Recovering Logical Structure from Charm++ Event Traces

Katherine E. Isaacs[†,⋆], Abhinav Bhatele[⋆], Jonathan Lifflander[§], David Böhme[⋆], Todd Gamblin[⋆], Martin Schulz[⋆], Bernd Hamann[†], Peer-Timo Bremer[⋆]

[†]Department of Computer Science, University of California, Davis, CA
[⋆]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA
[§]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL
[†]keisaacs@ucdavis.edu, [⋆]bhatele@llnl.gov, [§]jliffl2@illinois.edu

ABSTRACT

Asynchrony and non-determinism in Charm++ programs present a significant challenge in analyzing their event traces. We present a new framework to organize event traces of parallel programs written in Charm++. Our reorganization allows one to more easily explore and analyze such traces by providing context through logical structure. We describe several heuristics to compensate for missing dependencies between events that currently cannot be easily recorded. We introduce a new task ordering that recovers logical structure from the non-deterministic execution order. Using the logical structure, we define several metrics to help guide developers to performance problems. We demonstrate our approach through two proxy applications written in Charm++. Finally, we discuss the applicability of this framework to other task-based runtimes and provide guidelines for tracing to support this form of analysis.

CCS Concepts

•**General and reference** → **Measurement; Metrics; Performance;**

Keywords

asynchrony, task-based models, trace analysis, performance

1. MOTIVATION

Task-based programming models and their associated runtimes are receiving renewed attention because they can exploit fine-grained parallelism and heterogeneous hardware without overburdening developers. Examples of such runtimes include Charm++ [14, 15], the Open Community Runtime [2], Legion [4], OmpSs [7], and Cilk [9]. All of these approaches are based on encapsulating data and/or computation in independent tasks and scheduling them dynamically based on their dependency graph. This provides significant benefits in terms of extracting concurrency, tolerating operating system (OS) jitter, and utilizing heterogeneous resources such as accelerators. Unfortunately, it also makes understanding the behavior of applications written in these runtimes more challenging than those written in process-centric models such as MPI. The overdecomposition into tasks increases the overall complexity and the scheduling of tasks on processing elements is hard to track. Furthermore, the runtime adds non-trivial processing often hidden from the users.

Task-based runtimes (TBRs) result in fine-grained program execution based on distinct tasks and explicit dependencies. Tracing provides detailed records of all events of interest, e.g., function calls, message sends or receives, or in the case of TBRs – tasks. Analysts can study the exact sequence of events from an execution. However, as applications become more complex and grow in numbers of processors, tasks, and events, understanding these traces becomes challenging. It is hard to relate the observed order of events to the original algorithms. This causes a disconnect between the developers' mental picture of an application and the information encoded in the trace. Recently, Isaacs et al. showed that a suitable reordering of events in MPI applications can capture the missing context and is often better aligned with the developers' intended logical structure [13]. This new order leads to more advanced analysis techniques such as a definition of lateness for events. Additionally, it enables the reorganization of traditional parallel timelines into more intuitive visualizations [12] as shown in Figure 1.

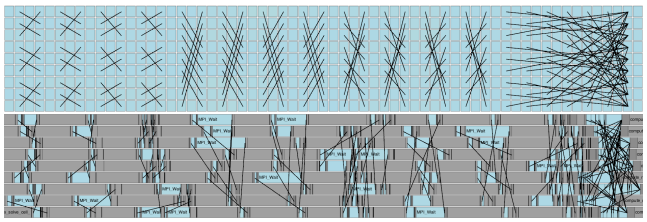


Figure 1: Logical structure (top) and physical time (bottom) in a 9-process NAS BT [3] trace.

Task-based models, however, present new challenges in determining this developer-intended organization from recorded traces and require substantially new techniques because (1) the ratio of tasks to processors is no longer 1:1 per phase; (2) logically linked tasks may now migrate across processors; (3) the non-deterministic nature of scheduling work does not necessarily permit a happened-before relationship between events on those processors and may obscure patterns of dependencies; (4) some dependencies between events may be internal to the programming models' runtime and not available in the recorded trace data; and (5) traditional metrics assume tasks are statically scheduled to hardware units and use the implicit dependencies to compute metrics like lateness. In TBRs these dependencies are no longer valid as tasks could be reordered and are thus not useful to determine execution inefficiencies.

We address some of the challenges mentioned above and present a novel approach to reorder and analyze trace data from programs running on top of Charm++, a popular asynchronous, message-driven TBR. We analyze dependencies in Charm++ event traces and

SC '15, November 15–20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807634

order events where some dependencies may not be recorded. The new algorithm addresses non-deterministic behavior by aggressively reordering events based on an idealized forward replay within each phase. This reordering approach results in an improved ordering for message-passing programs as well. We define new metrics using our computed ordering, such as *differential duration* and *imbalance* to capture information about execution performance. Our major contributions are:

- An approach to ordering events in traces of Charm++ applications that emphasizes logical structure familiar to developers;
- An algorithm for reordering operations in time, applicable to several programming models;
- Metrics calculated over the new event organization to aid in the performance analysis of these traces;
- Case studies demonstrating that our algorithm results in meaningful trace organization.

We modify the tracing capabilities of Charm++ to record the necessary data and apply our new algorithm. Throughout our explanation, we use Jacobi 2D, a simple Charm++ program that computes heat distribution via Jacobi iteration. We demonstrate the ability to find structure and highlight events of interest in two proxy applications, LULESH [1] and LASSEN [22]. Finally, we discuss how our algorithm applies to other TBRs.

2. TASK-BASED RUNTIME SYSTEMS

The terms *task* and *process* both denote independent units of execution. Tasks are execution units that may have a lifetime shorter than that of the entire program, can share processors, and may be migrated between processors. In contrast, we assume a process exists for the entire program execution, has exclusive use of its own processor, and does not migrate.

A *task-based programming model* is an execution model of parallel computation where the fundamental unit of execution is a *task*, which is scheduled based on its *dependencies* on other tasks. A *task-based runtime* (TBR) system manages the creation, processor assignment, communication, and migration of tasks. The runtime may also implement advanced constructs in the task-based model, such as load balancing and collective communication. An advantage of TBRs is that the units of execution are finer-grained than the processes of traditional models. Consequently, there are typically more tasks than processors. This grants the runtime the flexibility to migrate work in order to use the available resources more efficiently. This can lead to better performance as well as tolerance to OS jitter and hardware faults without requiring developers to write load balancing or checkpointing schemes.

Tracing Dependencies and Developer Intent. A central benefit to recording trace data rather than aggregated profiles is the ability to track dependencies. There are generally three types of dependencies: (1) *control* dependencies determine what code to run next, which can manifest explicitly through messages, or implicitly through shared memory or the ordering of observed events within a task; (2) *data* dependencies may be tied to explicit control dependencies (e.g., a message carrying both data and directives for control flow) or manifest independently (e.g., a task waiting on a message carrying only data); and (3) *resource* dependencies where a task requires a shared resource such as a processor or semaphore.

The dependencies described above are essential in determining meaningful orderings of events in a trace. As our logical structure is meant to evoke the developers' intended organization, we are interested in the control dependencies. In process-centric message-passing programs, control dependencies are explicitly bundled with messages and largely guaranteed through the ordering of events in each process. While task-based models may preserve the explicit manifestations of control dependencies, they can make the implicit dependencies more difficult to infer and introduce resource dependencies as multiple tasks can require the same processor.

Programs are generally written to perform some computation on a domain (e.g., physical space or matrices). Developers must decompose this domain to compute on it in parallel. They must also manage relationships among the pieces or sub-domains. Therefore, data dependencies between the sub-domains are central to the developers' organization and thus logical structure. In process-based models, these data dependencies are often implicit as each sub-domain 'lives' on a process. For TBRs, we use data dependencies to group the tasks logically – transforming process timelines to sub-domain timelines.

