On communication determinism in parallel HPC applications

(Invited Paper)

Franck Cappello *†, Amina Guermouche ‡, Marc Snir†
* INRIA Saclay-Île de France, F-91893 Orsay, France
‡ Université Paris Sud, LRI, INRIA Saclay-Île de France, F-91405 Orsay, France
† University of Illinois at Urbana-Champaign - College of Engineering, Urbana, IL, USA
fci@lri.fr, guermou@lri.fr, snir@illinois.edu

***Abstract*—Current fault tolerant protocols for high performance computing parallel applications have two major drawbacks: either they require to restart all processes even in the case of only a single process failure or they have a high performance overhead in fault free situation. As a consequence none of existing generic fault tolerant protocols matches needs of HPC applications and surprisingly, there is no fault tolerant protocol dedicated to them. One way to design better fault tolerant protocols for HPC applications is to explore and take advantage of their specific characteristics. In particular we suspect that most of them present some form of determinism in communication patterns. Communication determinism can play an important role in the design of new fault tolerant protocols by reducing their complexity. In this paper, we explore the communication determinism in 27 HPC parallel applications that are representative of production workloads in large scale centers. We show that most of these applications have deterministic or send-deterministic communication patterns.**

I. Introduction

Exploring and designing efficient fault tolerant protocols for parallel applications has always been a major research topic of the HPC community [1], [2], [3], [4]. Fault tolerant protocols are essential for rollback recovery and they are also useful for migration and preemptive scheduling.

The role of fault tolerant protocols is to capture information of a distributed execution, during fault free execution, in such a way that the execution could be restarted and eventually terminates successfully producing a correct result. Several decades of research have generated a large variety of fault tolerant protocols [5]. In classifications, they are often grouped in different families: a) coordinated checkpointing [6], [7], b) uncoordinated checkpointing, possibly with message logging [8], [9], [10] and c) communication induced checkpoint [11]. These protocols do not fit the HPC context because they have a high overhead in fault free situation (message logging protocols), require to restart all processes in case of even a single process failure (coordinated checkpointing) or may lead to restart the execution at its beginning (uncoordinated checkpointing without message logging). These weaknesses are significant enough to motivate further research.

One key differentiator between existing fault tolerant protocols is the level of determinism they assume on applications. Coordinated checkpointing and uncoordinated checkpointing without message logging were designed to work in the extreme situation where all processes actions can be non deterministic. These fault tolerant protocols compute checkpoints without considering any assumption on the execution following a restart after a failure. This execution after restart can be completely different and incomparable to the original execution before the occurrence of the fault. As a consequence these protocols need to restart all processes. Message logging protocols improve this aspect and do not need to restart all processes because they assume more determinism: the application code for every process should be piecewise deterministic [12]. One way to model the execution of the piecewise deterministic process is to consider that its state is only influenced by deterministic local actions and message receptions. The later may be the only non-deterministic events and all non-deterministic actions are modeled as non-deterministic receptions. One property of the piecewise deterministic assumption is that for any given process across multiple executions (or re-executions), sends are guaranteed to be identical if non-deterministic actions are replayed identically. This property is used by message logging protocols to

Supported by INRIA-Illinois Joint Laboratory on PetaScale Computing. Supported in part by NSF grant 0833128

avoid restarting all processes [1]. Thus considering more application determinism helped to solve some issues of coordinated checkpointing protocols. However message logging protocols have other limitations: they have a high performance overhead in fault free situation because they need to copy all messages and store message copies and message receptions order of every process on stable storage.

In principle, if communications in parallel HPC applications were deterministic, the overhead of fault tolerant protocols could be drastically reduced since most of it is due to guarantee consistent restarts and correct termination of distributed executions in the presence of non-determinism. The initial results in [13] indicates this to be the case. However, in many HPC MPI applications, programmers relax execution determinism to increase concurrency, reduce coordination overhead and improve scheduling. Resulting codes ofter use minimal synchronization and allow a high degree of asynchrony. The objective of this paper is to analyze a significant panel of HPC applications and explore the existence of determinism sources that could be exploited later to design better fault tolerance protocols [2]. More precisely we want to explore the existence of determinism in communication patterns since HPC applications rarely feature non determinism in their computation sections [3].

