path) and profiled (Prof) critical paths ($k = 1$) by comparing the path's length with base execution time. Note that whereas base time *includes* all communication and synchronization time, our paths *exclude* it; they only include task compute time. Nevertheless, base time is a good proxy for path compute time because *critical* network time for either communication or synchronization is usually small.

The validation charts show percent error relative to base time. Most deviations are within 1–3%. The exceptions are GTC, PFLOTRAN, and PgRank. GTC's consistently negative errors are a case that violates the above assumption about base time. As GTC scales, its sendrecvs result in a critical path that contains network time. PgRank's path also includes network time due to personalized all-to-alls with large payloads. PFLOTRAN's deviations are due to overhead-induced measurement errors.

7.5 Effectiveness of Path Summarization

The Path-summary structure (Fig. 4(c)) summarizes program paths so that at a phase end, only needed tasks or samples are committed. A key question is how effectively it compresses paths into a small fixed space. Figure 13 shows the number of Path-summary intervals needed to represent paths in maximum resolution. Most paths can be fully captured using fewer than 32 intervals. These applications have nearest-neighbor communication. The exceptions are Sweep and SVM. In Sweep, path lengths are related to $\sqrt{P}$; SVM's shrinking phase is related to ranks.

8 RELATED WORK

Critical paths are important for performance analysis. Unlike per-process profiles or traces, program paths provide information on program dependences. For example, when load imbalance causes waiting time, a standard profile may show that the application spends time in an MPI progress engine wait loop. It does not show — as does representative paths (Figure 2) — the exact task that receives disproportionate work.

Prior work on critical paths has been motivated by two classic problems. First, how to collect paths given that paths are exponential in the number of branches. Second, how to use paths for diagnostic insight given that paths are long and hard to present.

Representative Paths. All prior work that we are aware of focuses on either the most/least critical path or the top/bottom k paths. Alexander et al. [2] give several algorithms for finding the top k most critical paths. Unfortunately, because the number of paths is exponential in branches, the top k program paths are often nearly identical. Representative paths at several percentiles provides more diagnostic information.

Critical Paths. The straightforward critical path algorithm [8] requires retaining multiple paths per process. Yang et al. [38] present

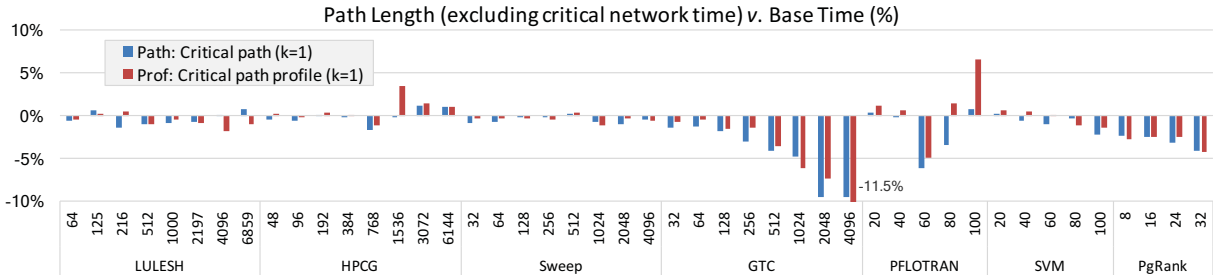

Figure 12: Validation of full and profiled critical paths, relative to base execution time.

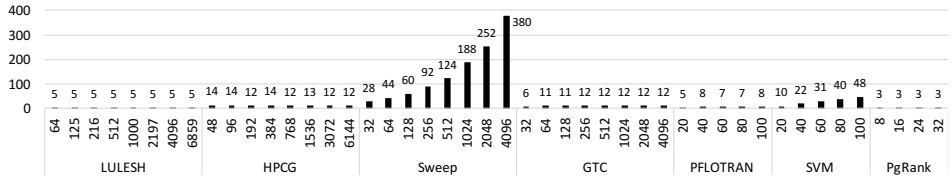

Figure 13: Maximum length of path-summary structure for full-resolution attribution.

a longest path method that retains one path per MPI rank. Several others adopt this approach [9, 11, 16, 17, 25, 26]. Schulz [27] presents a slack-based method that uses online pruning rules to reduce the size of data collection down to one path per rank.

