Determinism and Reproducibility in Large-Scale HPC Systems*

Wei-Fan Chiang
School of Computing,
University of Utah, USA
wfchiang@cs.utah.edu

Ganesh Gopalakrishnan
School of Computing,
University of Utah, USA
ganesh@cs.utah.edu

Zvonimir Rakamarić
School of Computing,
University of Utah, USA
zvonimir@cs.utah.edu

Dong H. Ahn
Lawrence Livermore National
Laboratory, Livermore, CA
ahn1@llnl.gov

Gregory L. Lee
Lawrence Livermore National
Laboratory, Livermore, CA
lee218@llnl.gov

ABSTRACT

The ability to reproduce simulation results (*external determinism*) goes a long way towards enhancing the trustworthiness of high performance computing simulations. The ability to replay schedules (*internal determinism*) greatly facilitates reproducing bugs, and helps reduce wasted programmer productivity. In this paper, we consider these issues in the context of software libraries and APIs used in today's mainstream high performance computing (HPC) systems as well as expected to be used in upcoming high-end systems. After cataloging the main sources of external and internal nondeterminism, we summarize two thrusts in our current research: (1) mechanisms to control internal nondeterminism by active schedule control, and (2) techniques that may help assess the extent of result nondeterminism in floating point calculations.

Keywords

Determinism, Reproducibility, Concurrency, Dynamic Testing, Non-associative operations, Floating-point Arithmetic

1. INTRODUCTION

High performance computing (HPC) is often regarded as "the third pillar of science" that facilitates new scientific discoveries through simulation. The ability to reproduce simulation results (*external determinism*) goes a long way towards enhancing the fidelity of HPC simulations, and hence the scientific discoveries themselves. It also facilitates uncertainty quantification, a key validation step towards predictive computation where the changes in simulation outputs must be directly correlated to the changes in some simulation model parameters, and not accidentally affected by the vagaries of floating-point calculations or the side effects of missed data races. The ever-growing complexity of high-end HPC systems also means that one must provide good control mechanisms on *internal* or *schedule* nondeterminism as well. Without them, "Heisenbugs" and other bugs that rarely emerge would be difficult to reproduce and fix.

Approaches to achieve determinism must be based on the scale of problems being solved, the application domain in which they are solved, and the specific libraries and APIs used. This paper summarizes our preliminary research on understanding as well as providing determinism in large-scale HPC systems and the scientific applications that run on them. We present two classes of results: (1) classifying some of the key sources of nondeterminism that present significant challenges to high-end systems, (2) techniques that help assess (and that may eventually help control) the impact of nondeterminism in both the internal and external sense. Given that achieving determinism can incur significant runtime and resource costs, a long-term goal of our research is to achieve *best effort* determinization with an ability to estimate the costs of achieving various levels of determinization.

We first begin with relevant working definitions or connotations for the term 'determinism' [**?**]:

- *Data-race free:* the presence of data races can result in execution-order-dependent final results;
- *Determinate (external determinism):* a system can be considered deterministic if the final results are the same regardless of the internal execution details;
- *Internal determinism:* even the internal execution steps (traces) are required to be the same;
- *Functional determinism:* this refers to the absence of mutable state updates, inherently giving rise to determinate outcomes;
- *Synchronous parallelism:* some regard synchronously parallel execution models as deterministic, as opposed to asynchronous models.

In this paper, we will employ *external determinism* and *internal determinism* whenever clarification is needed; we do not consider the other definitions or connotations.

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-608712). The Utah group was supported by NSF Awards OCI 1148127 and CCF 1241849.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WoDet '13 Houston, Texas, USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

Determinism Challenges in High-End HPC Systems.

There are several challenges in providing determinism at scale in HPC systems.

• Enforcing deterministic execution ordering in the presence of *large-scale* concurrency can incur significant performance penalties. Key advances are essential in this area. Many parallel programming models employ nondeterministic constructs and scheduling policies to improve performance; these features work against attempts to determinize.