Next sections introduce two forms of communication determinism that we call "Communication-determinism" and "Send-determinism". We suspect that these forms of communication determinism is dominant in HPC applications. The rest of the paper will demonstrate this hypothesis. Before introducing the analyzed applications, we present in section III formal definitions of Communication and Send-Determinism. Then we introduce, in section IV, 27 application codes and benchmarks considered in this study. Section V gives the most significant send-deterministic and non-deterministic communication patterns found in the 27 applications codes. Section VI presents the result of this analysis.

II. Communication Determinism and Send-determinism in MPI HPC applications

Many MPI HPC applications follow a SPMD programming style where computational algorithms are implemented by separate dependent communications and computations sections, although communications and computations may overlap during the execution. Typically, MPI HPC applications use point to point and collective communications. We assume in this paper that collective communications are implemented atop point-to-point communications and the implementation is determinisitic. The last condition is satisfied by the MPI implementations we are familiar with. Therefore, w.l.o.g., we can restrict our attention to point-to-point communication. We ignore in this paper MPI features that are rarely used, such as *MPI_Cancel*, or features that can be handled by extending the arguments in the paper, such as I/O or one-sided communication.

For sake of clarity and precision, we present relevant point-to-point communication functions of MPI. In MPI, point to point communications use mainly two functions: *MPI_Send()* for send and *MPI_Recv()* for reception. *int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)* sends the message specified by the buffer address *buf* and the arguments *datatype* and *count* to the process *dest*. The message has the tag *tag*. The communicator argument *comm* specifies a *communication domain* that is both a name space (*dest* is interpreted relative to *comm*) and a separate communication plane (communications using different communicators do not interfere with each other). *int MPI_Recv(void *buf, int count, MPI_Datatype datatype,int source, int tag, MPI_Comm comm, MPI_Status *status)* is the function used to perform receives. The process receives a messages with the tag *tag* from the sender *source*. The received data is placed in the buffer indicated by *buf, count* and *datatype*. The argument *status* returns information on the received message.

The *source* argument and the *tag* argument can either or both have special “don’t care” values that match any source and/or any tag. Messages are ordered; a receive will receive the first sent message from the specified source that has a matching tag; if no source is specified, it will receive the first matching message from some source.

These are blocking calls: The progression of the calling code is stopped until the communication is completed [4]. These two functions have non-blocking variants: *MPI_Isend()* and *MPI_Irecv()*. These functions return before the completion of the communication, in order to allow its overlapping with computation; they have an extra *request* argument that is used to return a handle to the ongoing communication. The programmer

[1] to ensure a compatible re-execution, all restarting processes receptions are controlled to make sure that same emissions are replayed until reaching the point where the fault occur or the last communication before the end of the application.

[2] Proposing new fault tolerant protocol is out-of-scope of this paper.

[3] while an application may use random number generation, it generally set deterministically the seed at the beginning of the execution leading to the generation of a deterministic sequence of numbers.

[4] Note that there are several versions of these functions with different semantic of the termination.

can use other functions to test the status of the communication. *MPI_Test (mpi_Request *request, int *flag, MPI_Status *status)* tests for the completion of the communication indicated by *request*; if the communication is complete, then information on the received is returned in *status*. *MPI_Wait (mpi_Request *request, int *flag, MPI_Status *status)* blocks until the communication is complete. In addition, functions are provided to wait for the completion of any or all requests in a list.

