University of Delaware
Department of Computer & Information Sciences
Technical Report 2013/002

Title: **Dynamic Barrier Relaxations for Explicit Stencil Computations**
Authors: Adam Hammouda, Andrew R. Siegel, and Stephen F. Siegel
Status: *Submitted for publication*

Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark DE 19716
USA
`http://vsl.cis.udel.edu`

# Dynamic Barrier Relaxations for Explicit Stencil Computations

ADAM HAMMOUDA and ANDREW R. SIEGEL, Argonne National Laboratory
STEPHEN F. SIEGEL, University of Delaware

Next generation HPC computing platforms are likely to be characterized by significant, unpredictable nonuniformities in execution time among compute nodes and cores. These inherent load imbalances are expected to arise from a variety of sources—manufacturing discrepancies, dynamic power management, runtime component failure, OS jitter, software mediated resiliency, and TLB/- cache performance variations, etc. It is well understood that existing algorithms with frequent points of bulk synchronization will perform relatively poorly in the presence of these performance fluctuations. Thus, re-casting classic bulk-synchronous algorithms into more asynchronous, course grained parallelism is a critical area of research for next generation computing. In the present analysis we propose a robust class of parallel algorithms for explicit stencil computations in the presence of such non-uniformities in process execution time. These algorithms are benchmarked using the 2D heat equation as a model problem, and they are tested in the presence of simulated nonuniform computational noise. The performance is compared to a classic bulk synchronous implementation of the model problem.

Categories and Subject Descriptors: B.8 [**Hardware**]: Performance And Reliability; B.8.1 [**Performance And Reliability**]: Reliability, Testing, and Fault-Tolerance; C.4 [**Computer Systems Organization**]: Performance Of Systems—*Fault tolerance*; *Modeling techniques*; *Performance attributes*; *Reliability*; *availability*; *and serviceability*; F.1 [**Theory of Computation**]: Computation By Abstract Devices; F.1.2 [**Computation By Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions*; *Invariants*; *Pre- and post-conditions*; *Specification techniques*; G.1 [**Mathematics of Computing**]: Numerical Analysis; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear systems (direct and iterative methods)*; *Sparse*; *structured*; *and very large systems (direct and iterative methods)*; G.1.3 [**Numerical Analysis**]: Partial Differential Equations—*Finite difference methods*; *Finite element methods*; *Iterative solution techniques*; I.6 [**Computing Methodologies**]: Simulation and Modeling; I.6.5 [**Simulation and Modeling**]: Model Development; I.6.5 [**Simulation and Modeling**]: Types of Simulation—*continuous*; *parallel*

General Terms: Algorithms, Reliability, Performance, Experimentation, Verification

Additional Key Words and Phrases: PDE, stencil, heat equation, relaxed BSP, BSP, bulk synchronous parallelism, barrier relaxation, resilience, dynamic stencil, dynamic barrier relaxation, dynamic ghost zone, dynamic ghost zone optimization, course grained parallelism

**ACM Reference Format:**
Adam Hammouda, Andrew R. Siegel, and Stephen F. Siegel, 2013. Dynamic Barrier Relaxations for Explicit

---

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.
This research used the University of Delaware's *Chimera* computer, funded by U.S. National Science Foundation award CNS-0958512. S.F. Siegel was supported by NSF award CCF-0953210.
Author's addresses: Adam Hammouda and Andrew Siegel, Argonne National Laboratory, 9700 South Cass Ave, Bldg 240, Argonne, IL 60439-4847; Stephen Siegel, Verified Software Laboratory, Department of Computer & Information Sciences, University of Delaware, Newark, DE 19716.

DOI:http://dx.doi.org/10.1145/0000000.0000000

Stencil Computation. *ACM Trans. Parallel Comput.* 0, 0, Article 00 ( 2014), 29 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

## 1. INTRODUCTION

Many common HPC algorithms are implemented using a *bulk synchronous* model, with frequent synchronization points (either explicit or implicit) among processes and between phases of the algorithm [Valiant 1990; Gropp et al. 1999; Kjolstad and Snir 2010]. Alternative task based, coarse grained algorithmic formulations have gained much less traction over the past 20 years. More importantly, though, there has been relatively little incentive to pursue such approaches—the bulk synchronous paradigm has performed relatively well on existing leadership class compute architectures, where process execution times have been relatively uniform in the presence of uniform workloads, and high local work to communication volumes have been effective at hiding global communication latency to achieve good scalability.

