International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 1, pages 82–95, January 2011

Detecting and Using Critical Paths at Runtime in Message Driven Parallel Programs

Isaac Dooley
Parallel Programming Laboratory
University of Illinois at Urbana Champaign
201 N. Goodwin Ave.
Urbana, Illinois 61801, USA

Anshu Arya
Parallel Programming Laboratory
University of Illinois at Urbana Champaign
201 N. Goodwin Ave.
Urbana, Illinois 61801, USA

Laxmikant V. Kalé
Parallel Programming Laboratory
University of Illinois at Urbana Champaign
201 N. Goodwin Ave.
Urbana, Illinois 61801, USA

Received: July 9, 2010
Revised: October 31, 2010
Accepted: December 12, 2010
Communicated by Akihiro Fujiwara

Abstract

Detecting critical paths in traditional message passing parallel programs can be useful for post-mortem performance analysis. This paper presents an efficient online algorithm for detecting critical paths for message-driven parallel programs. Initial implementations of the algorithm have been created in three message-driven parallel languages: *Charm++*, *Charisma*, and *Structured Dagger*. Not only does this work describe a novel implementation of critical path detection for the message-driven programs, but also the resulting critical paths are successfully used as the program runs for automatic performance tuning. The actionable information provided by the critical path is shown to be useful for online performance tuning within the context of the message driven parallel model, whereas it has never been used for online purposes within the traditional message passing model.

Keywords: Critical Path Detection, Message Driven Parallel Languages, Adaptive Runtime Systems

1 Introduction

A critical path is a type of important path through the execution of a parallel program. Specifically, critical paths provide a lower bound on the execution time for a program and identify the activities

in the program that in turn limit the performance of a parallel program. In the past, critical path detection schemes were developed for some typical message passing models such as PVM [7] and MPI [10]. These approaches record a distributed Program Activity Graph (PAG) as a program executes by storing local portions of the PAG on each processor while augmenting each message sent between processors with information about a critical path leading up to the message send. The critical path can be extracted through a backwards traversal of the distributed PAG.

Although researchers have developed methods for detecting critical paths within the message-passing models of parallel computation, they have not previously detected critical paths at runtime within a message-driven execution model of parallel computing. In this work, we implement an efficient critical path detection mechanism inside the Charm++ message-driven distributed object system. The Charm++ programming model has fundamental differences from the more widely used approach of programming at the level of communicating processors. These differences require revisiting and adapting the known algorithms for critical path profiling, but the differences also provide fertile new ground for novel uses of the resulting critical path profiles. This paper describes both the implementation of critical path detection for the message-driven programs and how the resulting critical paths can be successfully used for online automatic performance tuning and for other tasks.

We show that the critical path profiles obtained online by our implementation can be used to automatically tune the performance of a complicated real-world quantum chemistry application, improving its performance by 10.2%.

2 Message Driven Parallel Programs

The most widely used parallel programming model for distributed memory systems is the message passing model which has become standardized in MPI [3]. An alternative model is the message-driven execution model. In this model, the programmer does not write programs explicitly for a set of processors as is done in MPI, but rather the programmer describes the computation as a set of tasks whose computation is driven by messages sent by other tasks. The tasks are mapped onto the computational resources dynamically by the runtime system.

The message driven execution model is a paradigm that has proven to be successful for parallel programming. Scientific simulation codes such as NAMD [1] and OpenAtom [11] are written using this model. The message driven approach could also be called *data driven* because tasks are dynamically scheduled when the prerequisite data (usually in the form of messages) is available. Directed acyclic graphs (DAGs) can be used to describe a program's pattern of computation and communication, with the edges in the graph representing the dependencies between all the computation tasks in the parallel program. Tasks can be scheduled in any order as long as all the dependencies for each task have been satisfied before the task is executed. The parallel runtime system can record the DAG when the tasks in the program are executed.

A dynamic message driven program must therefore include a scheduler responsible for executing tasks once dependencies have been fulfilled. For this paper, we focus on the *Charm++* language [9] and two languages that each extend it: *Charisma* and *Structured Dagger*.

3 Program Activity Graph Terminology

As a parallel message-driven program executes, its execution can be represented by a directed acyclic program activity graph (PAG), composed of tasks and their dependencies. This section defines specifically what a program activity graph represents for message-driven parallel programs. Figure 1 shows a small example PAG composed of 8 tasks that ran on 4 processors.

Task: After its prerequisite dependencies have been fulfilled, a task is executed by a scheduler on a single processor. Each task may send messages that fulfill dependencies for other tasks.

Task Prefix: A task prefix is the portion of a task from its beginning to the point where a message is sent or the task ends. There exist $m + 1$ task prefixes for each task which sends m messages. The weight of each task prefix is the the execution time for the task prefix.