• Sources of nondeterminism are becoming increasingly complex and diverse. Many applications exhibit intricate dependencies on third party packages, runtime, and operating system services that often introduce nondeterminism. Thus, controlling nondeterminism in the applications themselves alone is not enough. In addition, the increasing heterogeneity along with the increasingly diverse and resource-constrained execution environments makes this already complex picture even worse. These trends necessitate community-wide efforts that are yet to take root.

• The inability to distinguish between test failures and test-result changes due to nondeterminism can severely undermine regression testing. A test outcome in which the exact numeric results are not reproduced can, for instance, mean either a numerical result change (due to operator non-associativity) or a missed data race (given how hard data-race detection is, in general).

• The inability to replay executions deterministically hampers correctness verification. Often, one avoids reasonable code changes, for fear of introducing external nondeterminism. For example, Loh [**?**] mentions this: *"we found cases, for example, where we refrained from changing the source code because changing* `((2*pi)*k)/N` *to* `2*((pi*k)/N)` *or changing* `X*(1/deltat)` *to* `X/deltat` *changed floating-point results subtly. We do not know if the results were more accurate or less, only that they were slightly different. These differences prevented us from making the source code more readable or run faster."*

• Uncertainty quantification (UQ) is a suite of emerging computational methods to quantify the accuracy of a simulation. It works by perturbing problem parameters (*e.g.*, oxide thickness of a CMOS transistor) and seeing how it affects the outputs (*e.g.*, the transconductance of the transistor). To calibrate the sensitivity of parameter perturbations, UQ approaches typically attempt to separate the most sensitive parts of the simulation from the less sensitive parts. Ideally, these studies must benefit from formally characterized levels of result reproducibility, and such characterizations are missing. Further, UQ studies must also be unencumbered by floating-point non-associativity and bugs such as data races.

Project PRUNE.

We are in the very early stages of our project Providing Reproducibility on Ubiquitously Nondeterministic Environment (PRUNE) that aims to investigate and provide scalable ways to control nondeterminism. This paper begins by presenting the domain of our focus, and the parallel programming libraries and APIs that play a central role in this domain (§**??**). We then point out the specific sources of nondeterminism that can arise in this domain, hoping to calibrate these sources in terms of importance and to discover additional sources we may have overlooked (§**??**). In §**??**, we present solutions at two levels: internal determinization for execution traces (§**??**) and preliminary studies of external nondeterminism in the floating point domain as a function of the reduction tree shape (§**??**). Throughout the text we discuss future plans, and we present our concluding remarks in §**??**.

2. CHARACTERIZATION OF APPLICATION DOMAIN

We now present the APIs, libraries, and programming constructs that are widely used both in today's mainstream systems as well as anticipated to be used in future high-end HPC systems. In §**??**, we will describe the sources of nondeterminism specific to these choices.

Message Passing and Shared Memory Programming.

We confine ourselves to limited subsets of the MPI 2.0, OpenMP, and Pthread standards. These choices are natural in the area of HPC, given the degree of adoption that these APIs have received. We anticipate that OpenMP will be used in modalities that stray outside its usual fork/join execution style. There are constructs in OpenMP that spawn threads and tasks, and we expect that many applications will involve these and other similar constructs. Pthreads is expected to be an integral part of our work, especially given the momentum behind the C11 and C++11 standardization efforts. In addition to OpenMP and Pthreads, accelerators in the form of vector processing or graphic processing unit are being increasingly exploited in HPC systems. Languages such as Cilk [**?**] with their task spawn/join model as well as hardware transactional memories will also find increasing use. Each of these shared memory models comes with its own class of data races and issues pertaining to both internal and external determinism. There are opportunities to develop model-specific data race checking and schedule control methods.

Resource Considerations and "External Factors".

With growing scale, HPC systems are increasingly faced with various computational resource limits, including available memory and network bandwidth. Resource allocation schemes will, therefore, have a semantically visible impact on behavioral reproducibility. For instance, decisions taken with respect to MPI message handling and buffer allocation can have visible semantic effects.