As computer architectures evolve on the path to exascale computing, conditions no longer favor a bulk synchronous approach. This is the result of a large number of underlying phenomena which taken together are expected to manifest themselves increasingly as irregular, uneven process execution times even for identical workloads. This inherent process non-uniformity is likely to lead eventually to extreme degradation in the performance of algorithms that require frequent global synchronization among processing elements.

In the present analysis we focus on the algorithmic impact of these intermittent slowdowns rather than the underlying causes. Our particular focus is explicit timestepping algorithms for standard stencil computations, such as those employed in shock capturing hydrodynamic simulations [Fryxell et al. 2000]. We discuss below related resiliency research for such stencil based algorithms. To put our work in its proper context though, we first briefly discuss some of the key general research issues in the related fields of resilience and fault tolerance.

### 1.1. Related resilience research

With mean time to failure (MTTF) soon expected to be far smaller than the overhead time required for existing fault tolerance techniques [Bergman et al. 2008; Amarasinghe et al. 2009; et. al. 2012; Beckman et al. 2006; Hoefler et al. 2010], resilience and fault tolerance has become front and center in many aspects of exascale research [Brown et al. 2010; Heroux and Geist 2013; Bergman et al. 2008; Amarasinghe et al. 2009; et. al. 2012; Beckman et al. 2006; Hoefler et al. 2010]. Such research aims to address a wide variety of issues, including the increasing cost of global communication barriers [Ghysels et al. 2013; Ashby et al. 2012], OS Noise [Beckman et al. 2006; Hoefler et al. 2010], and the latency induced by correcting faults themselves [Liu et al. 2008], to name only a few of the better understood problems. A full spectrum of anticipated resilience challenges was summed up in the 2010 DOE workshop report on cross-cutting technologies at exascale [Brown et al. 2010].

There are a number of consequences of these phenomena either directly or indirectly on the application side. The focus of the present analysis, non-uniformities in process runtimes, represents a shift from what has historically been the predominant focus of fault tolerance research—data integrity and reliability of solution accuracy. In such cases, recovery from faults has been limited to questions of *state*, whether addressed from a systems [Elnozahy et al. 2002; Plank et al. 1998; Lu et al. 2013] or an applications perspective [Huang and Abraham 1984; Chen and Dongarra 2006; Bosilca et al. 2009; Gunnels et al. 2001; Roy-Chowdhury et al. 1996; Geist and Engelmann 2002; Oboril et al. 2011]. Alternative solutions have been put forward recently to integrate these approaches, by allowing software frameworks to communicate data reliability

and performance information dynamically to hardware and vice versa [Gupta et al. 2009; Berzins et al. 2010; Berzins et al. 2012; Fujita et al. 2013].

### 1.2. Bulk Synchronous Parallelism

For our analysis, we take an application-centric approach and employ a relaxed bulk synchronous parallel (BSP) strategy for mitigating the impact of process non-uniformity on time to solution. In its most general form relaxed BSP is a bridging model between hardware and software, proposed in response to Valiant's seminal work on the BSP model [Kim et al. 1998; Fahmy and Heddaya 1996; Valiant 1990]. Relaxed BSP is a way of thinking about program design on every layer of the runtime stack which seeks to minimize synchronization constraints inherent in parallel programming. Relaxing synchronization barriers has been of interest to algorithmic researchers long before the formalization of the bridging model [Chazan and Miranker 1969], and it has recently been employed as a strategy for optimization on state of the art machines [Geist and Engelmann 2002; Ghysels et al. 2013].

While BSP has not been typically applied toward the problem of non-uniformities in process runtime, the relaxed BSP approach has been used ubiquitously outside of the fault tolerance community. Research in stencil-based solver optimization has for all intents and purposes used this way of thinking about their codes for decades, although it has never articulated the connection as far as we know. We thus review some of what has been done in this research, and attempt to recast its vocabulary in the context of barrier relaxation.