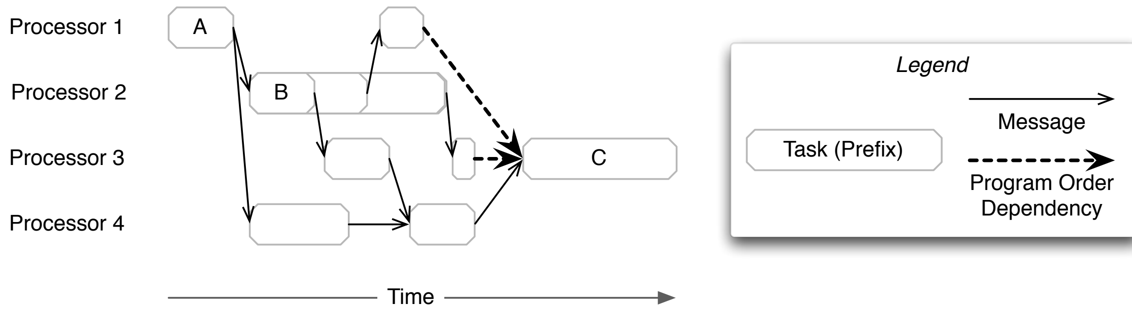


Figure 1: Example timeline view of a parallel program activity graph. Task A represents a task that multicasts a messages to two other processors. Task prefix B can execute once its message from A arrives. Then task prefix B sends a message. Task C can only execute after three preceding tasks have completed.

Initial Task: The beginning of a program is the execution of a single initial task in the parallel runtime system that in turn spawns one or more startup tasks specified in the program.

Terminal Task: A terminal task causes the parallel program to terminate.

Message Edge: A message edge represents a dependency from a sending task prefix to the execution of a task spawned by the message.

Program Order Dependency Edge: Other dependencies due to sequencing requirements in the program are represented by program order dependency edges. In a message-driven system, these sequencing requirements could be implemented as messages.

Program Activity Graph (PAG): A PAG represents an execution of a parallel program, with one vertex for each task prefix and a set of edges comprising the dependencies between the task prefixes. The PAG is therefore a directed acyclic vertex-weighted graph.

Task In-Degree: Each task is executed once a set of messages have arrived and all other order dependencies have been satisfied. The in-degree of a task is the number of incoming message edges and program order edges to the task. The in-degree for all non-initial tasks is ≥ 1.

Task Out-Degree: Each task prefix either results in the sending of a message, or the completion of the task. Each message send is the start of a message edge while the completion of the task results in zero or more program order dependency edges in the program activity graph. The out-degree of each task is thus the number of messages sent by the task, plus the number of program order edges produced by the end of the task. All non-terminal tasks have out-degree ≥ 1.

Path: A path is an alternating sequence of task prefixes and edges in the PAG beginning with some task prefix and ending with another task prefix, where each task prefix is incident to both the edge that precedes it and the edge that follows it in the sequence.

Path Duration: The path duration is the sum of the node weights (task prefix execution durations) along the path. The path duration represents the minimum possible execution time of the path, with unlimited processors and an infinitely fast network.

Critical Path (t):

For each task t in the PAG, its critical path is the path of maximal path duration which ends at t and starts at the initial task.

The path duration of the critical path for a phase of an application represents a lower bound on the execution time for the application phase. It does not include any communication times along the path or the computation times for other unrelated concurrent tasks.

Critical Path Profile (t):

The critical path profile for any task t is the critical path for t augmented with information about the task prefixes comprising the path. The information about each task prefix depends upon the desired use of the critical path profile. It is anticipated that the information will identify the type of the task and the specific instance of the task prefix, in a manner that is meaningful to an online automatic performance tuning system or an offline performance analysis tool.

4 Algorithm for Determining a Critical Path

To determine the critical path for a program execution, we use an approach similar to the approach described in [6]. In both approaches, a distributed PAG is constructed at runtime, but the exact details of what is stored in the table is different. In our approach, each processor maintains a table of all task prefixes that have executed locally. Figure 2 shows an example of a PAG and the local information stored on each processor. To store the necessary information, an entry is added to the local processor's table each time a message is sent or when a task completes. Information that uniquely identifies the sending task prefix is appended to each message. Specifically, each message is augmented with two values, one records the duration of the path that led to the message send and a second uniquely identifies the sender-task-prefix in the sending processor's table. Messages must also contain a field specifying the index of the sending processor, but this field already exists in all Charm++ messages.

The critical path is determined for each task prior to its execution once all incoming dependencies have been satisfied. The path descriptors contained in all incoming messages or dependency edges are merged by selecting the one with maximal duration. All non-maximal incoming paths are ignored. Keeping the maximum incoming path maintains the invariant that each critical path extended along a dependency edge is maximal (critical). Figure 3 shows an example of three incoming message dependencies for a task. The incoming path with maximum cumulative path duration is stored in the processor table to be able to trace back any critical path that includes the task.

When a path is propagated forward via a message or other dependency, the duration of extended path is found by adding the time spent executing the task prefix to the duration of its maximal incoming path. The new value representing the whole path duration is then stored in the newly prepared message.