MPI applications can exhibit nondeterminism in multiple ways. In particular, the MPI process can be multi-threaded, with a nondeterministic concurrent execution; the timing of a message arrival can impact execution logic, when *MPI_Test* is used; and the relative arrival time of messages sent by different sources to the same destination can affect the matching of sends to receives, when the receives specify a "don't care" source. However, HPC codes tend to eschew non-determinism, or use it in very limited form. A typical form of non-determinism is code where multiple concurrent incoming messages will be handled in the (non-deterministic) order they arrive. (This can be done by posting multiple non blocking receives then using an *MPI_Waitany* call to complete one of the posted receives.) The execution logic of the receiving process will depend on the arrival order of the messages. However, it is typically the case that the outcome of the computation will node depend on the arrival order: The execution of the process will consist of multiple sections, where there is non-determinism within each section, but the state at the end of each section is deterministic. Furthermore, non-determinism "does not propagate": the order and content of sent messages is not affected by receive order. We call such execution *send-deterministic*.

Verifying that communication patterns in MPI HPC applications are deterministic or "send-deterministic" is important since this will allow the design of new fault tolerant protocols. The objective here is not to define and demonstrate the correctness and properties of some new protocols but to introduce their main principles: Deterministic or Send-deterministic communication patterns means that message sends for every process are identical across executions, from a given set of input parameters. Thus instead of logging all messages, as needed by message logging protocols, to reconstruct the state of failed processes, these messages could be reconstructed from the re-execution of their senders. Since deterministic and send-deterministic communication patterns guarantee that same messages will be resent by the re-execution of their senders, the state of a process can be re-built from its re-execution and the re-execution of the processes that sends it a message. Thus, in principle, messages do not need to be logged to lead a process to a point where it execution is coherent with the other processes. Note, however, that some precautions are needed to avoid a domino effect (forcing the whole execution to restart from the beginning). In [13], we present two new fault tolerant protocols based on these principles.

Before checking the existence of determinism and send-determinism in MPI HPC applications, let us define more formally these notions.

III. Formal definition of determinism types in parallel computation

Characterizing MPI HPC applications with regards to the determinism of their communication patterns requires a rigorous definition of the introduced notions: determinism and send-determinism. We first give the formal definition of communication determinism and then we give the formal definition of send-determinism. These definitions will be used as the basis for the analysis of the communication patterns in the MPI HPC 27 applications considered in this work. In the following, we focus only on the determinism in communication operations and do not address determinism of the entire computation.

We model a parallel computation as consisting of a finite set of processes and a finite set of channels connecting any (ordered) pair of processes. Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order sent.

We assume we have a set P of n processes. Each process $p \in P$ is defined by a set of states C_p, and initial state c_0^p, a set of events V_p and a partial transition function $\mathscr{F}_p : C_p \times V_p \longrightarrow C_p$; if $p \neq q$ then $V_p \cap V_q = \emptyset$. A state $c \in C_p$ is *final* if there is no valid transition from state c.

A channel is modeled as an unbounded queue. A *send* event $send(p, q, m) \in V_p$ adds message m to the tail of channel (p, q); a *receive* event $receive(q, p, m) \in V_q$ deletes the message m from the head of channel (p, q).

A computation consists of a sequence $E = e_1, e_2, \ldots$ of events. A system configuration consists of a vector of states $C = \{c_i, 1 \leq i \leq n, , c_{i,j}, 1 \leq i, j \leq n\}$ where c_i is the state of process i and $c_{i,j}$ is the state of channel (i, j). The sequence of configurations resulting from execution E is defined as follows.

The initial configuration of the system has each process in its initial state and each channel is empty.

Let $C^i = \{c_p^i, c_{p,q}^i\}$ be the configuration at step i. Assume that $e_i \in V_p$. Then,

$$c_q^{i+1} = \begin{cases} c_q^i & \text{if } q \neq p \\ \mathscr{F}_p(c_p^i, e^i) & \text{if } q = p \end{cases}$$

In addition, if $e_i = send(p, q, m)$ then message m is appended to channel (p, q) and if $e_i = receive(q, p, m)$ then message m is deleted from channel (p, q).

The execution is *valid* if

- $\mathscr{F}_p(c_p^i, e^i)$ is defined at each step
- If the event $receive(q, p, m)$ occurs then message m is at the head of channel (p, q).
- The last state of each process is final.