2.1 The Charm++ Runtime System

The Charm++ programming model and runtime system represent a specific flavor of TBRs and embody the principles of over-decomposition, adaptive overlap of computation and communication, and asynchronous message-driven scheduling and execution of tasks [14, 15]. In Charm++, a *chare* is a C++ object that is promoted to a parallel object that can be migrated between processors. Chares are responsible for encapsulating data, performing tasks, or both. Chares can either be associated with the application-level code or the Charm++ runtime. For our purposes, we group application-level tasks by their parent chares, but group all runtime tasks by their parent process. The application-level grouping observes the data dependencies between tasks as encapsulated by the chare.

Tasks are defined by *entry method*s of a chare. These entry methods are scheduled for execution via remote method invocations, whose parameters are marshalled and translated into messages by the runtime. The runtime routes each message to the processor where the corresponding chare (and hence the destination task) lives. Per-processor queues of these messages are maintained by the runtime as well. A chare is awakened when the runtime scheduler dequeues its corresponding message and the task gets scheduled. Entry methods or tasks are guaranteed to be executed without interruption. When the method completes on a processor, the runtime selects the next message from that processor's queue, thereby awakening the message's associated chare and method to begin execution. Therefore, control dependencies between tasks are explicitly bundled with these messages sent between chares.

Chares can be grouped into indexed collections called *chare array*s over which messages and other operations can be executed. For example, one chare can invoke an entry method on every chare in an array through a single call (similar to a broadcast operation in MPI).

Structured Dagger. Structured Dagger (SDAG) is an alternative style of specifying control dependencies in Charm++ programs. Developers can specify the dependencies between blocks of C++ code that are denoted with the `serial` keyword. Each `serial` is treated as an entry method. The `serial` executes when the dependencies specified by the developer have been met. A `serial` can be dependent on the C++ control flow in the SDAG code or on the arrival of specific messages by using a `when` clause, e.g., `when entry_method() serial { /* do something */ }`.

We must infer some happened-before relationships since SDAG code is implemented by the runtime and not directly traced. The Charm++ compiler creates generic entry methods for each `serial` code section. Each such method is named in a standard fashion which includes a number related to its parsing order. Therefore, `serial` methods close in numbering may be close in control flow

order. For each chare, if we observe an event of `serial` n followed by an event of `serial` $n+1$ in true time, we infer that the first event happened-before the second. The `serial` following a `when` clause is guaranteed to occur immediately after the dependencies specified by that `when` clause have been fulfilled. If we find an entry method that occurs right before a `serial`, we absorb it into that `serial`'s entry method for our ordering algorithm.

Tracing Framework in Charm++. The Charm++ runtime comes equipped with a tracing framework that can record events for application chares. The framework records the begin and end times of each entry method executed on each process as well as messaging events. A visualization tool called Projections can read these traces and produce a process timeline visualization [16, 21, 20].

3. ORDERING TRACE EVENTS

We use the term *logical structure* to denote an exact ordering of events reflecting patterns of parallel dependency behavior as designed by the application's developers. Generally, programming such applications requires breaking the problem into smaller pieces, or phases, some of which require parallel interaction. The logical structure separates these phases which may be interleaved in physical time. Similarly, within each phase, the events are ordered by their dependencies into discrete *logical steps* rather than their true physical time order and displacement. This matches how the events were programmed: with logical ordering rather than known timing information guiding the design.

Our ordering algorithm has two main stages: phase-finding and ordering. In the phase-finding stage (Section 3.1), we partition all of the events into phases. We begin with very small partitions and the relationships between them. These form a directed acyclic graph (DAG). We successively merge smaller partitions into larger ones. The partitions at the end of this stage are our phases and the dependencies among their tasks form a DAG of phases.

In the ordering stage (Section 3.2), we consider each previously computed phase individually. First, we determine an order of the events for each chare in the phase. Then, we use that order and the relationships of events associated with different chares to determine a phase-wide order. In this order, each event is assigned a local logical step. Finally, we use the ordering implied by the phase DAG to offset the local steps into global ones. This completes our logical structure as each event is assigned an exact position in logical time.

3.1 Partitioning into Dependency Phases

We begin by partitioning events into fine-grained phases. These phases should capture related interactions among chares. To find these phases, we begin with the smallest partitions that must belong in the same phase and any relationships among those partitions. We consider the partitions as vertices and the inter-partition relationships as directed edges in a partition graph.

We merge partitions based on several heuristics described below. As we merge, we maintain the partition graph relations – if partitions P and Q merge, the (non-Q) successors of P become the successors of the newly merged PQ. At the end of our merging process, we consider the final partitions to be our phases. The graph over them defines the ordering of phases, and thus must be directed and acyclic, otherwise no order is possible. Should a cycle exist in the partition graph, we infer these partitions are part of the same phase – their constituent events must be ordered at a finer granularity. Thus, we merge partitions that form strongly connected components. We perform this *cycle merge* after applying any heuristic that may create cycles. By doing this, we ensure each step of our partitioning algorithm starts and ends with a DAG over partitions.

The phase DAG is used to create a global ordering of the events. This implies a global ordering of events per chare. As such, the phase DAG must have additional properties that disallow ambiguity in the order of events associated with each chare. We discuss the requirements for this property in Section 3.1.4.

Finally, we refer to partitions that only contain dependencies between application chares as *application partitions*. Those with dependencies between application and runtime chares or purely between runtime chares are *runtime partitions*. Throughout our phase-finding scheme, we only merge application and runtime partitions during cycle merges. We perform this step to separate the developers' view of the program (application phases) from more detailed information about the runtime system (runtime phases). Cycles in the partition graph indicate we cannot disambiguate them; thus, they must be merged.

3.1.1 Initial Partitions

The initial trace data consists of enter and exit times for each entry method and entry method invocation calls ('sends'). We call any set of events that is guaranteed to execute serially within a single unit of execution a *serial block*. In task-based models, the serial blocks are intended to be fine-grained to aid in scheduling work. As such, we infer that dependency events within a serial block belong to the same larger phase. We use this knowledge to group events into initial partitions. However, we may break a serial block into multiple initial partitions to observe the separation of application and runtime phases. Dependencies between application and runtime or vice versa subdivide the serial block as shown in Figure 2. Following this step, we have a graph of initial partitions

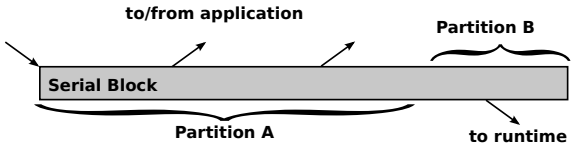


Figure 2: Initial partitions can contain sequences of dependency events within a serial block. Partition boundaries are made when the dependencies cross application-runtime boundaries.

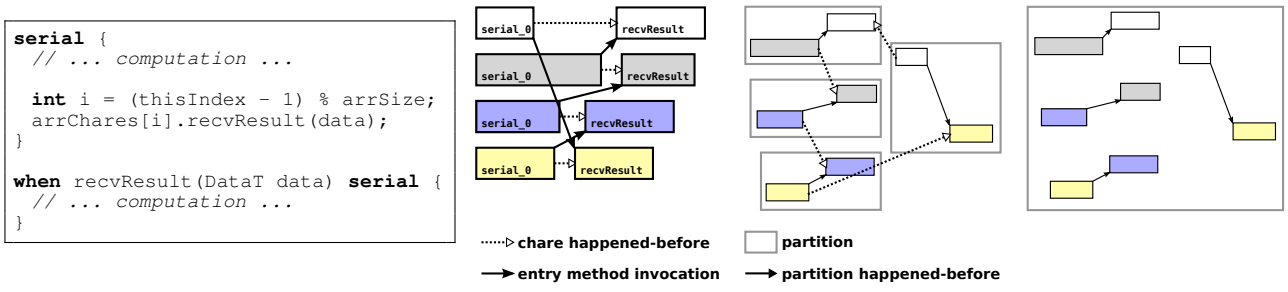
connected by dependency edges. The edges consist of (1) remote invocations, (2) happened-before relationships between application and runtime partitions of the same serial block, and (3) happened-before edges inferred from Structured Dagger code (Section 2.1). At this stage each partition contains events from a single chare only.