When the critical path profile is required for a task t, a backward traversal through the distributed PAG is performed. At each step in the traversal, the information about the task prefix is retrieved and then its maximal incoming dependency edge is followed backward.

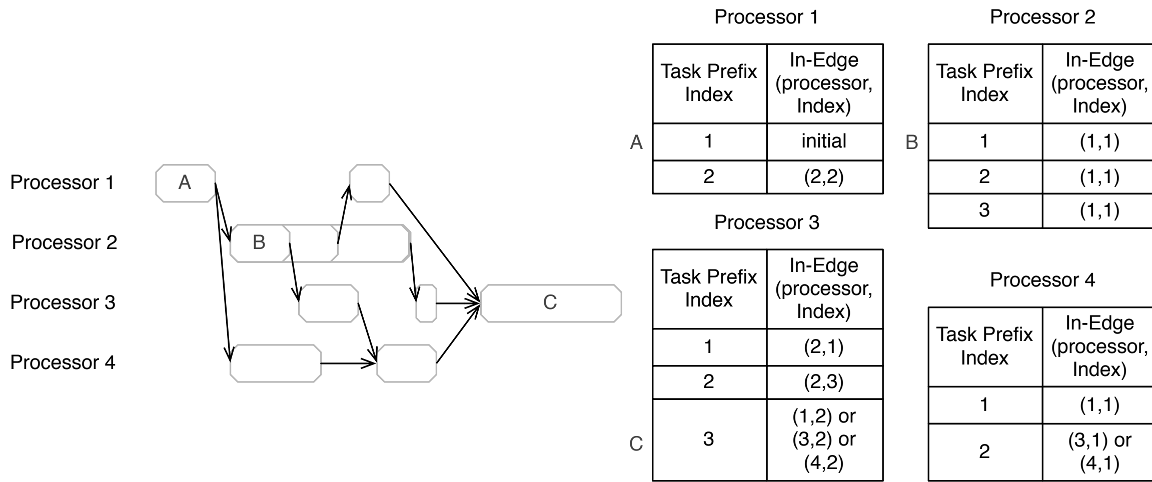
5 Implementations

We have implemented the critical path detection algorithm inside the Charm++ runtime system. This implementation supports standard Charm++ programs as well as those written using the Structured Dagger or Charisma languages.

To implement the critical path profiling algorithm, the following portions of the Charm++ runtime system were modified[1]:

- A new module was created.
- The module's startup routines on each processor create a table to hold the local portion of the PAG.
- The envelope used for all Charm++ messages was expanded to hold the critical path duration and a reference to the sender's table entry.
- The message send functions were modified to fill in the envelope fields with the information about the currently executing task, after creating a new table entry.
- Methods for performing the backwards traversal over the PAG were created within the new module.
- Macros were created to simplify the storing of the maximal known incoming edge and the comparing of it with each new incoming dependency.
- Instances of the macros were added to the Charm++ reduction methods.

[1] This implementation can be found in the publicly available development version of Charm++. The new module is located in the src/ck-cp directory.



Processor 1

	Task Prefix Index	In-Edge (processor, Index)
A	1	initial
	2	(2,2)

Processor 2

	Task Prefix Index	In-Edge (processor, Index)
B	1	(1,1)
	2	(1,1)
	3	(1,1)

Processor 3

	Task Prefix Index	In-Edge (processor, Index)
	1	(2,1)
	2	(2,3)
C	3	(1,2) or (3,2) or (4,2)

Processor 4

Task Prefix Index	In-Edge (processor, Index)
1	(1,1)
2	(3,1) or (4,1)

Figure 2: The tables created on each of 4 processors representing this example PAG. The specific entries for task prefixes with in degree greater than zero depend upon the actual program execution, but here all multiple possibilities are shown.

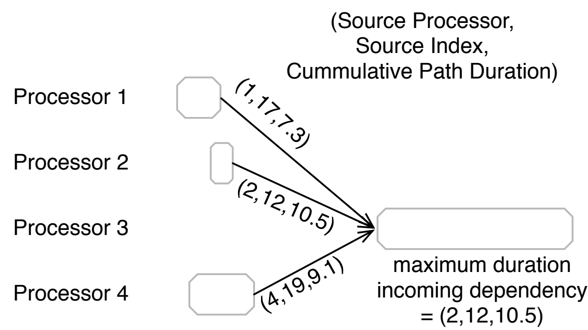


Figure 3: Illustration of how three incoming message dependencies are merged by recording only the one with maximal path duration. Each message contains information pointing back to a table entry for its sending task, as well as the critical path's duration.

```
class myClass: public CBase_myClass {
  ...
  MERGE_PATH_DECLARE(A);
  ...
  void recvGhost(msg *m) {
    buffer_msg(m);
    MERGE_PATH_MAX(A);
    if(received_all_msg()){
      MERGE_PATH_RESET(A);
      subsequent_task();
    }
  }
};
```

Figure 4: Multiple dependencies occur when buffering messages prior to executing a task.