We denote by $\mathscr{E}$ the set of valid executions. For each execution E we denote by $E|p$ the subsequence of E consisting of events in V_p and by $C(E)|p$ the sequence of states of process p after each event in $E|p$. Then

Definition 1: A parallel computation is *deterministic* if, for each p, $C(E)|p$ is the same for any $E \in \mathscr{E}$.

The computation is *communication-deterministic* if, for each p, $E|p$ contains the same subsequence of communication events (sends and receives) for any $E \in \mathscr{E}$.

The computation is *send-deterministic* if, for each p, $E|p$ contains the same subsequence of send events for any $E \in \mathscr{E}$.

This abstract model can be related to MPI programs as follows:

Program transitions are modeled by non-communication events; if the program is determinisitc, then each state is associated with a unique valid event.

The execution of an MPI send is a transition caused by the event that appends the sent message to the channel connecting the sender to the receiver. The transition function can be defined so that this event is the only valid event for a given process state.

The arrival of a message is modeled by a receive event. One can think of this event as moving a message from the communication channel into a process buffer – the process state is changed to represent the new buffer content. The actual execution of the MPI receive operation is an internal program transition.

Before analyzing the nature of communication determinism in HPC applications considered in this study, we introduce the methodology as well as the applications.

IV. Methodology and analyzed Applications

The methodology followed in this study consisted in a static analysis of the application codes. While automatic analysis would be needed for analyzing a large number of applications, we decided to use a manual analysis. To analyze the nature of the determinism in these applications, based on the formal definitions of communication-deterministic and send-deterministic program, we decomposed each of them as a sequence of computation and communication patterns (computation patterns are deterministic). For every application, we analyzed the deterministic nature of all communication patterns and characterize the communication determinism of the application as following: 1) an application is communication-deterministic if and only if it contains only communication-deterministic communication patterns, 2) an application is send-deterministic if and only if it contains at least one send-deterministic communication pattern and no non-deterministic communication pattern, 3) an application is non-deterministic if and only if it contains at least one non-deterministic communication pattern. We considered 27 applications, benchmarks or kernels that we consider representative of a large fraction of applications run on production platforms. They are well recognized benchmarks, kernels used by many applications, codes of general science applications or they will serve for the evaluation of the next generation supercomputers (of 10+ Petaflops). Several codes are mixing Fortran 90 and MPI. We assume in the following that Fortran 90 is not a source of non determinism in the order of message reception.

The first set of applications belongs to the NERSC-6 benchmarks [14]. The NERSC center serves 3,000 scientists throughout the USA working on problems in combustion, climate modeling, fusion energy, materials science, physics, chemistry and computational biology. The suite consists of seven applications, spanning a wide range of science disciplines, algorithms, concurrencies and scaling methods. In this study, we considered the following applications: CAM, GTC, IMPACT, MAESTRO, PARATEC. CAM is the dominant computational code of the fully coupled CCSM3 climate model and is considered as one of the most demanding codes of the climate workload. CAM calculations relies on spectral (FFTs) and structured grids Methods. It uses many collective communications as well as blocking and non blocking point-to-point communications. GTC is used for fusion energy research, like for studying neoclassical and turbulent transport in tokamaks and for investigating hot particle physics. It uses particle and structured Grids methods. Its code includes collective communications, MPI_SendRecv() and point-to-point communication with the use of wildcards. IMPACT is used for accelerator sciences and represents typical beam dynamics simulation workload. IMPACT code was used for the design of many linear accelerators. It relies on spectral (FFTs), particle and structured grids methods. IMPACT mainly uses point-to-point communications in blocking and non-blocking versions. MAESTRO is an astrophysics code used to simulate the long time evolution leading up to a supernova explosion. It relies on explicit and implicit solvers and uses sparse

linear algebra, structured and unstructured grids methods. MAESTRO uses BoxLib, a library of Fortran90 modules that facilitates development of block-structured finite difference algorithms. It uses collective communications as well as non-blocking point-to-point communications. PARATEC (PARAllel Total Energy Code) performs quantum mechanical total energy calculations. It represents typical material science workloads run at NERSC. It uses dense linear algebra, spectral methods (FFTs) and structured grids methods. PARATEC uses collective communications, blocking and non-blocking point-to-point communications.