As one specific example, the *eager limit* [**?**] defines how much buffering is available in the MPI runtime to buffer the message contents of an MPI send. In advanced MPI libraries, this can be a function of how buffer *credits* are managed between the sender and the receiver [**?**]. The amount of buffering available can vary run-to-run. This, in turn, can have other observable effects. In our previous work [**?**], we have shown that MPI can deadlock both when *too little buffering* is provided, and when *too much buffering* is provided.[1]

The load on the interconnection network can also be a factor that governs determinism. In MPI, a network delay may cause an ordering variation of "wildcard receives," leading to an exceptional condition that does not appear in other

[1]Too much buffering can allow MPI Waits associated with the sends to return early. This in turn allows the issue of later occurring MPI sends, and these sends can increase the number of senders that can match an already posted MPI wildcard receive.

variations. Further, in advanced implementations of MPI collective operations, the reduction trees can be a function of the network load. Also, in the case of OpenMP, the placement of OpenMP tasks on cores (core affinity policies) can affect the reduction trees seen by OpenMP thread teams.

All the above resource decisions are a strong function of the problem scale. Therefore, the same problem when run at different scale can exhibit both internal and external nondeterminism due to the changing nature and allocation of resources. There are also several external factors over which one has limited control. These include:

- How the code is compiled, including dynamic compilation methods;
- How the MPI and OpenMP runtimes are designed.

Dialogues with compiler and runtime designers are crucial to ensuring that application developers will have flexible ways to exert control over the MPI and/or OpenMP runtimes. Such control can be valuable during testing and verification, especially in situations where determinizing in particular ways (*e.g.*, building specific kinds of reduction trees or exercising specific thread schedules) can help with isolating root-causes of bugs, or with guaranteeing bitwise reproducible results.

3. SOURCES OF NONDETERMINISM

One can classify nondeterminism and its consequences depending on many factors:

- *Objectives.* Is (internal) nondeterminism control being used to reproduce an already encountered bug? Or is it to understand and handle external nondeterminism due to result variability?
- *Tolerance.* How much nondeterminism is one willing to tolerate in a specific context? The larger the scale, the more expensive finer nondeterminism control becomes.
- *Transparency.* Do the control mechanisms involve algorithmic changes, or do they avoid such changes by controlling the behavior of the runtime, for instance, enforcing the same reduction tree (*e.g.*, for MPI) or the same loop execution order (*e.g.*, for OpenMP)? Demmel et al. [**?**] touch upon this point, suggesting that there are essentially two solutions for controlling floating-point non-associativity related external nondeterminism: (1) fix the reduction tree or (2) employ algorithmic changes such as *pre-rounding*.

Keeping the above types of contextual information in mind, we can now go through the application domains characterized in §**??** and make remarks about the specific sources of nondeterminism.

Within threads. Consider a single thread performing vector operations. Depending on the alignment of the memory data it deals with, the vector execution may involve a special case non-aligned step followed by the general case of vector operations. The manner in which the results are assembled affects external determinism (*e.g.*, floating point).

Between threads within a process. Clearly, shared memory data races are notorious for causing both types of nondeterminism. They are also extremely expensive to detect, although significant progress is being made in some areas [**?**]. If threads involve the use of locks or transactions, then depending on the nuances of the transaction semantics, both types of nondeterminism can be introduced.

Data races that occur between threads depend also on whether they are associated with OpenMP's traditional worksharing, or its new tasking and Cilk-style task spawn/join or further Habanero Java-style async/finish. These categories of races are expected to be easier to detect [**?**, **?**]. The degree of execution controllability of runtimes (*e.g.*, OpenMP and Cilk/Habanero Java runtimes) can determine the success of detection.