To use the new critical path detection capabilities, a Charm++ program must be modified to add macros at each point where the program in-degree is greater than one. Charisma or Structured Dagger programs, however, do not require any modification because their compilers have been adapted to automatically insert the macro instances wherever required. Sections 5.1, 5.2, and 5.3 provide examples of all the various places in each of the three languages where the in-degree for a task could be greater than one.

5.1 Charm++ Programming Model

The first language supported by the new critical path detection scheme is Charm++. All Charm++ programs are written mostly in C++, with a small interface portion that is parsed by a very simple translator that generates C++ code.

In the Charm++ language, there are two places where in-degree is greater than 1. In one of these two places, the user must augment their code with simple annotations specifying that multiple incoming messages are dependencies for a certain task.

1. Reductions from multiple objects to a single destination entry method result in an in-degree greater than one. The reduction framework in Charm++ has been modified to correctly compute the maximal incoming paths along any reduction tree, so the user does not need to modify an application to handle the dependencies that arise due to reduction calls.

2. The user can buffer incoming messages explicitly until all necessary messages have arrived, at which point the execution of some task is performed. Each time the user buffers a message, a new implicit dependency is created and the in-degree of the task increases. Figure 4 shows an example of such explicit buffering. To correctly handle the critical paths, the user must augment the Charm++ program with macros specifying that there are multiple incoming message dependencies. To do this, the user must add three macros to declare, merge, and reset the paths: `MERGE_PATH_DECLARE(t)`, `MERGE_PATH_MAX(t)`, and `MERGE_PATH_RESET(t)`. After all dependencies have arrived, the stored path information is reset for possible use in a future iteration. Each macro takes a parameter that distinguishes between multiple sets of dependencies that could be declared within the same scope in the source code. For example, if a single class has two tasks, each with multiple incoming dependencies, then `MERGE_PATH_DECLARE(A)` and `MERGE_PATH_DECLARE(B)` could be used to create data structures that store the two different incoming maximal paths.

5.2 Structured Dagger Programming Language

The Structured Dagger language is an extension to Charm++. It allows a programmer to express a complicated control flow with various dependency patterns easily. In Structured Dagger programs, messages are buffered automatically until all input dependencies for an object have arrived, at which

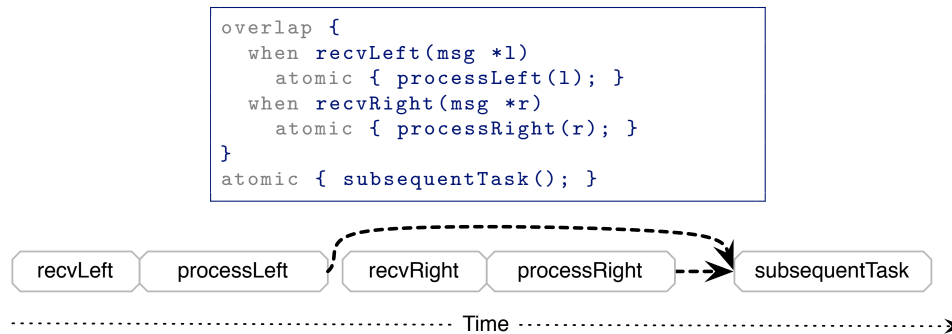


Figure 5: In a Structured Dagger program, the task following an overlap block will depend upon program order dependency edges produced by each of the overlapped tasks.

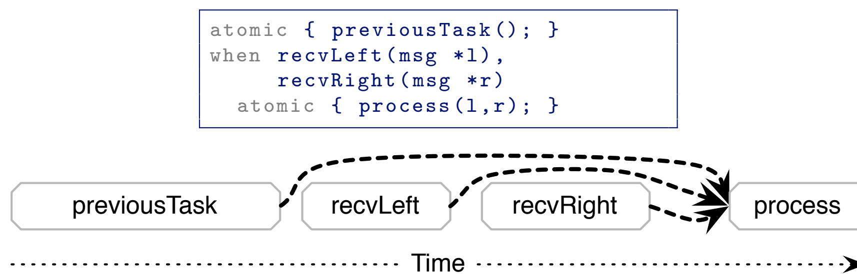


Figure 6: In this Structured Dagger example, the `process` task depends upon 2 messages as well as the program order dependency from the task preceding the `when` statement.

point the object's entry method is invoked. The dataflow patterns and all associated dependencies in the program are clearly expressed in the language, so the programmer does not need to add extra annotations to the program.

For clarity and to help implementers of similar languages, described below are the types of dependencies that must be handled for languages similar to Structured Dagger. Structured Dagger provides language constructs that impose ordering restrictions between the ends of some tasks and the beginnings of other tasks. Structured Dagger also provides language constructs for message sending and receiving.