3.1.2 Inter-chare Dependency Merge

We split the trace into phases to represent parallel actions defined by the developers. Therefore, partitions containing the matching end points of a remote method invocation belong in the same phase. We merge partitions along these dependencies. The resulting partitions may span multiple chares. Merging by dependencies can result in cycles in the partition graph. We conduct a cycle merge immediately afterwards to maintain the DAG ordering. Algorithm 1 outlines this process. A simple example is shown in Figure 3.

3.1.3 Serial Block Repair

There may be partitions that *would* have been merged in the previous dependency and cycle merges had we not split the serial blocks into application- and runtime-related portions. We now restore those merges. The partition DAG includes edges from the happened-before relationships within the broken serial blocks. For each partition, we merge all directly succeeding partitions that share



(a) Program (b) Trace (c) Dependency merge (d) Cycle merge

Figure 3: (a) Each chare in the array `arrChares` invokes `recvResult` on its neighbor. (b) In the trace, we know of these invocations as well the dependency between the first `serial` and the `when`. (c) We merge blocks so that matching ends of the invocation are in the same partition. These partitions inherit the dependencies of their contained blocks. (d) In this case, those dependencies form a cycle, so they are merged into a single partition.

Algorithm 1: Merging Partitions Across Dependencies

```
dependency_merge (trace, partitions);
    for message in trace do
        send = get_source(message);
        recv = get_sink(message);
        p = get_partition(send);
        q = get_partition(recv);
        if p ≠ q then
            schedule_merge(p, q);
        end
    end
    partitions = merge_scheduled(partitions);
    partitions = cycle_merge(partitions);
```

Algorithm 2: Restoring Merges Across Split Blocks

```
repair_merge (trace, partitions);
    for event in trace do
        p = get_partition(event);
        prev = get_serial_happened-before(event);
        q = get_partition(prev);
        if p ≠ q ∧ is_runtime(p) = is_runtime(q) then
            merge(p, q);
        end
    end
    partitions = cycle_merge(partitions);
```

the same type (entry method) of split serial block as shown in Figure 4. These merges may also create cycles, so we conduct another cycle merge; see Algorithm 2.

A similar merge can be done for neighboring `serials`. Suppose a set of chares participate in `serial` n in a single phase. If those chares then immediately participate in `serial` $n+1$ in several partitions, this may indicate the control flow of one multi-chare group to another, indicating the latter `serial` partitions should be merged.

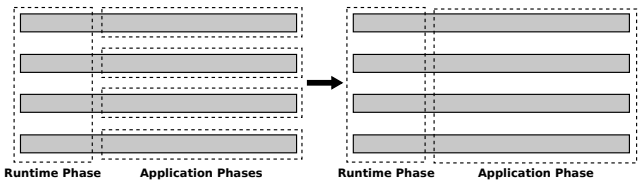


Figure 4: After merging phases due to dependencies and cycles, we merge phases that *would* have been merged had we not broken the initial partitions at runtime-application boundaries.

3.1.4 *Enforcing Orderability and Inferring Missing Dependencies*

In Section 3.2, we will use the phase DAG to establish a total order of events for each chare. To ensure such an ordering is possible, we must enforce extra properties on the DAG. We refer to the set of partitions at the same maximum distance from the beginning of the partition graph as a *leap*. Using this definition, we require the following two properties of the partition DAG:

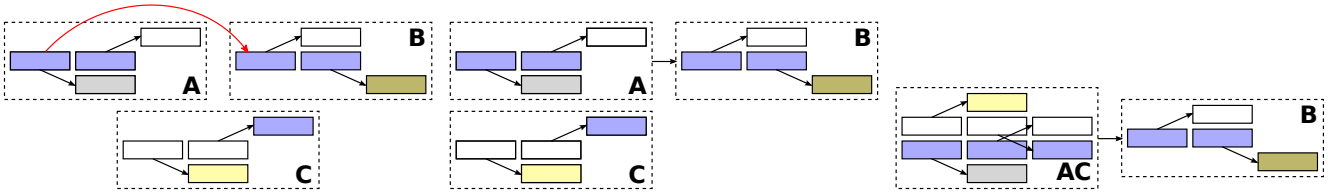
1. The partitions in each leap do not overlap in chares.
2. Each partition has successors that span all of its chares except the chares that do not appear in any successor leaps.

Together, these properties ensure that there is a single path through the phase DAG for each chare. This in turn guarantees that no two events of the same chare can be assigned the same logical step.

We may not have enough control dependencies necessary to properly order all the partitions. For example, control decisions made through the runtime may not have been traced. This lack of dependencies can result in a disconnected partition DAG. In some cases the DAG might still meet the above criteria, but often it leads to leaps with overlapping partitions. We attempt to infer the missing dependencies by examining the physical time order of events. If we still cannot meet property (1), we assume that the overlapping partitions are part of the same phase and merge them. At this point, the partitions are finalized as phases. However, we still must order application and runtime partitions with overlapping chares. We use a more liberal physical time comparison to do so. While this meets property (1), branching in the phase DAG can violate property (2), so we add edges based on leap structure. We explain these operations below.

Inferring Dependencies from Partition Sources. The initial events in each partition must be sources (sends) as matching send and receive events were merged in Section 3.1.2. These events start the phase and have the fewest dependencies. We infer that the ordering between these partition-starting events indicates an ordering between their partitions. We compare the physical time of these partition-starting sources per chare and use the result to add happened-before relationships (Figure 5). When necessary we merge cycles created by these added dependencies; see Algorithm 3.

Merging Concurrent, Overlapping Phases. The additional de-



(a) Physical time ordering (red) of initial source events of the blue chare.

(b) A → B is created and leaps are calculated.

(c) A and C are merged as they are in the same leap.

Figure 5: We infer happened-before relationships using physical time between initial source events within partitions. We merge partitions at the same leap with overlapping chares. Event color indicates chare.

Algorithm 3: Inferring Dependencies

```
infer_dependencies (partitions);
    for p in partitions do
        for event in p do
            if is_source(event) then
                next = find_next_source(event);
                q = get_partition(next);
                if p ≠ q then
                    add_edge(p, q);
                end
            end
        end
    end
    partitions = cycle_merge(partitions);
```

pendencies of the previous step may not solve the chare-overlap problem. Some chares may be represented by only receive events in a partition and thus no dependencies will be added. The problem of not having enough dependencies to order partitions is the complement of the cycle problem where we have too many dependencies. We again infer that the inability to order suggests we should merge into a larger phase and handle the relationships in the ordering stage instead (Figure 5c and Algorithm 4). This is the final merge in our phase-finding algorithm. The resulting partitions are our phases.

Algorithm 4: Merging By Leap

```
infer_merge (partitions);
    all_leaps = compute_leaps(partitions);
    k = 0;
    while k < |all_leaps| do
        leap = all_leaps[k];
        for p, q in partitions(leap) do
            if chares(p) ∩ chares(q) then
                schedule_merge(p, q);
            end
        end
        partitions = merge_scheduled(partitions);
        k = k +1;
    end
```

Enforcing DAG Properties. We maintained the application/runtime partition division during the previous (non-cycle) merges. This means the phase DAG may still have some chares that appear in two phases at the same leap – an application phase and a runtime phase. We impose an order between these two phases again based on the physical time of their initial sources. If there is no chare overlap between initial source events, we compare across chares on a per-processor basis. By adding these phase-dependencies, we ensure our first DAG property.

Finally, we add dependencies to ensure property (2). We preserve the leap structure from the previous steps and thus only add relationships that maintain this leap order. We determine whether the direct successors of each phase contain all chares in that phase. If there are missing chares, we find the next leap containing the missing chares. We add happened-before relationships between the original phase and all phases in that found leap that contain the missing chares. If no such leap exists, we have verified the phase is in the largest leap containing those missing chares, meeting property (2). Algorithm 5 outlines this process. Figure 6 demonstrates the necessity of this step and shows the added relationship.

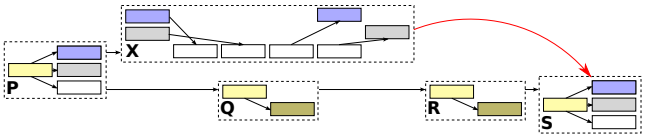


Figure 6: Without a relationship between phases X and S, both would have a gray chare event at the eighth step. To prevent this, we create a happened-before relationship from X to S.