Between processes. MPI wildcard receives can affect internal nondeterminism, depending on the exact MPI send commands picked up by a wildcard receive. Often these wildcard receives are in a loop, and sometimes they are also followed by a reduction-style operation: in these cases, even though the same collection of sends matches with the receive, their sequential order can be different. We will elaborate on some mechanisms to handle this type of nondeterminism in §**??**.

MPI collectives, especially those with non-associative reduction operators, are notorious for causing external nondeterminism due to floating point result non-reproducibility.

Related to resources and timing. In §**??**, we have already pointed out how the vagaries in terms of MPI buffer availability can cause nondeterminism due to schedules that cannot be replayed or sometimes "unexpected" deadlocks. Resource deadlocks and unexpected crashes can also be caused by resource leaks, in turn caused by various "free" operations (*e.g.*, request objects leaked by forgotten MPI waits, communicator and type creates not followed by corresponding free operations) MPI probes, test/wait operations in the some/all combination also are a function of execution dynamics. Some of the behavioral changes can show up when the code is ported to a new platform.

Related to floating-point. We relegate §**??** to study external nondeterminism due to non-associative floating-point operations. We also briefly look at what common loop optimizations can do, confirming that optimizations preserve dependencies, and hence do not introduce any external nondeterminism. Similar facts have been observed in related work on comparing sequential programs and their parallelized versions using symbolic execution [**?**]. The gist of this discussion is that reduction trees are a rich source of external nondeterminism while types of commonly employed loop transformations are not.

4. PRELIMINARY APPROACHES

This section presents two concrete directions the PRUNE project plans to take over the coming months. We first (§**??**) describe a tool called Distributed Analyzer of MPI (DAMPI) [**?**, **?**, **?**] that realizes a scalable distributed algorithm for replaying certain types of nondeterministic schedules in MPI applications. We will summarize our current ideas to extend DAMPI to our research on determinism. Next (§**??**), we present our initial exploration on how reduction trees involving non-associative floating-point operations can affect the final result. Our eventual goal is to control both internal and external nondeterminism behaviors using these tools.

4.1 Internal Determinism Control

For applications with nondeterministic MPI calls such as `MPI_Irecv(MPI_ANY_SOURCE)`, many conditions affect ordering variations of the message matching of these calls, which can lead to different visible effects. Some of the ordering variations are hard to reproduce, as not only can internal

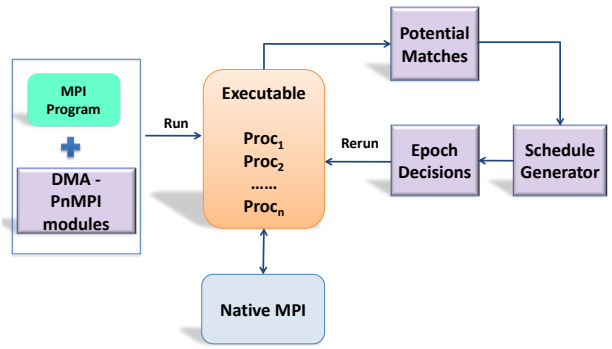

Figure 1: Distributed MPI Analyzer (DAMPI).

factors cause them, but also external ones, including transient network delays and placements of MPI rank processes. To verify the correctness through this complexity, simply modulating the absolute times at which MPI calls are issued (*e.g.*, by inserting nondeterministic sleep durations, as done by stress-testing tools) is ineffective because most often this does not alter the way in which racing MPI sends match with MPI's nondeterministic receives deep inside its runtime. Also, such delays unnecessarily slow down the entire testing.

In our recent work [**?**, **?**, **?**], we have shown the efficacy of our distributed algorithm and its implementation embodied in a tool called DAMPI. DAMPI takes an MPI application along with its inputs and runs the target application on an HPC machine. It first runs the target under the "native" schedule while recording the actual matches using a message piggy-back mechanism (Figure **??**). Then, it calculates alternative match possibilities with very low omission rates using a scalable Lamport clock-based [**?**] algorithm. Finally, DAMPI generates a determinization recipe (schedule generation and epoch decisions) and reruns the target while allowing users to pick specific determinizations of the internal nondeterminism of message matching.