1. All concurrent tasks specified inside an `overlap` block must complete before any subsequent task begins. Thus there are program order dependencies from the ends of the overlapped tasks to the beginning of the subsequent task. Figure 5 shows an example of this pattern.

2. Each `when` clause requires that one or more messages have been delivered prior to executing the following statement. Additionally, it requires that the preceding statement has also finished executing. Figure 6 shows an example of this pattern with two message dependencies and one program order dependency.

5.3 Charisma Programming Model

The Charisma language [8] is built upon Charm++. It allows a programmer to express various static dataflow and producer-consumer patterns easily. The dataflow patterns and all associated dependencies in the program are clearly expressed in the language, so programmers do not need to modify their programs for use without the critical path detection scheme.

For clarity and to help implementers of similar languages, we will describe the types of dependencies that must be handled for languages such as Charisma. Charisma provides language constructs that impose ordering restrictions between the ends of some tasks and the beginnings of other tasks. Charisma also provides language constructs for producing and consuming messages.

There are two places in Charisma programs where tasks can have in-degrees exceeding one. The Charisma compiler has been modified to record the proper critical path information for these types of dependencies.

1. A statement can consume multiple input parameters:
   `workers[i].compute(lb[i+1], rb[i-1]);`

2. A reduction results in multiple dependencies flowing into a single task:
   `(+error) <- workers[i].getData();`

6 Overhead

The overhead of using the critical path detection mechanism is small. Figure 7 plots the overhead for two simple benchmark programs. The benchmark programs were created to measure the costs associated with recording the table entries on each processor and the increased size of each message. Two versions of each benchmark program are run, and their results are compared to determine the overhead. In the first version, the critical path functionality is entirely disabled. The envelopes are not augmented with critical path table references, nor are the critical path tables allocated. The second version is compiled against a version of Charm++ containing the critical path mechanisms described in section 5. These experiments were performed on the Cray XT5 system named Kraken at NICS.

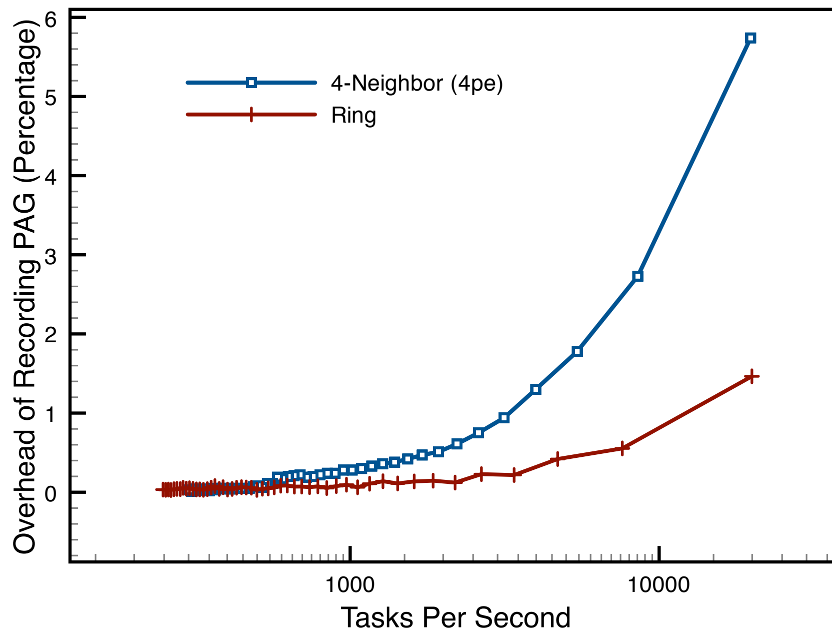


Figure 7: Overhead of recording critical path information for varying computational grain sizes in both a ring benchmark program and a 2-D grid program where each processor communicates with 4 neighbors during each step.

The first benchmark program sends a small message around a ring of Charm++ objects. Each object, upon receipt of the message, performs some amount of CPU work, and then sends a copy of the message to the next object in the ring. The amount of work performed by each object is varied to

simulate various computational grain sizes. If the amount of work is small, the ring proceeds faster, while if the amount of work is greater, the ring proceeds more slowly. Each of the ring executions is timed, and the granularity of the program is calculated: $tasks\ per\ second = \frac{number\ of\ hops\ in\ ring}{ring\ execution\ time}$. The benchmark results shown in Figure 7 correspond to an execution of the benchmark with 40,000 hops around the ring for each granularity sample to amortize away perturbations. The second benchmark program exhibits more complicated communication patterns than the first ring benchmark. Specifically, a 2-D grid of chares is created, with each chare communicating with 4 nearby neighbors each step. Again, varying amounts of fake computations are performed each step to simulate various grain sizes.