Algorithm 5: Enforcing DAG Properties

```
enforce_chare_paths (partitions, all_leaps);
    last_map = ∅;
    k = maximum_leap;
    while k >= 0 do
        leap_k = all_leaps[k];
        seen_chares = ∅;
        for p in partitions(leap_k) do
            p_chares = chares(p);
            seen_chares = seen_chares ∪ p_chares;
            missing_chares = p_chares - chares(children(p));
            found_leaps = get_leaps(last_map, missing_chares);
            sort(found_leaps);
            for leap in found_leaps do
                found_chares = ∅;
                for q in partitions(leap) do
                    overlap = missing_chares ∩ chares(q);
                    if |overlap| > 0 then
                        add_edge(p, q);
                        found_chares += overlap;
                    end
                end
                missing_chares = missing_chares − found_chares;
            end
        end
        for chare in seen_chares do
            last_map[chare] = k;
        end
        k = k −1;
    end
```

3.2 Step Assignment

Within each phase, we assign local logical steps to each event. Once the per-chare order is set, step assignment is straightforward: we apply the happened-before dependencies between tasks, enforcing that a receive is at least one step after its matching send. The initial sources in each phase have a local step of 0. All other events have a local step of one over the maximum of the events that happened-before them – the prior event along the chare or their matching send when they are a receive. Once the local steps have been determined, we offset them by the maximum step of their phase DAG predecessors to set the global steps.

3.2.1 Reordering of Operations

The most direct ordering policy for events of each chare is by physical time. We allow this option, but in terms of representing the developers' understanding one can often do better by reordering. The physical time order is the result of non-deterministic factors, affected by imbalance in computation, travel time over the network, and queuing policy of the runtime. We reorder to show a structure of dependencies unaffected by these concerns.

The order of events within a serial block is determined explicitly by the developer, so we focus on ordering these blocks. We assign each event a clock value w along its chare. The absolute value of w is immaterial; it is only the value of w with respect to the other blocks of the chare that matters. The initial sends in each phase are assigned a w value of 0. The subsequent sends increment until the end of their serial block. The receives matching these sends are assigned the value $w_{send} + 1$. Sends occurring after a receive count up from the w of that receive.

We use w to order the serial blocks of each chare. Events within a serial block maintain their order. First, the serial blocks are ordered by the w value of their initial event. Then ties are broken by the chare ID of the event that invoked the serial block. The first comparison imparts logical order from the start of the phase. The second imposes an ordering on the serial blocks that logically happened at the same time. Figure 7 shows an example of this comparison in practice. If these two steps are not sufficient to order the serial blocks, we go back a step, comparing the source blocks. The ordering may not be ideal in all cases and pathological examples can be constructed. However, prior knowledge of the simulation could improve the ordering. For example, if the chares represent neighbors in 3D space, an ordering that takes this data topology into account will likely be more intuitive than tie-breaking by chare ID.

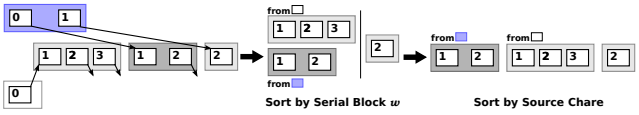
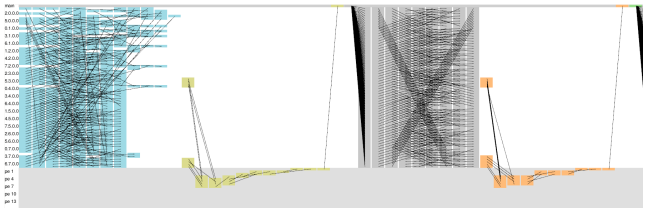
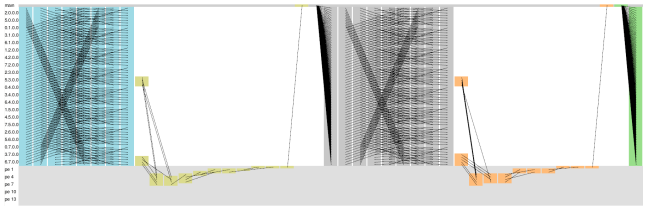


Figure 7: The blue and white chares lead to sink events on the gray chare with the same w. In the first ordering comparison, these two events cannot be ordered, so a second comparison orders by the chare of the matching source event, placing the event from the blue before that of the white.

Figure 8 demonstrates reordering for the first two iterations of Jacobi 2D. Events are colored by their phase membership. There is an alternating pattern of application phase and runtime phase. The runtime chares are drawn on the bottom with a gray background. Without reordering, the first application phase is not compact or recognizable. After reordering, both the first and second application phases reveal a shared a communication pattern that is not apparent in either of the non-reordered versions.



(a) Physical time order of task events



(b) Reordered task events

Figure 8: Two iterations of Jacobi 2D with 64 chares on 8 processors. Colors indicate phase. Steps were assigned with events (a) in recorded order; and (b) reordered.

Reordering for message-passing models. We extend the reordering approach to process-based message-passing programs. In this model, each serial block contains a single send or receive event. The assignment of w remains the same for the initial sends and the receives. Unlike in the task-based model where each source only has one guaranteed prior sink (the one at the beginning of its serial block), these sends may have multiple receives that happened-before them. The sends are recorded from where they are called by the developer. We cannot infer any dependency on the ordering of the receives on the process. Therefore, we do not allow the sends to be reordered. The later sends are assigned the value

$$w_{send} = 1 + \max\{w_{receive} | recieve \rightarrow send\}$$

This value ensures the send maintains its position after the receives, but that the receives may be reordered. Furthermore, receives that come after the send in physical time may be reordered to come before it as shown in Figure 9. However, this scheme cannot move a receive from before to after the send as we have no dependency information indicating when that should occur. For example, if the receive marked 6 in the figure were meant to go after the send, we could not discern that. We must assign the send the value of 7.

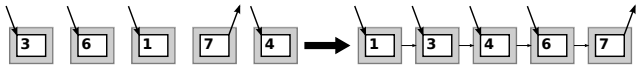
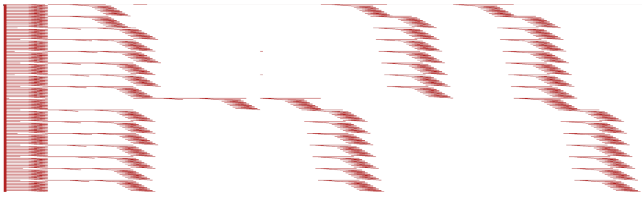
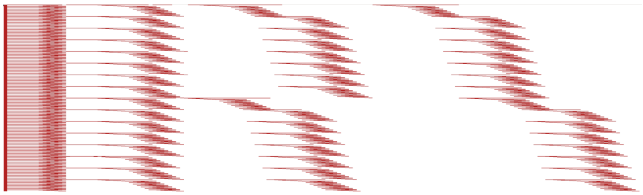


Figure 9: The send is assigned the value 7 since the maximum value of the receives before it is 6. The receive after it is moved before it based on value.

Figure 10 shows two organizations of a trace of an early version of a merge tree algorithm [18] written in MPI and executed on 1,024 processes. The top image shows the results of the message-passing trace organization of Isaacs et al. [13] using our stepping without reordering. As Isaacs et al. originally observed, there are non-ideal alignments in the early steps. Data-dependent load imbalance causes some groups of processes in the merge tree algorithm to send their second phase messages before other groups have finished their first. With reordering, those early steps reveal more parallel structure.



(a) Physical time order forces some events to the right.



(b) Reordering recovers parallel structure of initial steps.

Figure 10: Logical structures of a traced 1,024-process MPI merge tree algorithm. Irregular receive order forces some events to be stepped much later than their peers. Reordering restores regularity.