The second set of applications is included in the Sequoia Benchmarks [15]. This suite of codes is an evolution of the ASCI Purple benchmarks. Several codes have been rewritten and added to the suite. The Sequoia supercomputer will run codes for predicting stockpile performance. We analyzed the five application codes from the suite: Sphot, UMT, AMG, IRS and lammps. We also analyzed IOR, an I/O benchmark using MPI calls. SPhot implements Monte Carlo photon transport on a small, 2D structured mesh. Its algorithm is embarrassingly parallel. UMT performs 3D, deterministic, multi-group, photon transport on an unstructured mesh. It uses a transport algorithm solving the first-order form of the time-dependent Boltzmann transport equation. AMG uses algebraic multi-grid algorithms to solve large, sparse linear systems derived from implementing physics simulations on unstructured or block structured meshes. IRS iteratively solves a 3D radiation diffusion equation set on a block-structured mesh. It represents an important physics package representative of computation and memory access patterns used in several production physics packages. lammps can simulate a wide variety of different particle systems. lammps uses nearest neighbors and reduction operations. SPhot, UMT, AMG IRS and lammps use collective communications, blocking and non-blocking point-to-point communications. IOR is used for testing the performance of parallel filesystems from various HPC access patterns. We will discuss the nature of its specific communication pattern in next section.

The third set of applications concerns QCD calculation and is part of the USQCD suite. QCD simulations play an important role to better understand the fundamental laws of physics. USQCD suite contains software enabling high performance lattice quantum chromodynamics computations across a variety of architectures. Lattice QCD calculations allow understanding the results of particle and nuclear physics experiments in terms of QCD, the theory of quarks and gluons.The application packages used by USQCD are: the Chroma Code, the Columbia Physics System (CPS), FermiQCD and the MIMD Lattice Collaboration (MILC) Code. We analyzed CPS and MILC. CPS is a large set of lattice QCD codes that are essential for QCD applications. CPS uses collective MPI communications, blocking and non-blocking point-to-point communications. The MILC Code is a set of codes developed by the MIMD Lattice Computation (MILC) collaboration for doing simulations of four dimensional lattice gauge theory on parallel machines. MILC uses collective MPI communications, blocking and non-blocking point-to-point communications with wildcards.

SPECFEM3D [16] is a software package that simulates southern California seismic wave propagation. SPECFEM3D won the Gordon Bell Award for Best Performance at the ACM/IEEE SuperComputing'2003. The package contains several software including a mesher and a solver. SPECFEM3D uses a new method introduced by its authors for the calculation of synthetic seismograms in 3-D earth models: the Spectral Element Method (SEM). SPECFEM3D uses collective communications and MPI_SendRecv() where source and destination and tag parameters are deterministic.

We analyzed another geophysics application, RAY2MESH [17], because it is available in two versions: a SPMD version and a Master-Worker version. RAY2MESH is a high performance set of software tools for seismic tomography aiming to compute a realistic geophysic Earth model. RAY2MESH uses massive quantities of seismic waves records to ray-trace waves propagation inside a global Earth mesh with thin cells. The two versions use different MPI communication patterns. The Master-Worker version uses a pull model of scheduling.

In terms of small benchmark codes, we analyzed the NAS parallel Benchmarks NPB 3.3 [18]. These programs are derived from computational fluid dynamics (CFD) applications to help evaluate the performance of parallel supercomputers. The benchmark suite consist of five kernels and three pseudo-applications. In this study, we analyzed LU, MG, BT, SP, CG, EP; FT and DT. These programs use a large variety of MPI function calls including collective communications, blocking and non-blocking point-to-point communications.

In addition to applications run in production center, we analyzed several kernels used in many applications: 1) SUMMA, the matrix product kernel of the ScaLAPACK, 2) Linpack, the kernel used for the top500, 3) a Jaccobi kernel, 4) a nbody kernel.

V. Examples of communication patterns

In this section, we present the most common communication patterns found on analyzed applications. We focus only on patterns that are not communication-deterministic. This includes send-determinitic and non-derterministic communication patterns.