Ultimately, the overhead for the ring benchmark executing 10,000 task prefixes per second on each processor will incur an overhead of about 1%. The overhead is caused by the recording of the information necessary to reconstruct a critical path profile. That is, each task executes for about $100\mu s$ with an overhead of $1\mu s$. The overhead for the 4-neighbor program is higher, as expected, because each task sends four times as many messages per step than the ring benchmark.

An experiment was performed to measure how much of the overhead was attributable to the two main mechanisms used to record a critical path or PAG. In this experiment, a portion of the critical path measurement system, namely the code that allocates and updates a table on each processor, is removed. Then the 4-neighbor benchmark program was run to measure the overhead when the critical path duration is propagated in the message envelopes, but the tables are not created.

In this experiment, the baseline 4-neighbor benchmark executes 3769 tasks per second. The first version of the program propagates critical path information through all messages, but does not record table entries and hence would not be able to perform a backwards traversal of the path. The second version does include the full critical path profiling system described in this paper, including the recording of entries in tables on all processors. The first version incurs an overhead of 0.19%. The second version incurs an overhead of 0.31%. Thus about 61% of the costs incurred by the critical path monitoring are caused by the adding 12 bytes to each message envelope, the timer calls measuring the lengths of each task, and the propagating of information found in the messages any time a new message is sent. The remaining 39% of the costs are caused by the recording of information in the distributed PAG tables across the processors.

7 Using Critical Path Profiles

In related work, critical path profiles were gathered in online or semi-online manners, but the resulting critical path profiles were only used for offline performance analysis [7, 4, 6, 5, 10, 12]. One novel contribution of this work is to show that critical path profiles can be used both for automatic online performance tuning and for offline manual performance analysis.

There are already multiple uses of the new critical path detection mechanisms in the Charm++ runtime system. Section 7.1 describes a simple online automatic task prioritization scheme that improves a complicated real-world application's performance by 10.2%. Section 7.2 describes some initial work in using critical paths at runtime to reduce the volume of performance trace data that is gathered for use in offline post-mortem performance analysis tools. Finally, sections 7.3 and 7.4 describe the uses of the critical paths offline in a more traditional manner to guide manual performance analysis.

7.1 Automatically Tuning Task Priorities

One of the most obvious uses of critical path profiles at runtime is to automatically adjust the scheduling priorities for tasks. In the Charm++ programming model, users can associate priorities with each message. Each processor has a scheduler that chooses messages one-at-a-time to execute from a priority queue. Each task in a Charm++ program is simply the non-preemptable execution of a message on a single processor. Typically, Charm++ programs over-decompose a problem so that at any point in time a processor's scheduler queue will contain many messages that need to be executed. The priorities for each message allow high-priority tasks to be executed earlier than lower priority tasks whose messages may have been created or arrived earlier on a processor. In many

Charm++ programs, message priorities are hand tuned using both the programmer's intuition and experimental runs testing different configurations. Such manual tuning is time consuming and may not be portable.

An automatic message prioritization scheme was created in the Charm++ runtime system. The scheme extracts from a critical path profile the types of tasks found along the path, i.e. the *entry method index* associated with each task. The autoprioritization mechanism then modifies message priorities when outgoing messages are prepared by the runtime system just prior to being sent. In the current implementation, only messages allocated with priority bits are modified. The priorities on the messages are adjusted based on whether or not the destination task type is found within a critical path profile. A message whose target entry method is found along the critical path profile is given a high priority while messages destined for other entry methods are given low priorities. Using this simple autoprioritization scheme, speedups can be observed in real applications.

The developers of the OpenAtom quantum chemistry application [2] have found that by manually tuning message priorities, the application performance varies by about 10%. Unfortunately, the manually chosen priority configurations that work well on one parallel machine and input problem do not work well on other machines or with other input problems. Thus if the priorities could be automatically tuned, the performance of the application would improve for many of its users and the effort required to manually tune the program would be eliminated. Our test shows that indeed an automatic message prioritization scheme is useful.

To test the effectiveness of the automatic message prioritization scheme, we made two minor sets of modifications to the OpenAtom source code. The first modification was to add macros to mark the multiple incoming dependencies for certain tasks. These changes required adding the `MERGE_PATH_DECLARE` and corresponding `MERGE_PATH_MAX` and `MERGE_PATH_RESET` macros in 6 locations within 3 different classes in the source code. The second modification was to add a call to `useThisCriticalPathForPriorities()` after a specific iteration to start traversing the critical path and request that it be used for message prioritization. All of these changes were easy to make.

To run the program, we used the *water 32 70* system and ran the program on 64 processors of the Cray XT5 machine, Kraken, at NICS. The configuration files were modified to enable the prioritization of three types of messages, of which only two are enabled by default. We did not modify any of the specific message priority coefficients. Then the application was run for 40 application iterations. The first half of the iterations used the default message priorities while the second half used the automatic message prioritization scheme based upon the critical path gathered at iteration 20. The iteration timings output by the program were analyzed to determine the benefits of the automatic prioritization scheme relative to the default message priorities. Specifically, two startup iterations and the two fastest and slowest iterations for each case were ignored, resulting in 15 remaining iteration times for each case.