3.3 Complexity

The first stage of our algorithm involves successive operations performed on the partition graph $G_P(V, E)$. Initially the nodes of this graph are messaging events grouped by their calling entry method. Thus, $|V|$ is bounded by the number of messaging events. Most events have at most 3 edges – their matching send or receive, and then the happened-before and after events in the entry method should they exist. Broadcasts events have more edges, but these edges will be merged away during the dependency merge (Algorithm 1). As the merges find short phases, the graph grows large in leaps with relatively few edges. Therefore, $O(V + E)$ operations such as the cycle merges are closer to linear in $|V|$. Event-centric operations such as the dependency merge, block-repair merge, and dependency inference (Algorithms 1, 2 and 3) iterate over each event and its message partners and are thus close to linear in $|events|$ as well. The leap merge (Algorithm 4) can aggregate linearly in $|V|$. Algorithm 5, the addition of phase dependencies to prevent overlapping steps, works backwards through leaps to keep track of where chares are represented so searching need not be repeated. The chares must be checked, but this is limited by the events in the phase. There are few phases at each leap and few direct dependencies between phases, keeping the number which must be checked low. Thus, in most cases the partitioning stage is near linear.

The ordering stage must replay each event and sort along each chare. If all events were partitioned into a single phase, the ordering would be $O(events \log events)$. However, most traces will have many relatively short phases. As each phase is handled individually, this stage could be parallelized.

Memory is the more limiting aspect. We suspect an out-of-core version could be developed to leverage the graph sparsity and loose relation to physical time.

In this work, we focus on the transformations required by the algorithm rather than the efficiency, leaving the suggested enhancements to future work. We demonstrate the efficiency of the algorithm in its current form in Section 6.1.

3.4 Comparison to Message Passing Models

The reordering approach presented in this section shares some obvious similarities with the original scheme for message-passing models proposed in Isaacs et al. [13]. However, issues such as the lack of dependency information and over-decomposition require substantially different techniques. We briefly compare the two approaches.

Message-passing models can assume that per-process events in physical time indicate a control flow order. Isaacs et al. assume happened-before dependencies based on that order. While this assumption is not always true, e.g., Figure 10, it does provide a wealth of additional dependencies to inform the partitioning stage.

Both algorithms create a logical structure of events grouped into phases through a partitioning and stepping stage. Isaacs et al. use a single inter-process communication event as the initial partition, while we are able to group more events when we have information about serial blocks. Both algorithms do an inter-timeline dependency merge and must merge cycles to create order. However, because of strict time-order within-process dependencies, the message-passing algorithm never has overlapping processes at the same leap or lack of connections between leaps. Thus, it does not have to consider the DAG properties we enforce in Section 3.1.4.

The message-passing algorithm does not reorder events while assigning steps. Furthermore, in contrast to our algorithm, it uses a more complicated step assignment that prioritizes aligning its send events, discouraging overlapping of sends and receives. Our algorithm does not do this as we expect more overlap of source and sink dependencies in asynchronous models. In general, we prioritize the sink events as, unlike in message-passing, the sinks rather than the sources determine the structure by initiating serial blocks.

4. PERFORMANCE METRICS

Traditional metrics of lateness or delay [13], which measure the difference in completion time among operations at the same logical timestep, show differences from the ideal in bulk synchronous applications. However, they are not suitable for asynchronous task-based applications as the order of tasks is potentially non-deterministic. In these models, we do not expect all events occurring at the same step logically to be executed at the same time, nor do we necessarily consider that ideal. Instead, we consider the ideal to be efficient use of processors. One indicator that processors are not being used efficiently is idling. We create a metric to show where idling is being experienced. We create two more metrics to further examine how non-idle time was spent with respect to the organization of events. We calculate metrics over a variety levels: the individual events, serial sub-blocks, serial blocks, and phases. Our organization allows us to identify these levels and make localized comparisons taking into account the program structure.

Idle Experienced. We highlight idling in the organization of events with the *idle experienced* metric. This metric is non-zero at events that are preceded by idle time on their processor. The event occurring directly after recorded idle time has idle experienced of the length corresponding to the preceding idle. However, in calculating idle experienced, we do not stop at that event. We continue forward in physical time along the processor to the first event of each subsequent serial block. If that event was waiting on a dependency that started before the end of the idle time, that event is also assigned idle experienced. We stop searching the subsequent events when we find one dependent on an event that happened after the recorded idle time. Figure 11 shows this assignment for an idle span on one process.

Figure 12 shows how some tasks experience idle while waiting for the reduction in Jacobi 2D. In some instances this compares times across processors in which case clock synchronization problems can lead to erroneous results. Applying post-processing algorithms [25, 5] to ameliorate this issue would help. However, as we use this

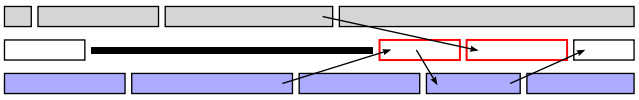


Figure 11: The two red-outlined serial blocks following the idle period (black bar) on the middle process experience the idle as they wait for their dependencies. The next block does not as it is dependent on a block starting after the idle.

metric to determine where we should focus our attention, blocks that would not be assigned idle experienced due to synchronization offsets are unlikely to be of as much interest as their idle time could only be on the order of the clock skew.

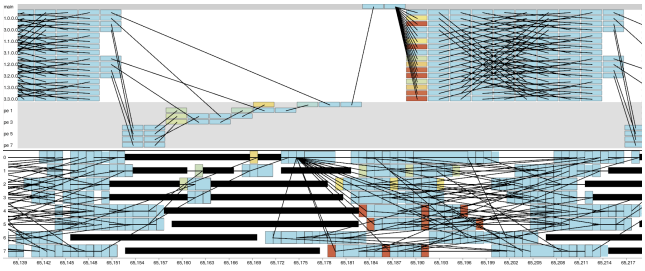


Figure 12: Idle experienced by events in a 16-chare execution of Jacobi 2D in logical and physical time.

Differential Duration. In many cases, computations in the same phase that occur at the same logical timestep are the same action and thus can be assumed to require the same amount of time. We use the dependency events to divide serial blocks into event-delimited units of computation and compare their duration. For each event in a serial block, we define a sub-block to span the time from the previous event in that block to the end of the event. Any leftover duration is assigned to either the event associated with starting the serial block if it was recorded, or the last event in the serial block (Figure 13). Assigning sub-blocks in this fashion allows us to compare durations across the logical steps associated with the events defining each sub-block. The *differential duration* is the excess time spent in the sub-block with respect to the shortest time at that logical step. Figure 15 shows this metric mapped in both the logical time chare view and the physical time processor view. One chare experiences a longer computation block (orange) than its peers.

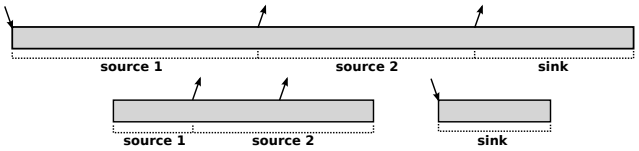


Figure 13: Three examples showing how serial blocks are divided based on the dependency events.

Imbalance. We can use the duration we calculated previously to determine computation imbalance at the phase level. For each processor, we sum the total duration in that phase and calculate the maximum difference between the processors with the most and least work. We can also see the spread of this imbalance by calculating per processor the difference between its total duration and the minimally loaded processor. Figure 14 shows imbalance per processor mapped to each event. The iteration with the long event, shown in Figure 15, has higher imbalance than the one after it. In processor space, the high imbalance is shown in orange for a single processor. In chare space, it is shown on two chare timelines as both are executed on that one imbalanced processor. This metric captures the variation of imbalance across the processors.

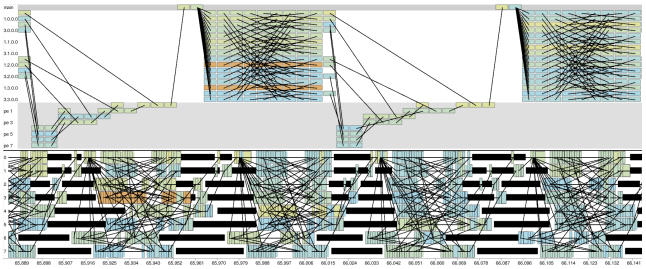


Figure 14: Processor imbalance shown per event for a 16-chare execution of Jacobi 2D. The iteration with the long event from Figure 15 shows greater imbalance both for the chare of that event and the other chare on the processor.