### 1.3. Stencil-based solvers

Stencil computations have long been optimized to relax synchronization points. However the optimizations are most often cast as communication avoiding, communication overlapping [Demmel et al. 2008], or masking [Avizienis et al. 2004]. In this context, non-uniformities in process runtimes have not been of interest to application developers. This is because research has been focused on maximizing data locality of a stencil to exploit cache and memory hierarchy size and structure as well as off chip bandwidth constraints in the case of single process optimizations [Rosser 1998; Datta 2009; Wonnacott 2000; Douglas et al. 2000; Strout et al. 2001; Kamil et al. 2006; Rivera and Tseng 2000; Mccalpin and Wonnacott 1999; Frigo and Strumpen 2005]. In the case of multi-process optimizations the interest lay in reducing communication costs [Rosser 1998; Chronopoulos and Gear 1989; Wonnacott 2000; Ding and He 2001; Demmel et al. 2008; Ballard et al. 2012; Georganas et al. 2012]]. A great deal of research applicable to barrier relaxation has also been carried out in the development of formal techniques for compiler pre-processing of sequential codes for automatic parallelization, as well as in the development of pre-processing techniques for the optimization of existing parallel codes [Rosser 1998; Beletska et al. 2011; Beletska et al. 2010; Wonnacott 2000]. Whatever the motivation or vocabulary, all of these approaches have in one way or another aimed to tweak stencil computations in order to reduce synchronization. Therefore, we need to recast some of this work in a manner which employs the language of barrier relaxation, in order to illustrate both what we draw from in previous work, and from where we depart.

A general strategy employed in many studies of stencil optimization has been to add redundant computation for the purpose of reducing synchronization points of a stencil by some factor $t$ (known as a $t$-step reduction—a reduction factor understood in terms of time-steps) [Wonnacott 2000; Ding and He 2001; Demmel et al. 2008; Chronopoulos and Gear 1989]. $t$ in this case tends to be relatively small [Chronopoulos and Gear 1989; Demmel et al. 2008; Ding and He 2001], and in Demmel et al. [2008] the as-

sumption that $t \ll \frac{n}{p}$ for most computations is explicitly stated (where $n$ :=number of grid points in the stencil, and $p$ :=the number of processes). This approach can be thought of as a $t$-step barrier relaxation, leaving $\frac{r}{t}$ barriers yet unrelaxed, where $r$ represents the total number of timesteps in the simulation. This approach is valuable, but tends to be designed primarily for applications involving very few steps—for example, the intermediate steps of a Krylov subspace method [Chronopoulos and Gear 1989; Demmel et al. 2008]. Other $t$-step relaxation techniques attain their synchronization reductions by allowing for the numerical precision of a stencil at intermediate timesteps to be inaccurate, in order to get to a final solution faster [Oboril et al. 2011; Geist and Engelmann 2002; Ding and He 2001]. Still other approaches have been taken to generate $t$ based on dynamically collected performance information [Allen et al. 2001; Meng and Skadron 2009].

Our interest in explicit stencil computations for real world physics applications makes small $t$-step relaxations impractical as a resilience technique. This is because our applications of interest will require at least many thousands or hundreds of thousands of steps to complete, with accompanying synchronizations at each step. If our goal is to reduce synchronizations, a few steps out of hundreds of thousands is hardly satisfactory. Further, the process non-uniformities of exascale machines have the potential to introduce delays in execution on an individual process on the order of hundreds or thousands of time-steps of an algorithm.