The resulting performance of the automatically prioritized portion of the application's execution was 10.2% faster than the other portion that used the default priorities. In both portions of the execution, the critical path algorithm is enabled, so any overheads associated with the critical path detection are present in both portions.

7.2 Performance Analysis Data Reduction

A second use of critical path profiles is to reduce the volume of data produced for post-mortem performance analysis at runtime. The critical path profile itself can be used online for filtering trace data produced by parallel programs run on many processors. At the end of the program, the critical path profile is produced and broadcast to all processors. The processors use this resulting critical path profile to determine how to filter their local performance trace logs before writing them to disk. The volume of performance data that needs to be analyzed offline is much smaller. Such savings are significant for large program runs on hundreds of thousands of processors.

To use this feature in the current implementation, the parallel program can simply make a call near the end of its computation to `traceCriticalPathBack()`. This call will trace the critical path back to its origin at which point the entire path will be broadcast to all processors. The recipients of the broadcast will instruct the performance log tracing framework to not output the log files to

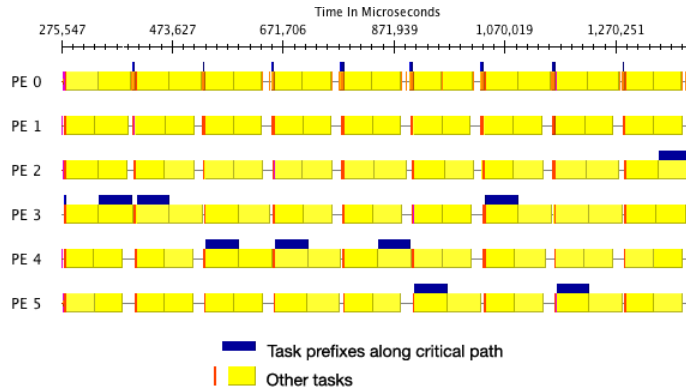


Figure 8: The critical paths can enhance visualizations of performance analysis in post-mortem analysis tools. This shows a timeline view of 6 processors of a Charm++ program simulating the 2-D wave equation. The task prefixes along the critical path are displayed as dark blue bars above on top of each processors' tasks.

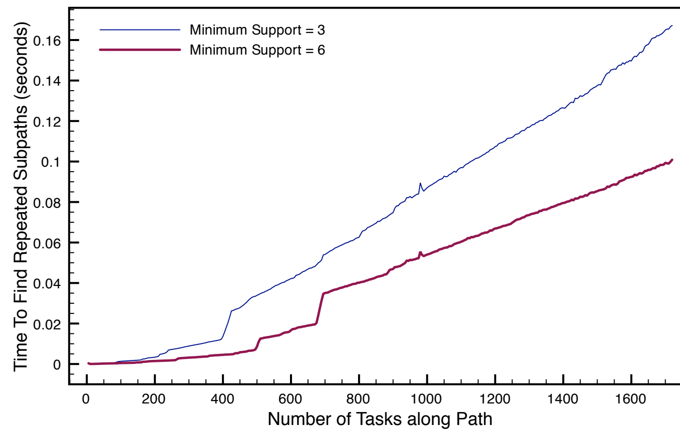


Figure 9: Time to compute frequently repeated sub-paths for varying lengths of input critical paths. The minimum support level is the number of times a sub-path must appear for it to be considered frequent.

disk if the processor is not found along the critical path tasks. Thus any uninteresting processors, namely those not along the critical path, will not write to disk their potentially large trace logs. The exact savings in data volume are program dependent.

7.3 Post-Mortem Performance Analysis

Critical paths can be displayed in a post-mortem performance analysis tool to help visually identify features of the program's execution in a timeline view. Our system provides the ability to generate the necessary trace log data from a critical path profile. Figure 8 shows such a performance analysis visualization of a Charm++ program that solves the wave equation over a 2-d grid. Each processor contains 2 partitions of the 2-D problem domain. Thus about half of the program's execution time is spent along the critical path. There is a neighbor communication of ghost values each step, and each worker task is augmented with code that merges the incoming paths as described in section 5.1.

a b b b b b c d e f g g g g g h i j b b b b b k l m n o p o p q r b b b b s t i u v w b b b b x x y v w b b b b
b A A x y v w b b b b x x x y v w b b b b b x x x y v w b b b b b x x x a b b b b b b c d e f g g g g g h i j b
b b b b b k l m n o p o p q r b b b b s t i u v w b b b b b A A x y v w b b b b b A A x y v w b b b b b A A x
y v w b b b b b x x x y v w b b b b b x x x **a b b b b b b c d e f g g g g g h i j b b b b b k l m n o p o p q r**
b b b b s t i u v w b b b b x x y v w b b b A A x y v w b b b b x x x y v w b b b b A x x y w b b b A A x a b
b b b b b c d e f g g g h i j b b b k l m n o p o p q r b b b b s t i u v w b b b b b A x x y v w b b b b b A x x
y v w b b b b b x x x y v w b b b b x x x y w b b b b x x A **a b b b b b b c d e f g g g g g h i j b b b b b k l**
m n o p o p q r b b b b s t i u v w b b b b x x y v w b b b b x x x y v w b b b b x x x y v w b b b b x x x y v
w b b b b b x x x **a b b b b b b c d e f g g g g g h i j b b b b b k l m n o p o p q r b b b b s t i u v w b b b b**