Experimental results show that DAMPI can effectively test realistic applications that run on more than a thousand MPI processes by exploiting the parallelism and the memory capacity available in HPC machines. It has successfully examined all benchmarks in the Fortran NAS Parallel Benchmark suite [**?**]. There, we demonstrated that DAMPI is able to provide nondeterminism testing coverage with instrumentation overhead of less than 10% compared to the ordinary testing that does not offer such coverage.

Planned Future Work. Currently, DAMPI exerts no control over collective operations. PRUNE plans to extend DAMPI in order to help debugging both logical errors and non-reproducible behavior with respect to floating-point calculations. To assist users in debugging varying floating-point behavior, one approach is to fix the reduction tree shape [**?**]. However, that can incur significant run-time costs at scale, especially if one shapes the reduction tree in an *ad hoc* manner. That suggests a few research directions:

- Using DAMPI, determinize collective operations containing non-associative reductions in different ways in an attempt to isolate the parts of the code most numerically sensitive to the varying results. This step clearly depends on the values that are passed through the reduction tree. Studies in §**??** show how one may be able to better understand the impact of the distribution of values on the overall results.
- If one determinizes a reduction tree in such a way that the *latest* calculated values are reduced first, the overall performance can suffer. Performance evaluation methods have a role to play in suggesting the preferred reduction-tree shapes. As mentioned in §**??**, reduction trees in OpenMP may also be decided based on task-to-core mappings and the underlying network topology.

4.2 Floating Point Nondeterminism

Reduction is a well known distributed operation performed on data spread across threads or processes. When carried out with a non-associative arithmetic operation, however, the reduction can produce slightly varying results, depending on the order in which its parallel operations are performed through its "reduction tree shape." Using a recent tool, Gappa++ [**?**], we will demonstrate the (perhaps obvious) fact that it is *not merely the tree shape but also the exact distribution of value ranges* that can affect the final result. The examples in Figure **??** and Figure **??** will empirically show that the reduction shape and data distribution affect floating-point precision, and hence introduce nondeterminism since the reduction results will vary.

These results further underscore the importance of researching the finer nuances of quantifying the impacts and degree of nondeterminism in reduction operations. Such research will then guide us in devising various determinization techniques that provide progressively higher levels of deterministic guarantees at the expense of progressively higher performance overhead. These different levels of determinizations can suit different challenges with respect to the nondeterministic floating point reduction. For example, debugging may require most expensive lock-step internal determinism. The verification phase may only require a reduction result to be a tight range with bounded imprecision, and guaranteeing this may incur lower performance overhead.

Gappa++ is a tool for analyzing numerical behavior of applications, and in particular for computing rounding errors in floating-point computations. It is an improved version of the Gappa proof assistant [**?**]: while Gappa performs analysis of rounding errors using interval arithmetic, Gappa++ relies on affine arithmetic that can more accurately bound computations with correlated errors. As an input, Gappa++ takes a floating point expression and ranges (*i.e.*, intervals) of input variables in the expression. The output of the tool is the computed floating point imprecision range for the input expression. The imprecision range indicates maximum deviation of the floating point result from the actual value.

Same Value Ranges, Different Trees.

Figure **??** shows three different reduction trees, where the operation used is floating-point addition and all operands are equal to a floating-point value L. For the purpose of this experiment, we will fix L to be in three different ranges $L_1 = [-0.1, 0.1]$, $L_2 = [-0.3, 0.3]$, and $L_3 = [-0.5, 0.5]$. Using Gappa++ to estimate the imprecision of floating-point operations due to rounding errors, we compute the impre-