Our applications of interest are real world high fidelity physics codes which can require that the total number of steps are on the order of $n$ ($n$ :=number of grid points in a stencil) for a simulation to give us meaningful results. Moreover, inaccuracies at intermediate steps are unacceptable for many of these real world applications of stencils. In particular, coupled multi-physics applications where an accurate final solution of a system depends on the solution of a partnered physics at very small timescales (For example, a neutronics code for a nuclear reactor might be coupled with the heat diffusion and fluid dynamics of the reactor's cooling water). Finally, in addition to allowing for large $t$-step reductions in synchronization, we would like for $t$ to be generated dynamically in accordance with the magnitudes of process non-uniformity which may vary from system to system and experimental run to experimental run. Aforementioned approaches to this kind of dynamism have to some degree assumed that while hardware may not be consistent with its theoretical peak performance, it will none-the-less deliver consistent performance every time it is used. Therefore, dynamism has only been employed to mitigate deviations from the theoretical peak performance of a system [Meng and Skadron 2009], or to account for built in heterogeneity of computational units on a grid [Allen et al. 2001].

Our contribution in this paper is thus to develop and test a fault tolerance technique for the barrier relaxation of explicit stencil computations [1]. We experiment with this technique using the 2D heat equation on a 5-point stencil with artificially generated and nonuniform process slowdowns. Our dynamic barrier reduction allows for a $t$-step relaxation of synchronization points where it is feasible for $t$ to be on the order of $\frac{n}{p}$. Our results show that factors from $2 - 30$ in resilience speedup can be obtained for a broad range of synthetic noise profiles.

## 2. ALGORITHMS

In this section we describe a "classic" and a new "resilient" algorithm for solving the 1-dimensional heat equation. Both algorithms extend trivially to a 2-dimensional form using a "slab" (or "striped") spatial decomposition. We have used the 2D versions in

[1] Where explicit is meant to refer to those computations of the general form $x_{n+1} = Ax_n$

| Symbol | Meaning |
|---|---|
| $u_i^n$ | the value of the discrete solution at time $n$ and position $i$ |
| nprocs | the number of processes |
| pid | the process ID |
| nsteps | the number of time steps |
| nx | the total number of cells in the global problem; nx $\geq$ nprocs |
| nxl | the number of cells owned by this process (excluding ghosts) |
| left, right | left (resp., right) neighboring process; may be null |
| $f$ | function yielding new value of cell given old values of left, self, and right |
| $g$ | function taking process ID and local index and returning global index |
| data | the array used to store the data on this process |
| *postSend* | start a send operation and return handle |
| *postRecv* | start a receive operation and return handle |
| $\mathsf{snd}_l$, $\mathsf{rcv}_l$ | variable holding request handle for send (resp., receive) of left ghost |
| $\mathsf{snd}_r$, $\mathsf{rcv}_r$ | as above but for right ghosts |
| *complete* | predicate which determines if a communication has finished |
| $t_l$, $t_r$ | number of time steps completed at local left (resp., right) border |
| $t_m$ | number of time steps completed at local peak (middle region) |
| $x_l$ | left border of middle region; $x_l = t_m - t_l$ |
| $x_r$ | right border of middle region; $x_r = \mathsf{nxl} + 2 + t_r - t_m$ |
| $\bar{t}$ | $t\%2$ (0 if $t$ is even, else 1) |
| $\mathsf{RQ}_{i,j}$ | queue of receive records for receives issued by process $i$ to process $j$ |
| $\mathsf{SQ}_{i,j}$ | queue of send records for sends issued by process $i$ to process $j$ |
| $\mathsf{sc}_{i,j}$ | count of send requests issued by process $i$ to $j$ |
| $\mathsf{rc}_{i,j}$ | count of receive requests issued by process $i$ to $j$ |
| $S(i,j,k)$ | send request handle for send issued by process $i$ to $j$ with ID $k$ |
| $R(i,j,k)$ | receive request handle for receive issued by process $i$ to $j$ with ID $k$ |
| $r$.id | the ID number of a send or receive record $r$ |
| $r$.buf | the buffer reference of a send or receive record $r$ |
| $r$.comp | flag indicating whether the send or receive operation has completed |
| $r$.data | the data extracted from the send buffer for a send record $r$ |

Fig. 1. Table of notation

our experiments (Section 5), but in this and the following section we restrict our discussion to the 1D problem for simplicity. The Table of Notation (Figure 1) gives a brief definition of all symbols used in Sections 2 and 3.