Critical Path Profiles. A critical path profile jettisons task sequencing to avoid the cost of collecting full paths. Unfortunately current methods [16] do not perform dynamic task discovery but require as input a set of static tasks such as function names. Moreover, they attribute critical path costs only to tasks, which is much lower resolution than standard program profiles (program statements) [1, 18]. Shotgun profiling uses statistical and hardware-assisted techniques to reconstruct (with low overhead) intra-thread dependence graphs [13].

Approximate Critical Paths/Tasks. Exact critical path analysis is costly. Sometimes it is sufficient to approximate and identify *likely* critical paths or tasks. Zhu et al. [39] identify likely critical tasks by retaining profile information only for tasks that do not block. The Adagio [24] runtime monitors program blocking to identify tasks that *cannot* be on the critical path. DuBois uses an analogous technique to determine when threads are critical [12]. Su et al. [29] develop a heuristic to detect when tasks are on a critical path and should receive scheduling priority. Blame shifting identifies likely critical tasks by profiling functions that execute while compute threads idle waiting for work [31].

Diagnostic Insight. Much prior work focuses on using critical paths for diagnostic insight. The IPS [38] tool reported blocking edges and procedures on the critical path. Scalasca generates metrics that highlight blocking and load imbalance [6]. Barford et al. identified bottlenecks in TCP stacks [5]. Carnival used waiting time analysis — postmortem comparison of different execution paths — to identify the causes of blocking [36].

9 CONCLUSIONS

Detailed critical path analysis of MPI program paths is usually avoided because of its space and time costs. This paper presents a family of scalable algorithms that could encourage more performance analysts to turn to program paths for insight. Representative paths provides more diagnostic insight than profiles, traces, or critical paths. Our algorithm for full paths not only provides more diagnostic insight than a per-rank trace, it also has the best possible space requirements. Our path profiles have the same space bounds as regular profiles, but provide more diagnostic insight.

We believe the most important contribution of our work is representative path profiling. Representative paths samples the distribution of all program paths, not just the most or least critical. Because path-based analysis respects dependences, there are no gaps between tasks and no task overlap. In contrast, with a standard profile analyzing tasks along any rank may ignore blocking time; and analyzing tasks across ranks may ignore overlap. Furthermore, path profiles are scalable with respect to both MPI ranks and execution time. Their sizes are nearly constant in practice and very small (100-1000 KB). Finally, with our work, path profiles attribute costs to program statements in full calling context, the same resolution as state-of-the art per-rank profilers. Furthermore, the overhead of path profiles is low (under 5%) when synchronization intervals are a few milliseconds. When overheads are higher (10–26%), the overheads of a standard (per-rank) call path profiler are similarly high.

When it is important to preserve task sequencing within program paths, our full paths algorithm requires space proportional to path lengths, the minimum. However, an open problem with full-path analysis is effective presentation to a human analyst. Paths are very long, and their dynamic structure may not correspond to static program elements.

Although our techniques are designed for a distributed memory programming model, we believe they can influence critical paths in shared memory. To achieve performance with shared memory,

maximizing locality is critical. Our algorithms are designed to move a minimal amount of data between locality domains.

ACKNOWLEDGMENTS

We are grateful for funding support from the U.S. Department of Energy's (DOE) Office of Advanced Scientific Computing Research as part of "Integrated End-to-end Performance Prediction and Diagnosis" and the "Center for Advanced Technology Evaluation" (CENATE).

Pacific Northwest National Laboratory is operated by Battelle for the DOE under Contract DE-AC05-76RL01830. A portion of this research was performed using PNNL Institutional Computing.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurr. Comput.: Pract. Exper.* 22, 6 (2010), 685–701. https://doi.org/10.1002/cpe.1553

[2] C. Alexander, D. Reese, and J. Harden. 1994. Near-critical path analysis of program activity graphs. In *Proc. of the Second Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, Los Alamitos, CA, USA, 308–317. https://doi.org/10.1109/MASCOT.1994.284406

[3] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proc. of the 1997 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/258915.258924

[4] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. 2016. PETSc Web page. http://www.mcs.anl.gov/petsc. (2016).