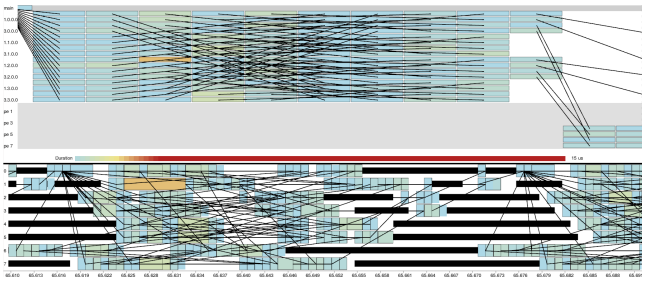


Figure 15: Differential duration on a 16-chare execution of Jacobi 2D. One chare experiences a significantly longer compute time (orange).

5. ADDITIONS TO CHARM++ TRACING

The Charm++ tracing framework traces events as they occur on each process. We require events to be semantically connected to chares and chare arrays. Each application event has an associated chare identifier (ID) and we expanded the format to also include a chare array ID. We recorded more control flow information by tracing common runtime behaviors, though as seen in Section 6, we did not capture all control information.

A commonly used Charm++ operation is a reduction over an array of chares. A reduction is performed by each chare invoking the `contribute` method on a process-level chare of type `CkReductionMgr`. When all the local contributions are gathered on a process, an entry method is executed that is specified in a callback passed to the `contribute` invocation. Previously, only the explicit messages in the reduction were recorded between processors. Hence, the local reduction events on the process between the local chares were not recorded.

Rather than abstracting these events into a single operation that hides the runtime chares, we added the tracing necessary to follow the local reduction on the process. This allows us to examine how the process-level chare, `CkReductionMgr`, is scheduled on the resources shared with the application-level chares. We added the dependency between the application and runtime as a message from one application chare on each `CkReductionMgr` representative and added the missing internal messages necessary to reconstruct the control flow structure.

The overhead of adding process-local reduction events is a small constant cost over the previous tracing overhead. The `contribute`

call can only be made inside an entry method, so there always exists a local event for the method that precedes it. Therefore, adding another short traced event for the local reduction event incurs a low cost that we have found to be negligible in practice.

6. CASE STUDIES

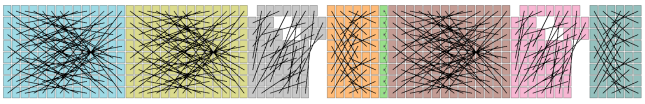
We demonstrate the accuracy of our approach through two proxy applications, LULESH [1] and LASSEN [22], both of which have Charm++ and MPI implementations, allowing us to compare logical structures. We executed these applications on an Infiniband cluster running a Red Hat-derived Linux distribution and the MVAPICH MPI implementation. We collected traces using Score-P [17] for MPI and the native logging for Charm++.

All figures were generated on an Intel Core i7-4770 with 32 GB of RAM using a modified version of Ravel [12]. In particular, we draw recorded idle time as thin black bars similar to the representation in Projections [21]. In the logical structure view, we group all runtime chares on the bottom. The logical structure for MPI traces was computed using the algorithm from [13] without modification.

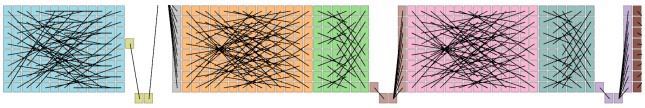
6.1 Inferring Structure in LULESH

LULESH is a proxy application for hydrodynamics simulations. We show that the logical structure we compute for the Charm++ trace corresponds to the logical structure for MPI, implying our structure is meaningful. We demonstrate what happens in our logical structure when we cannot infer dependencies as in Section 3.1.4. We then examine the time needed to compute logical structure for LULESH traces.

Figure 16 shows the logical structure computed from both the MPI and Charm++ traces. The first (blue) phase represents the problem setup. Afterwards, the MPI trace has a repeating pattern of three phases followed by an allreduce (green). After problem setup (blue), the Charm++ trace has a repeating pattern of two phases followed by an allreduce (in brown and later in purple). The two phases have mirrored communication patterns to the first and third repeating phases in MPI.



(a) MPI, 8 processes



(b) Charm++, 8 chares, 2 processors

Figure 16: Logical structure for LULESH traces from (a) MPI and (b) Charm++, colored by phase.

Figure 17 shows the logical structure computed for the Charm++ trace without inferring the dependencies and merging described in Section 3.1.4. The DAG properties are still enforced. The initial phase from Figure 16 is split into several smaller phases that are forced in sequence. Each of the phases before the allreduce is split in two. In both of these cases, the lack of dependency information connecting these smaller phases together results in this incorrect structure.

We apply our logical structure algorithm to several LULESH traces to examine its performance for increasing numbers of iterations and chares. Figure 18 shows a 64-chare LULESH execution at increasing iteration counts. The computation time is directly proportional to the number of iterations – it is not affected negatively by the doubling of phases. Figure 19 shows eight iterations of LULESH at increasing problem size. We hold the chare (sub-domain) size the same, resulting in increasing chare counts. The behavior is inconclusive. The amount of time performing the merge of Section 3.1.4 comprises the bulk of the additional time, likely due to the greater chare counts requiring more comparisons.

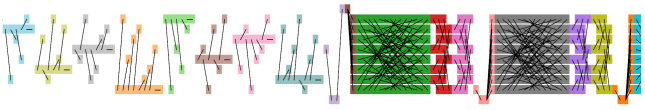


Figure 17: 8-chare LULESH logical structure computed without inferring dependencies as in Section 3.1.4. The initial phase from Figure 16(b) is broken into several phases placed one after another. Each phase before the allreduce is split in two.

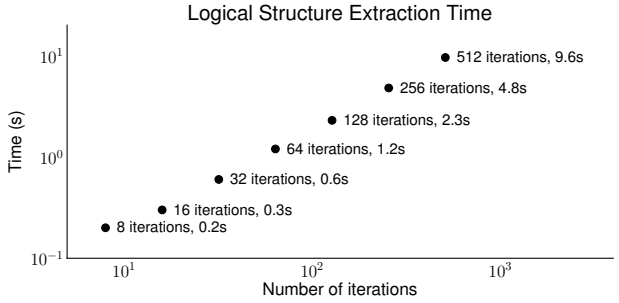


Figure 18: Time to calculate logical structure for a 64-chare LULESH execution at increasing iterations.

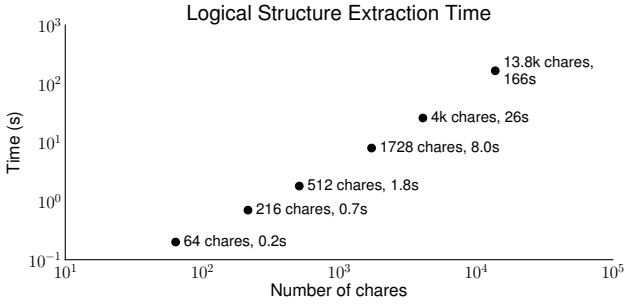
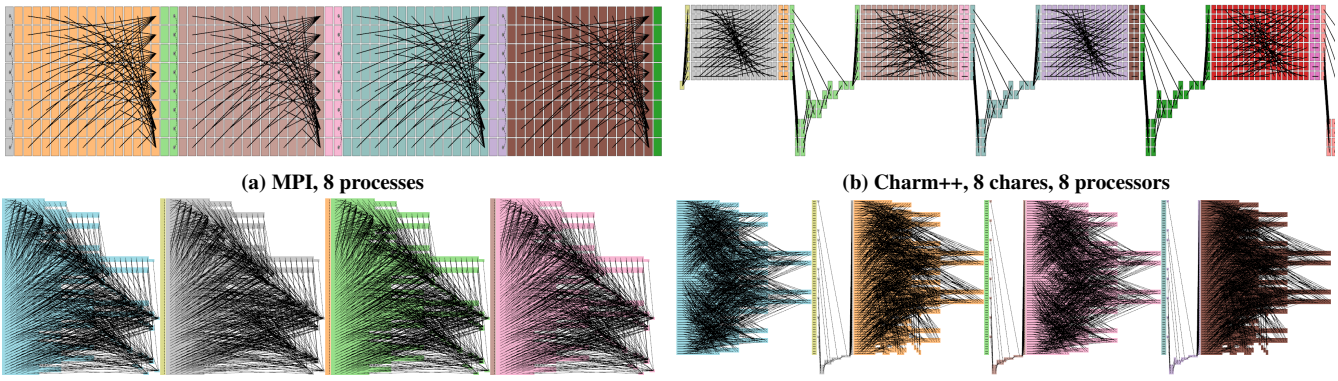


Figure 19: Time to calculate logical structure for eight iterations of LULESH at increasing chare counts.