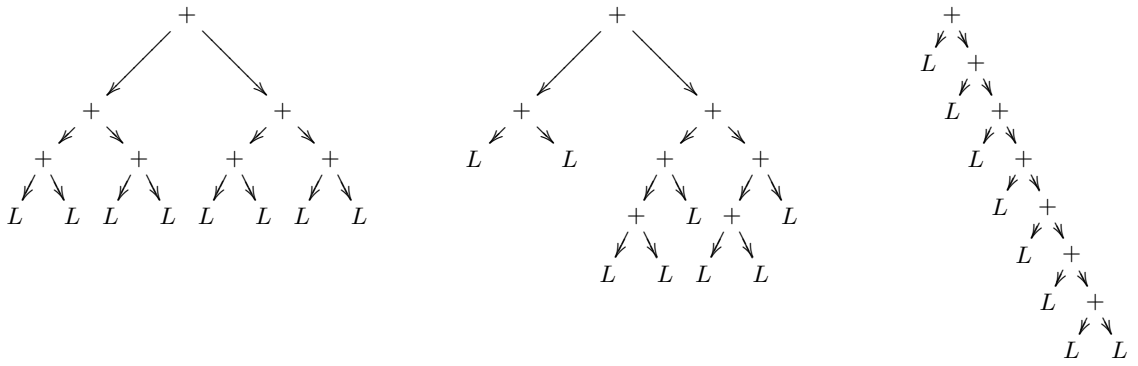

Figure 2: Examples of Different Reduction Trees. Performed operation is floating-point addition and all operands in leaves are equal to a floating-point value L.

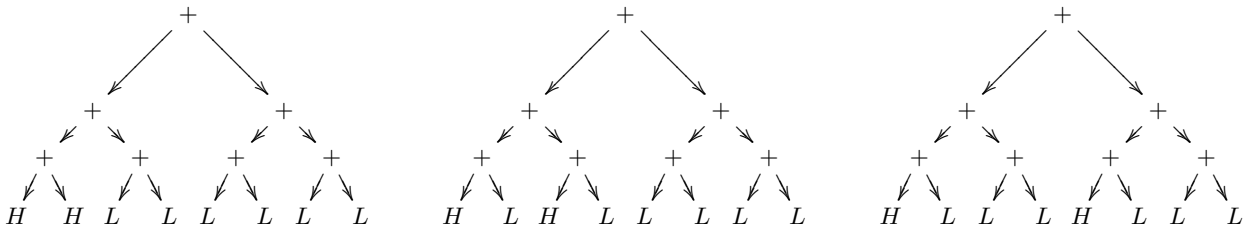

Figure 3: Examples of the same Reduction Tree with Different Distribution of Operands. Performed operation is floating-point addition. Six operands are equal to a floating-point value L and two are equal to H.

cision range I of the final reduction result for all the trees:

L	left tree I_l	middle tree I_m	right tree I_r
L_1	[-12*2^{-27}, 12*2^{-27}]	[-15*2^{-27}, 15*2^{-27}]	[-19*2^{-27}, 19*2^{-27}]
L_2	[-12*2^{-25}, 12*2^{-25}]	[-11*2^{-25}, 11*2^{-25}]	[-16*2^{-25}, 16*2^{-25}]
L_3	[-12*2^{-25}, 12*2^{-25}]	[-15*2^{-25}, 15*2^{-25}]	[-21*2^{-25}, 21*2^{-25}]

The imprecisions clearly vary. In particular, when L is set to L_1 and L_3, I_l is the smallest imprecision interval and is included in I_m, which is in turn included in the largest imprecision interval I_r. However, when L is set to L_2, I_m is the smallest imprecision interval. Therefore, we can conclude that the shape of the reduction tree influences the final reduction result in the presence of floating-point operations. Given that the shape of reduction trees is typically not fixed a priori and depends on many runtime factors beyond a user's control, reduction operations using floating-point operations will produce nondeterministic outcomes, as these examples illustrate.

Same Values, Same Trees, Different Placement.