A. Send-deterministic communication patterns

One of the most common send-deterministic patterns uses a sequence of *MPI_Irecv* and *MPI_Isend* followed by an *MPI_Waitall*. This pattern is send-deterministic because an Irecv can complete before another one that was posted before. But the fact that the source is known ensures that every data is received where it should be (there is no risk of confusion), for example:

```
for(i=0; i<nb; i++)
   MPI_Irecv(T[i],..., i, ...);
/*  where i is the source */
   MPI_Isend(x, ...,  i, ...);
/* where i is the destination */
MPI_WaitAll(2*nb, ...);
```

Note that there may be some variations, for example an *MPI_Send* instead of an *MPI_Isend*, or *n*MPI_Wait* instead of *MPI_Waitall* (where n is the number of Irecv). Another pattern, similar to the former, contains *MPI_WaitANY* instead of *MPI_WaitAll* or *MPI_WAIT*. It is found in the Sequoia lammp benchmark. *MPI_Waitany* waits for completion of one over several requests.

```
 for(i=0; i<nb_recv; i++)
    MPI_Irecv(T[i], ,i..,...);
 for(i=0; i<nb_send; i++)
    MPI_Send( ..., i, ...);
 for(i=0; nb_recv; i++)
    MPI_Waitany(...)
```

In some applications (USQCD-MILC), *MPI_Irecv* may have an *ANY_SOURCE* parameter as sender:

```
for(i=0; i<nb_recv; i++)
    MPI_Irecv(R[i], ...,
    ANY_SOURCE, tag[i]...);
for(i=0; i<nb_send, i++)
    MPI_Isend(S[i], ..., proc,
    tag[i], ...);
mpi_Waitall(...)
```

where $tag[i]$ is the communication tag. In this application, the tag is different for every sender. This tag avoids confusions since every communication has a different tag.
The tag can also be *ANY_TAG*, as in Sequoia IOR (with an *MPI_Recv*):

```
 if(rank==0)
   for(i=0; i<nb; i++)
      {
       MPI_Recv(hostname, ...,
       ANY_SOURCE, ANY_TAG,...);
       if(hostname == localhost)
       count ++;
      }
 else MPI_Send(localhost, ..., 0, 0, ...);
 //A global communication
 MPI_Bcast(&count, 1, mpi_INT, 0, comm)
```

This pattern counts the number of processes running on the same host (hosting the process with rank 0). The reception order has no impact on the final count. Thus we consider this pattern as send-deterministic. Note however, that there is another condition to satisfy in this specific pattern: the number of processes hosted with process rank 0 should be deterministic for a given set of input parameters and execution parameters.
A subtle example of send-deterministic pattern using *ANY_SOURCE* and *ANY_TAG* is described below:

```
if(rank ==0)
  for(i=0; i<nb_procs; i++)
   {
    MPI_Recv(T[i], ..., ANY_SOURCE, ANY_TAG,...);
   }
else MPI_Send(T[rank], ..., 0, 0, ...);
MPI_Barrier(...);
sort(T);
```

In that case, there is no way to ensure that the content of array T after the *MPI_Barrier()* is the same across several executions. Note however that in this application the reception order has no importance if we extend the notion of communication pattern to include operations on receive buffers preceding their utilization to influence the rest of the system. This is because: 1) the result is never sent and 2) just after the loop, T is sorted and will always lead to the same result (for a given set of input parameters). So the subtleness here is on the definition of the "communication pattern".

B. Non-deterministic communication patterns

An example of non-deterministic patterns is shown below (from Sequoia AMG benchmark):

```
for(i=0; i<n; i++)
  MPI_Irecv(... , tag2, ...)
for(i=0; i<n; i++)
  MPI_Isend(..., tag1, ...)
...
while()
 {
  MPI_IProbe(ANY_SOURCE, tag1,
  flag, status)
  while(flag ==true)
    {
     proc = &status.mpi_SOURCE;
     MPI_Recv(x, ..., proc, tag1, ...);
     //modify x
     MPI_Send(x, ..., proc, tag2, ...);
    }
 }
```

The non-determinism is due to the *MPI_IProbe* with the *ANY_SOURCE* wilcard. *MPI_IProbe* checks if there is an incoming messages from any source with tag1. If so, an *MPI_Recv* is posted. So, receive order is not deterministic. And since the following send depends on the receive, sends may also occur in a different order

from one execution to another. This does not respect one of the send-determinism conditions (for one sender, messages are always sent in the same order).