6.2 Comparing Performance in LASSEN

LASSEN is a proxy application that models the propagation of a wavefront through space. We used the default problem in which space is decomposed as a regular Cartesian grid. We used eight processors for all our runs. The Charm++ implementation was run in both 8 and 64-chare decompositions. The 8-chare Charm++ run is known to perform comparably with the MPI implementation, while the 64-chare one tends to perform better.

Figure 20 shows the structure results for the three traces as well as an MPI trace on 64 processors, all colored by their assigned phase. All four show a repeating pattern of a point-to-point messaging phase to several neighbors followed by an allreduce. In MPI, the allreduce is abstracted into its collective call and thus is shown as two steps (the call and the computation before it). In Charm++, the allreduce is visible as its reduction tree in the runtime chares



(a) MPI, 8 processes

(b) Charm++, 8 chares, 8 processors

(c) MPI, 64 processes

(d) Charm++, 64 chares, 8 processors

Figure 20: Logical structure for LASSEN traces from (a, c) MPI and (b, d) Charm++, colored by phase ID. In each, there is a repeated pattern of a point-to-point phase followed by a collective/runtime phase.

(bottom) and broadcast to application chares. The Charm++ traces show additional two-step phases between the large point-to-point phase and its subsequent allreduce. In this short phase, each chare invokes itself, indicating this is likely a pure control message to move the computation forward. In the Charm++ traces, the structure of the large point-to-point phase alternates. This is not true for the MPI trace. Both are looping through the same alternating data structures to create these messages, but the order of the elements in those data structures may differ by construction.

We observed a large amount of idle experienced every other broadcast (not pictured). To explore why this was occurring, we color by the differential duration metric (Figures 21 and 22). We see a repeating pattern of two events with much greater duration than their peers. The logical structure makes it easy to tell that the long events have the same chare and role each iteration, a conclusion that would require further investigation to reach using a traditional physical time and processor view. These events mark the first sends after two of the main computation events. They are likely so much longer because we are looking at early iterations and thus the wavefront is in a small region apportioned to a single chare.

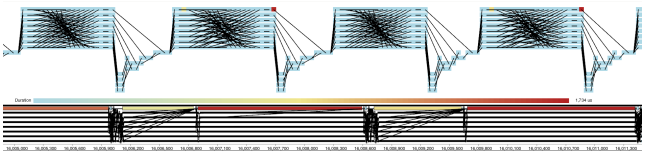


Figure 21: 8-chare LASSEN trace colored by differential duration. In the logical structure (top), a repeated pattern shows the same events of the same chare are associated with higher duration. Physical time (bottom) makes it difficult to discern these events are the same, as there is no indication where it falls in the control flow.

Many iterations later, we still see a repeated pattern of long duration, but for different chares and more of them (Figure 23). This makes sense as the wavefront grows in the domain. We chose different color ranges for the 8 and 64-chare runs for visibility, but the 64-chare run exhibited a maximum differential duration one fourth that of the 8-chare run due to splitting the wavefront into smaller pieces. To check how well these tasks are scheduled, we examined the trace using the imbalance metric (not shown) and saw that while still uneven, the work is spread more equitably in the 64-chare run. This led to less than half as much imbalance overall across processors and thus better performance.

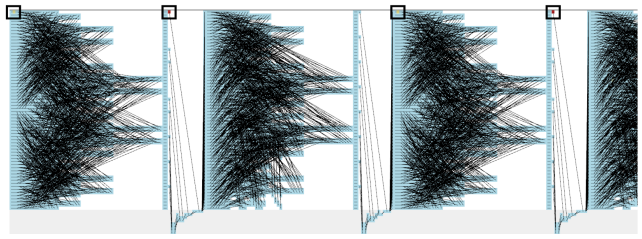
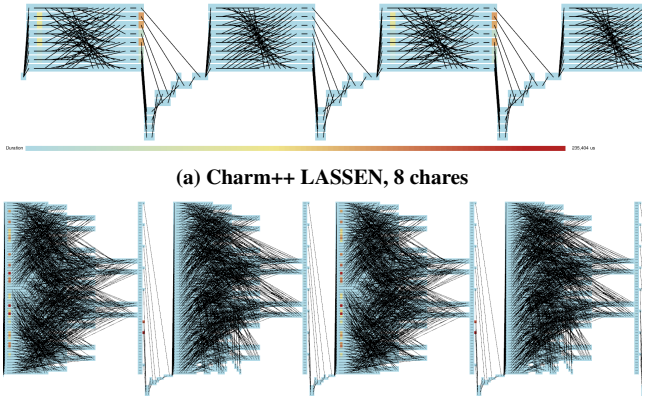


Figure 22: 64-chare LASSEN trace colored by differential duration. Like Figure 21, the logical structure shows repeated long events (boxed) over iterations.



(a) Charm++ LASSEN, 8 chares

(b) Charm++ LASSEN, 64 chares

Figure 23: As the wavefront propagates, more chares share the high differential duration.

7. APPLICABILITY TO OTHER TBRs

We have focused our efforts on Charm++ traces. Through this process, we have observed several principles that we believe will aid in adapting our methods to other task-based models and runtimes. First, the data dependencies of the application domain contribute significantly to the logical structure. Decomposing the domain and then managing the interactions among its pieces are central to designing a parallel program. We represent these data dependencies using timelines of sub-domains (encapsulated by chares in Charm++), rather than processes.

Second, an idealized replay of events within phases can recover parallel patterns previously obscured by asynchrony. Logical struc-

ture is meant to reflect the developers' thinking. The ideal scenario is relatively easier to reason about. We have shown our strategy improves structure for both Charm++ and MPI.

Finally, logical structure is represented as a phase DAG and maintaining this structure can help in finding phases. Too many dependencies or too few indicate that partitions should be merged. In Charm++, we had a dearth of dependencies, so we recognized true time differences between the sources of partitions as a good heuristic for ordering them. However, there are some cases in which this approach does not work. We examine one such case below and prescribe measurements that would aid analysis of any TBR trace.

7.1 Improving TBR Traces

While our case studies show our algorithm successfully finds structure in Charm++ applications, it can produce sub-optimal orderings when not enough data about the control flow is recorded. Parallel tracing utilities usually only record control information when it is explicit, as in the case of a message. In process-based traces, more control information can be automatically inferred from the ordering of events. Analysts with significant knowledge of their chosen parallel model may also be able to infer more about the control flow when examining a trace visually. To enhance analysis for task-based models, more control flow events should be explicitly recorded in the trace, allowing computational approaches such as ours to use this information.

Figure 24 shows the logical structure computed from a 16-chare run of a PDES mini-app. In PDES, each chare of the mustard phase, which runs the simulation, calls the completion detector when finished. The completion detector is represented by the gray phase (directly below the mustard phase). The call to it is not recorded in the trace. Because there is no trace data about the dependency, our algorithm places the mustard and gray phases concurrently rather than in sequence.

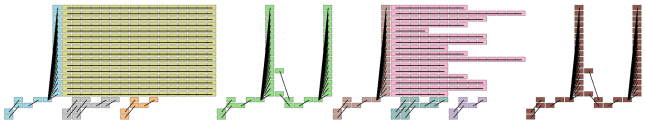


Figure 24: 16-chare, 4-process PDES logical structure, colored by phase. The gray phase is a completion detector called at the end of the mustard phase. This control dependency is not recorded, so there is nothing structurally to prevent both phases from covering the same global steps.