Figure 10: A frequently repeated sub-path is shown in bold

7.4 Phase Detection

A final use of critical paths is to automatically detect repeating phases in a parallel application's execution. If the application's behavior is relatively static and the program executes a large number of iterations, which is common in most scientific computations, then the critical path should reflect the repeated phases of the program. Searching for phases in complete trace data would likely take longer than searching for phases in the much simpler critical path through the program's execution.

To make the problem of finding phases in the critical path easier, a critical path can be translated into a string over an alphabet of different types of tasks. The problem of finding repeating phases of execution along the critical path is the same as the problem of finding frequently repeated substrings. A preliminary implementation has been created using a dynamic programming technique to quickly build up the sets of frequently occuring substrings. This implementation arbitrarily requires that the frequently used substrings must have a minimum support level of 6, meaning that any candidate substrings must appear at least six times in the whole string. The higher the minimum support level, the faster the algorithm runs, but very long infrequent strings might not be observed.

The final output from the technique is the substring with maximal weighted coverage in the whole string. The weighted coverage is calculated to be $number\ of\ substring\ instances \times (substring\ length)^2$. This weighted coverage measurement favors frequently occurring repeated substrings while especially favoring substrings with long lengths. Figure 10 shows a set of resulting repeated substrings, highlighted in the whole critical path. In the figure, each type of task is represented by a unique ASCII character. Multiple iterations, and their corrsesponding repeated portions are visible in the figure.

The execution time for the frequent sub-path technique is shown in Figure 9. The method appears to have linear execution time for the trace string produced by a run of OpenAtom, with some beneficial cache effects for small strings. The actual worst-case performance is data dependent. With a minimum support level of 6, it takes about 0.1 seconds to extract the frequent sub-path from an OpenAtom critical path profile of length 1721.

8 Future Work

In addition to critical paths, there is other useful information contained in a PAG. Two specific types of paths found in a PAG could be useful, namely *Near Critical Paths* and *Parallelism Exposing Paths*. In the future, the utility of identifying these paths could be examined. We are currently investigating new complex types of schedulers that use a PAG gathered at runtime to better schedule tasks, whereas in this paper, a simpler scheduling mechanism just adjusted message priorities.

Near Critical Paths: Paths with duration close to duration of the critical path are useful because they will become critical paths if the actual critical path is shortened. Hence, they can provide a bound on the execution time improvement in the program when a critical path is optimized.

Parallelism Exposing Paths: Nodes in the PAG that lead to large amounts of execution time across many tasks are important because they expose concurrency in the program. If these nodes, and their preceding critical paths are scheduled at higher priorities than other work, the potential concurrency in the program will be increased earlier in the program's execution. This would result in a potentially faster program. Although ideally all available concurrency ought to be exposed as early as possible, there may be costs associated with the degree of concurrency, and in programs

where this factor dominates, the paths exposing concurrency ought to be delayed. It is not yet clear how useful such paths are, or how they would be computed from a distributed PAG.

9 Conclusion

This paper presented a novel type of adaptive behavior that can be performed within a parallel runtime system. Specifically, for the first time, the paper described how to record critical paths for a running message-driven parallel program. It was then shown how such critical paths can be used for the online dynamic reconfiguration of the program. An example application, OpenAtom, saw speedups of 10.2% when its message priorities were automatically reconfigured based upon information extracted from an observed critical path profile.

References

[1] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[2] Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, and Glenn J. Martyna. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[3] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[4] J Hollingsworth and Barton Miller. Slack: A new performance metric for parallel programs. *cs.umd.edu*, Jan 1994.

[5] J. K. Hollingsworth and B. P. Miller. Parallel program performance metrics: a comprison and validation. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 4–13, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[6] Jeffrey K. Hollingsworth. An online computation of critical path profiling. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 11–20, New York, NY, USA, 1996. ACM.

[7] J.K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):1029–1040, Oct 1998.

[8] Chao Huang and Laxmikant V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[9] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[10] M Schulz. Extracting critical path graphs from mpi applications. *Cluster Computing, 2005*, pages 1 – 10, Sep 2005.

[11] Ramkumar V. Vadali, Yan Shi, Sameer Kumar, L. V. Kale, Mark E. Tuckerman, and Glenn J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16):2006–2022, Oct. 2004.

[12] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 366–373, 1988.