Another example of non-deterministic communication pattern is used in the Master/Worker implementation of the Ray2mesh application.

```
if(id == root)
 for(i=1; i<nb_procs; i++)
 {
  MPI_Recv(x,..., ANY_SOURCE, .., &status);
  MPI_Send(y, ..., status.MPI_SOURCE);
 }
else
  {
   MPI_Send(x, ..., root, ...);
   MPI_Recv(y, ..., root, ...);
  }
```

The Master-Worker version of Ray2Mesh implements a first serve first come scheduling where workers pull tasks from the master. A faster worker will get more tasks. So the physical time is used in this program to control the distribution of tasks to the worker. This is representative of many Master-Worker application implementations. Once the root process receives a request from a worker, it sends a task to the worker. The non-determinism is due to *MPI_Recv(..., ANY_SOURCE, ...)* and the send that depends on it. This program is not send-deterministic because the send order fundamentally depends on the non-deterministic receive order.

VI. Results analysis

In this section, we present the result of our communication determinism analysis over 27 HPC applications. Table 1 presents, for every application, the diversity of used communication patterns and the class to which every application belongs.

From these results, we can draw some conclusions. First very few applications have non-deterministic communication patterns. This is an important result since this makes relevant the research for new fault tolerant protocols exploiting the determinism in HPC applications communication patterns. Second a significant portion of the applications have send-deterministic communication patterns where programers introduce more asynchrony in their applications. We believe that this kind of applications will become prevalent in the near future because the multi-core trend pushes architectures and algorithms for more parallelism and less synchrony. As a consequence, it may be irrelevant to design fault tolerant protocols uniquely for applications featuring deterministic communication patterns. Third, some applications, like Sequoia-AMG present a high degree of communication determinism. However because they have some non-deterministic communication patterns, fault tolerant protocols designed for communication-deterministic and send-deterministic communication patterns will not work for these applications. If the number of non-deterministic communication patterns is small, like for Sequoia-AMG, then it may be worth to reprogram these communication patterns to turn then communication-deterministic or send-deterministic. Fourth, a large portion of the applications use collective communications. Some applications are using them extensively. In this study, we have considered their internal implementation as communication deterministic. If there is a need to inject more asynchrony and relax determinism for performance purpose, our advise if to use send-deterministic implementations and avoid non-deterministic ones. This will make sure that specific fault tolerant protocols designed for communication-deterministic and send-deterministic communication patterns stay relevant for a large fraction of applications.

VII. Conclusion

In this paper we explored the determinism nature of communication patterns in a large variety of HPC applications representative of production workloads. This research was motivated by the fact that better fault tolerant protocols could be designed if some form of determinism can be exploited in communication patterns. We analyzed 27 applications from different benchmark and application suites. As we expected, most of HPC applications feature a high degree of determinism in their communication patterns: they are either communication deterministic or send-deterministic. Very few applications have non-deterministic communication patterns. This result is significant since it makes relevant the research for new fault tolerant protocols leveraging the determinism in HPC application communication patterns. Apart from the design of new fault tolerant protocols, there are several perspectives of this research: how to turn non-deterministic communication patterns into more deterministic ones while maintaining performance? New developments in MPI allow several threads to make communication calls concurrently. Applications using this possibility will have less deterministic communication patterns compared to communication-deterministic and send-deterministic communication patterns. However, the outcome of any communication pattern may still be deterministic regardless of the order of communication operations inside each pattern. We will analyze and characterize this form of determinism and explore new fault tolerance protocols from it.

References

[1] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. *Fault-Tolerant Computing, International Symposium on*, 0:48, 1999.