In cases like the above, where much of the control information passes through the runtime, we must choose a level of abstraction with which to represent internal runtime dependencies. For example, MPI collective operations are represented as single calls though the actual use of resources during an MPI collective operation is complex. None of the underlying dependencies implementing it are recorded. The control flow through it at the application level is understood implicitly. Even within a send-receive pair in our Charm++ traces, there are a number of dependencies within the runtime that get invoked. We are not interested in that level of detail, especially as the developer has few options for replacement were a problem to be found.

To aid analysis of task-based parallel traces, the following should be retrievable from the trace without runtime-specific knowledge:

1. The correspondence between events, the data they act upon, and the runtime elements executing them.
2. Control flow between application events that passes through the runtime, either by tracing runtime information or as an abstraction.
3. Sets of events that cannot be divided by runtime scheduling. We have referred to these throughout as serial blocks.

Ideally, a common standard would be created so this data could be obtained from a variety of different models.

8. RELATED WORK

Charm++ has an associated performance analysis and visualization tool, Projections [16, 20, 21], that provides several statistical plots supporting profile analysis for common problem sources such as grain size, as well as a process-level physical timeline visualization. To decrease the number of processes to be examined, Projections has mechanisms for user-guided discovery of outlier processes. We focus on a finer level, that of chares and phases, to analyze metrics calculated over these levels and chare interactions.

Wheeler and Thain [27] extracted parallel structure in the form of an 'event description graph' from traces of shared memory programs. They searched for problematic subgraphs and visualized them with GraphViz [10]. To obtain the parallel structure, they limited their models to those where such a structure was available at runtime, in contrast to the traces we analyze.

Logical time has been used for trace analysis and visualization, with an emphasis on debugging [19, 26]. The use of the term "logical" in those works refers to discretization by happened-before relationships as the events were executed. Our use incorporates the logical mental model of developers and analysts with which to reason about parallel behavior.

Parallel traces are commonly visualized in physical time only by the resources (e.g. processes, threads) on which they were recorded [8, 21, 23, 24, 28]. Blochinger et al. [6] used binned time to create layered node-link diagrams representing thread execution graphs and highlighted potential problems calculated from those graphs on the visualization.

Automated tools like Scalasca [11] analyze event traces and can detect known problematic patterns, such as late arrival of a message, and compute a severity score that is mapped to source code and machine locations. However, Scalasca is currently limited to message-passing and does not support task-based programming models. Moreover, whereas Scalasca provides an aggregate sum of severity scores per process and source-code location, our metrics are mapped onto events in logical time.

9. CONCLUSION AND FUTURE WORK

We have presented an algorithm for transforming Charm++ traces from the physical time in which they are recorded into a logical structure. This logical structure aids developers and analysts in understanding dependency patterns and provides context for aberrant features in the trace. We demonstrated our algorithm can produce meaningful structure even when many control dependencies are not recorded by the trace. Our approach addresses difficulties in understanding non-deterministically scheduled tasks by a heuristic reordering scheme. Applying the reordering concept to message-passing models has also resulted in better representation of parallel structure. Additionally, we mapped metrics descrbing detrimental execution behavior onto the logical structure and showed how this can be used to analyze performance. While we have focused on Charm++ in this work, we expect our organization by data subdomains, constraints on phases, and reordering scheme to apply to other task-based models.

Despite these successes, we are limited in what we can transform by the dearth of control dependency data recorded. We have made a

set of recommendations regarding information that should be traced. As this becomes available, we will need to improve our algorithm. Inferring dependencies may no longer be necessary; instead, different phase detection methods will need to be explored. Further, while this approach aids in visual analysis, new visualization techniques are needed that scale to large numbers of parallel tasks and show lifetime and migration between processors.

Acknowledgments

We thank Laxmikant Kale, Nikhil Jain, Ronak Buch, and Bilge Acun for their expertise and assistance with Charm++ applications.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by the Office of Science, Office of Advanced Scientific Computing Research as well as the Advanced Simulation and Computing (ASC) program. LLNL-CONF-670046.

10. REFERENCES

[1] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[2] Open Community Runtime. Intel Open Source, 01.org/projects/open-community-runtime, 2012.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, and R. A. Fatoohi. The NAS parallel benchmarks. *Int'l J. of Supercomputer Applications*, 5(3):63–73, 1991.

[4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proc. ACM/IEEE Conf. on Supercomputing*, SC '12, pages 66:1–66:11, 2012.

[5] D. Becker, R. Rabenseifner, and F. Wolf. Timestamp synchronization for event traces of large-scale message-passing applications. In *Proc. European Conf. on Recent Advances in PVM and MPI*, PVM/MPI'07, pages 315–325. Springer-Verlag, 2007.

[6] W. Blochinger, M. Kaufmann, and M. Siebenhaller. Visualization aided performance tuning of irregular task-parallel computations. *Information Visualization*, 5(2):81–94, 2006.

[7] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive Cluster Programming with OmpSs. In *Euro-Par 2011 Parallel Processing*, volume 6852 of *Euro-Par'11*, pages 555–566. Springer-Verlag, 2011.

[8] J. C. de Kergommeaux, B. de Oliveira Stein, and B. P.E. Paje, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Comput.*, 26(10):1253–1274, Sept. 2000.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. ACM SIGPLAN 1998 Conf. on Prog. Lang. Design and Implementation*, PLDI '98, pages 212–223, 1998.

[10] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software : Pract. Exper.*, 30(11):1203–1233, 2000.

[11] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurr. Comput.: Pract. Exper.*, 22(6):702–719, Apr. 2010.

[12] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann. Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time. *IEEE Trans. on Vis. and Comp. Graphics, (InfoVis '14)*, 20(12):2349–2358, 2014.

[13] K. E. Isaacs, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. Ordering traces logically to identify lateness in message passing programs. *IEEE Trans. on Parallel and Distrib. Systems*, to appear.

[14] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108, Sept. 1993.

[15] L. V. Kale and A. Bhatele, editors. *Parallel Science and Engineering Applications: The Charm++ Approach*. CRC Press, Oct. 2013.

[16] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar. Scaling applications to massively parallel machines using Projections performance analysis tool. In *Future Generation Comp. Systems Special Issue on: Large-Scale System Perf. Modeling and Analysis*, volume 22, pages 347–358, Feb. 2006.

[17] A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. TschÃijter, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91. Springer Berlin Heidelberg, 2011.

[18] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. *Proc. ACM/IEEE Conf. on Supercomputing*, SC'14. Nov. 2014.

[19] T. J. LeBlanc, J. M. Mellor-Crummey, and R. J. Fowler. Analyzing parallel program executions using multiple views. *J. Parallel Distrib. Comput.*, 9(2):203–217, June 1990.

[20] C. W. Lee. *Techniques in Scalable and Effective Parallel Performance Analysis*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, Dec. 2009.

[21] C. W. Lee, C. Mendes, and L. V. Kalé. Towards Scalable Performance Analysis and Visualization through Data Reduction. In *Int'l Workshop on High-Level Parallel Prog. Models and Supportive Environments*, Apr. 2008.

[22] B. McCandless. Lassen. codesign.llnl.gov/lassen.php, 2013.

[23] 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.

[24] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualize and analyze parallel code. Technical report UPC-CEPBA 95-3, 1995.

[25] R. Rabenseifner. The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters. In *In Proc. EUROMICRO Workshop on Parallel and Distrib. Processing*, PDP, pages 477–484, 1997.

[26] C. Schaubschläger, D. Kranzlmüller, and J. Volkert. Event-based program analysis with DeWiz. In *Proc. Int'l Workshop on Automated Debugging AADEBUG2003*, 2003.

[27] K. B. Wheeler and D. Thain. Visualizing massively multithreaded applications with threadscope. *Concurr. Comput. : Pract. Exper.*, 22(1):45–67, Jan. 2010.

[28] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *HPC Applications*, 13(2):277–288, Fall 1999.