[5] Paul Barford and Mark Crovella. 2000. Critical Path Analysis of TCP Transactions. *SIGCOMM Comput. Commun. Rev.* 30, 4 (Aug. 2000), 127–138. https://doi.org/10.1145/347057.347416

[6] D. Bohme, F. Wolf, B.R. De Supinski, M. Schulz, and M. Geimer. 2012. Scalable Critical-Path Based Performance Analysis. In *Proc. of the 26th IEEE Intl. Parallel and Distributed Processing Symp.* IEEE Computer Society, Los Alamitos, CA, USA, 1330–1340. https://doi.org/10.1109/IPDPS.2012.120

[7] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems: Proc. of the 7th Intl. World Wide Web Conference* 30, 1 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

[8] K. Mani Chandy and J. Misra. 1982. Distributed Computation on Graphs: Shortest Path Algorithms. *Commun. ACM* 25, 11 (Nov. 1982), 833–837. https://doi.org/10.1145/358690.358717

[9] Jian Chen and R.M. Clapp. 2015. Critical-path candidates: Scalable performance modeling for MPI workloads. In *Proc. of the 2015 IEEE Intl. Symp. on Performance Analysis of Systems and Software*. IEEE, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/ISPASS.2015.7095779

[10] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance Conjugate-gradient Benchmark. *Int. J. High Perform. Comput. Appl.* 30, 1 (Feb. 2016), 3–10. https://doi.org/10.1177/1094342015593158

[11] I Dooley and L.V. Kale. 2010. Detecting and using critical paths at runtime in message driven parallel programs. In *Proc. of the 12th Workshop on Advances in Parallel and Distributed Computational Models (held in conjuction with IPDPS 2010)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–8. https://doi.org/10.1109/IPDPSW.2010.5470844

[12] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. 2013. Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 511–522. https://doi.org/10.1145/2508148.2485966

[13] Brian A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. 2004. Interaction Cost and Shotgun Profiling. *ACM Trans. Archit. Code Optim.* 1, 3 (Sept. 2004), 272–304. https://doi.org/10.1145/1022969.1022971

[14] G. E. Hammond, P. C. Lichtner, and R. T. Mills. 2014. Evaluating the performance of parallel subsurface simulators: An illustrative example with PFLOTRAN. *Water Resources Research* 50, 1 (2014), 208–228. https://doi.org/10.1002/2012WR013483

[15] Debra Hensgen, Raphael Finkel, and Udi Manber. 1988. Two algorithms for barrier synchronization. *Intl. Journal of Parallel Programming* 17, 1 (1988), 1–17. https://doi.org/10.1007/BF01379320

[16] J.K. Hollingsworth. 1998. Critical path profiling of message passing and shared-memory programs. *IEEE Trans. Parallel Distrib. Syst.* 9, 10 (Oct 1998), 1029–1040. https://doi.org/10.1109/71.730530

[17] Jeffrey K. Hollingsworth. 1996. An Online Computation of Critical Path Profiling. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/238020.238024

[18] Intel Corporation. 2017. Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe. (March 2017).

[19] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. Lawrence Livermore National Laboratory. 1–9 pages.

[20] Zhihong Lin, Stephane Ethier, and Jerome Lewandowski. 2010. Gyrokinetic Toroidal Code, v. 2. http://phoenix.ps.uci.edu/GTC/. (2010).

[21] Message Passing Interface Forum. 1999. *MPI: A Message Passing Interface Standard*. University of Tennessee, Knoxville. http://www.mpi-forum.org/docs/mpi-11.ps.

[22] John F. Monahan. 2011. *Numerical Methods of Statistics* (2nd ed.). Cambridge University Press, New York, NY, USA.