Applications	Collective communication calls	Send-Deterministic patterns	Communication-Deterministic patterns	non-Seterministic	Application Class
ScaLAPACK SUMMA	x	0	x	0	Deterministic
SP	13	1	6	0	Send-deterministic
BT	12	1	4	0	Send-deterministic
LU	x	0	x	0	Communitation-Deterministic
CG	x	0	x	0	Communication-Deterministic
MG	x	0	x	0	Communication-Deterministic
FT	x	0	x	0	Communication-Deterministic
EP	x	0	x	0	Communication-Deterministic
DT	x	0	x	0	Communication-Deterministic
Nbody	x	0	x	0	Communication-Deterministic
USQCD-CPS	2	31	0	0	Send-deterministic
USQCD-MILC	1099	517	111	0	Send-deterministic
Sequoia-UMT	52	1	1	0	Send-deterministic
Sequoia-lammp	867	4	33	0	Send-deterministic
Sequoi-IOR	18	2	0	0	Send-deterministic
Sequoia-AMG	41	76	4	1	Non-deterministic
Sequoia-Sphot	7	7	1	0	Send-deterministic
Sequoia-IRS	x	x	x	0	Send-deterministic
NERSC-CAM	700	61	4	0	Send-deterministic
NERSC-IMPACT	0	12	97	0	Send-deterministic
NERSC-MAESTRO	21	9	9	0	Send-deterministic
NERSC-GTC	x	0	x	0	Communication-Deterministic
NERSC-PARATEC	x	0	x	0	Communication-Deterministic
SpecFEM3D	x	0	x	0	Communication-Deterministic
Jacoby	x	0	x	0	Communication-Deterministic
Ray2mesh	7	2	0	0	Send-deterministic
Ray2mesh-MS	4	2	1	3	Non-deterministic

TABLE I: Communication determinism in parallel applications. 'x' is used to denote the presence of the pattern in the application. Numbers give the total occurrence of the patterns in the application

[2] Aurelien Bouteiller, Thomas Herault, Geraud Krawezik, Pierre Lemarinier, and Franck Cappello. Mpich-v project: A multiprotocol automatic fault-tolerant mpi. *International Journal of High Performance Computing Applications*, 20(3):319–333, 2006.

[3] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 471–478, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 03 2007.

[5] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[6] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[7] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *FJCC*, pages 1150–1158. IEEE Computer Society, 1986.

[8] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[9] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, 1987.

[10] Aurelien Bouteiller, Thomas Ropars, George Bosilca, Christine Morin, and Jack Dongarra. Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure recovery. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, New Orleans, USA, 2009.

[11] Francesco Quaglia, Roberto Baldoni, and Bruno Ciciani. On the no-z-cycle property in distributed executions. *Journal of Computer and System Sciences*, 61(3):400–427, 2000.

[12] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[13] Franck Cappello, Amina Guermouche, Thomas Herault, and Marc Snir. Revisiting fault tolerant protocols for hpc applications. *Technical Report of the INRIA-Illinois Joint Laboratory on Petascale Computing (TR-JLPC-09-02)*, 2009.

[14] Katie Antypas, J. Shalf, and H. Wasserman. Nersc-6 workload analysis and benchmark selection process. *Technical Report of the Lawrence Berkeley National Laboratory 1014E*, 2008.

[15] Lawrence Livermore National Laboratory. https://asc.llnl.gov/sequoia/benchmarks/. 2009.

[16] Laura Carrington, Dimitri Komatitsch, Michael Laurenzano, Mustafa Tikir, David Michéa, Nicolas Le Goff, Allan Snavely, and Jeroen Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62 thousand processor cores. *Proceedings of the ACM/IEEE Supercomputing SC'2008 conference*, pages 1–11, 2008.

[17] Marc Grunberg, Stéphane Genaud, and Catherine Mongenet. Seismic ray-tracing and earth mesh modeling on various parallel architectures. *J. Supercomput.*, 29(1):27–44, 2004.

[18] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, New York, NY, USA, 1991. ACM.