Figure **??** shows three reduction examples with the *same* reduction tree but with different distribution (placement) of floating-point operands at the leaves. In particular, six operands are equal to a floating-point value L and two are equal to H. For the purpose of this experiment, we will fix $L = [-0.3, 0.3]$ as before; we will also fix $H = [-100.3, 100.3]$. Using Gappa++, we again compute the imprecision range I of the final reduction result for all of the trees:

- left tree $I_l = [-773 * 2^{-25}, 773 * 2^{-25}]$,
- middle tree $I_m = [-772 * 2^{-25}, 772 * 2^{-25}]$,
- right tree $I_r = [-770 * 2^{-25}, 770 * 2^{-25}]$.

The imprecisions again clearly vary, although the differences are smaller in this case: I_r is the smallest imprecision interval and is included in I_m, which is in turn included in the largest imprecision interval I_l. Therefore, we can conclude that even the distribution of the floating-point operands in the leaves of the *same* reduction tree influences the final reduction result in the presence of floating-point operations. Supporting observations are made in related work with respect to different algorithms that calculate π [**?**].

Loop Tiling.

Loop tiling is a well-known compiler optimization technique that makes execution of certain types of nested loops more efficient by improving their temporal data locality [**?**, **?**]. Figure **??** shows our loop tiling example. The original code consisting of two nested loops performing matrix-vector multiplication is shown on the left. (We assume that matrices contain floating-point numbers.) Loop tiling is performed in two steps. First, the original code on the left is transformed by loop-splitting into the code in the middle. Each loop is split into the outer loop (tiling loop) traversing tiles (blocks) and the inner loop (intra-tile loop) performing iterations within tiles. The tiling loops are then moved outwards, and the final outcome of performing loop tiling is the code on the right. We performed a simple experiment to check whether loop tiling as performed in this example influences the final outcome of the computation. Using the Z3 automatic theorem prover [**?**], we proved that all three versions of the matrix-vector multiplication code generate the same floating-point expressions on each element of the out-

$i = 1 \rightarrow N\ j = 1 \rightarrow N\ Y[i] \leftarrow A[i,j] * X[j]$

$it = 1 \rightarrow N$ **by** $T\ i = it \rightarrow it + T - 1\ it = 1 \rightarrow N$ **by** $T\ jt = 1 \rightarrow N$ **by** $T\ i = jt = 1 \rightarrow N$ **by** $T\ j = jt \rightarrow jt + T - 1\ it \rightarrow it + T - 1\ j = jt \rightarrow jt + T - 1\ Y[i] \leftarrow Y[i] \leftarrow A[i,j] * X[j]$

$A[i,j] * X[j]$

Figure 4: Matrix-Vector Multiplication Loop Tiling Example.

put vector. We can conclude that loop tiling transformation typically preserves the order of performed floating-point operations, and hence does not introduce nondeterminism into optimized codes. Related work [**?**] also supports this observation by being able to compare a parallelized MPI program against a sequential MPI application using *Herbrand* (syntactic) equality of the arithmetic expressions that define the "net effect" of symbolically executing the parallel application against the predicate transformer relation of the sequential application.

5. CONCLUDING REMARKS

In this paper, we present our preliminary investigations to understand the sources and the severity of both internal and external nondeterminism in today's mainstream, as well as future, HPC systems. The cost of achieving "perfect" determinization in every context is rather high. One must therefore seek ways in which to calibrate the severity of nondeterminism in each specific context as well as develop methods to seek "best effort" determinization. The level of interest in this area is reflected by numerous approaches, including domain-specific ones [**?**, **?**].

To speculate a little, the evolution of specific metrics for nondeterminism appears to be a very useful step. Our studies in §**??** may be a step in this direction. Devising mechanisms for effecting determinism with affordable costs is another important goal. Our presentation of DAMPI in §**??** is to suggest the beginnings of such techniques. Eventually, there must be software tools and common resources that the community can share for studying and developing determinization techniques. This is also an important goal of the PRUNE project.

Acknowledgments.

We thank Dr. Anh Vo (Microsoft) for stimulating discussions, including understanding many aspects of modern MPI runtimes.