[23] A. Panyala, S. Krishnamoorthy, and D. Chavarria-Miranda. 2016. *Efficient Approximation of the Iterative PageRank Algorithm*. Technical Report. Pacific Northwest National Laboratory.

[24] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd Intl. Conf. on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 460–469. https://doi.org/10.1145/1542275.1542340

[25] A.G. Saidi, N.L. Binkert, S.K. Reinhardt, and T. Mudge. 2008. Full-System Critical Path Analysis. In *Proc. of the 2008 IEEE Intl. Symp. on Performance Analysis of Systems and Software*. IEEE, Los Alamitos, CA, USA, 63–74. https://doi.org/10.1109/ISPASS.2008.4510739

[26] Ali G. Saidi, Nathan L. Binkert, Steven K. Reinhardt, and Trevor Mudge. 2009. End-to-end Performance Forecasting: Finding Bottlenecks Before They Happen. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 361–370. https://doi.org/10.1145/1555815.1555800

[27] Martin Schulz. 2005. Extracting Critical Path Graphs from MPI Applications. In *Proc. of the 2005 IEEE Intl. Conf. on Cluster Computing*. IEEE, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/CLUSTR.2005.347035

[28] Martin Schulz, Greg Bronevetsky, and Bronis R. Supinski. 2008. *Recent Advances in Parallel Virtual Machine and Message Passing Interface: Proc. of the 15th European PVM/MPI Users' Group Meeting*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter On the Performance of Transparent MPI Piggyback Messages, 194–201. https://doi.org/10.1007/978-3-540-87475-1_28

[29] ChunYi Su, Dong Li, Dimitrios S. Nikolopoulos, Matthew Grove, Kirk Cameron, and Bronis R. de Supinski. 2012. Critical Path-based Thread Placement for NUMA Systems. *SIGMETRICS Perform. Eval. Rev.* 40, 2 (Oct. 2012), 106–112. https://doi.org/10.1145/2381056.2381079

[30] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *Proc. of the 2010 ACM/IEEE Conf. on Supercomputing*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.47

[31] Nathan R. Tallent and John Mellor-Crummey. 2009. Effective Performance Measurement and Analysis of Multithreaded Applications. In *Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/1504176.1504210

[32] Nathan R. Tallent, John Mellor-Crummey, and Michael W. Fagan. 2009. Binary Analysis for Measurement and Attribution of Program Performance. In *Proc. of the 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM, New York, NY, USA, 441–452. https://doi.org/10.1145/1542476.1542526

[33] Ohio State University. 2017. MVAPICH2 2.3 User Guide. http://mvapich.cse.ohio-state.edu. (March 2017).

[34] A. Vishnu, J. Narasimhan, L. Holder, D. Kerbyson, and A. Hoisie. 2015. Fast and Accurate Support Vector Machines on Large Scale Systems. In *Proc. of the 2015 IEEE Intl. Conf. on Cluster Computing*. IEEE, Los Alamitos, CA, USA, 110–119. https://doi.org/10.1109/CLUSTER.2015.26

[35] Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57. https://doi.org/10.1145/3147.3165

[36] Jr. Wagner Meira, Thomas J. LeBlanc, and Alexandros Poulos. 1996. Waiting time analysis and performance visualization in Carnival. In *Proc. of the 1996 SIGMETRICS Symp. on Parallel and Distributed Tools*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/238020.238023

[37] Harvey Wasserman. 1995. ASCI Sweep3D Benchmark Code, v. 2.2b. http://wwwc3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html. (December 1995).

[38] C.-Q. Yang and Barton P. Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *Proc. of the 8th Intl. Conf. on Distributed Computing Systems*. IEEE, Los Alamitos, CA, USA, 366–373. https://doi.org/10.1109/DCS.1988.12538

[39] Wenbin Zhu, P.G. Bridges, and AB. Maccabe. 2005. Online Critical Path Profiling for Parallel Applications. In *Proc. of the 2005 IEEE Intl. Conf. on Cluster Computing*. IEEE, Los Alamitos, CA, USA, 1–9. https://doi.org/10.1109/CLUSTR.2005.347048