### 2.1. Problem definition

*2.1.1. Continuous problem and discretization.* The differential form of the 1D heat equation is

$$\frac{\partial u}{\partial t} = -\alpha \frac{\partial^2 u}{\partial x^2}. \tag{1}$$

Its forward time center spaced (FTCS) discretization is

$$u_j^{n+1} = f(u_{j-1}^n, u_j^n, u_{j+1}^n) \stackrel{\text{def}}{=} u_j^n + \frac{\alpha \Delta t}{\Delta x^2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n), \tag{2}$$

This is a 3 point stencil, where the value of $u$ is given at discrete point $j$ at time $n$ in $u_j^n$, where $0 \leq j < \mathsf{nx}$, $0 \leq n \leq \mathsf{nsteps}$.

*2.1.2. Data distribution.* Let the value of $u$ be given at nx different points; nx is referred to as the *global problem size*. Consider a set of nprocs $\geq 1$ processes. Each process

can be identified by a unique integer process ID, pid, where $0 \leq \mathsf{pid} < \mathsf{nprocs}$. Form a *block decomposition* of the array $u$ over the processes [Scott et al. 2005], by choosing integers $0 = m_0 < m_1 < \cdots < m_{\mathsf{nprocs}} = \mathsf{nx}$. Process $i$ "owns" the contiguous block of cells $m_i, \ldots, m_{i+1} - 1$. The number of cells owned by process $i$ is $m_{i+1} - m_i$ and will be stored in a local variable nxl. To avoid bookkeeping complexities, we have assumed $\mathsf{nxl} \geq 1$ on every process (however, the value of nxl need not be the same on every process).

The main data on a process will be stored in an array of length $\mathsf{nxl} + 2$, with indexes $0$ and $\mathsf{nxl} + 1$ holding ghost cells. We let $g(\mathsf{pid}, j)$ denote the global index corresponding to the cell with local index $j$ on process pid. Note that this makes sense even for the ghost cells ($j = 0$ or $j = \mathsf{nxl} + 1$), as these also correspond to global cells. We say $g$ is *undefined* on cell $0$ of process $0$ and on cell $\mathsf{nxl} + 1$ of process $\mathsf{nprocs} - 1$.

We have chosen this fixed boundary condition for convenience. Others would do just as well, without affecting the essential characteristics of the algorithms.

*2.1.3. Message-passing model.* The algorithms use a simple form of *asynchronous nonblocking communication*. This is a refinement of the standard asynchronous message passing model in which the send and receive operations issued by a process may execute concurrently with the main thread of control in that process.

In the standard model, there is a queue of buffered messages for each ordered pair of processes $(i, j)$. A send operation issued by process $i$ to process $j$ enqueues a message on the corresponding queue; a receive operation dequeues a message.

In our nonblocking model, a call to *postSend* (resp., *postRecv*) starts a send (resp., receive) operation and returns immediately a *handle* to the ongoing operation. The function *complete* takes such a handle and returns *true* if the operation has completed, else *false*. If a send operation has completed, the message data has been completely copied out of the send buffer, so that buffer can be safely re-used; it does *not* necessarily indicate that the message has been received, only that it has been entered into a queue. The completion of a receive operation indicates the message data has been completely copied into the receive buffer, which can then be read by the process. It is erroneous for a process to access a send or receive buffer if the operation is not complete. A more formal description of the model is given in Section 3.1.

The function *postSend* takes two arguments: the PID of the destination process and a specification of the send buffer. In our case the buffer is always a single cell of array data and will be specified by an expression of the form $\&\mathsf{data}[a][b]$, where $a$ and $b$ are integer expressions. We will use this C notation for the "address of" an expression, as well as the notation "*p" to return the value stored at the location specified by an address $p$.

Function *postRecv* takes the PID of the source process and the address of the receive buffer. For both functions, if the PID argument is not in the range $[0, \mathsf{nprocs} - 1]$ the call is a no-op and returns null.

### 2.2. Classic nonblocking algorithm

Algorithm 1 encodes a well-known approach for computing (2), using nonblocking communication based on the data distribution described in Section 2.1.2 (cf. [Gropp et al. 1999, Sec. 4.9]). Each process executes a copy of the code shown. The two-dimensional array data can store two copies of the section of the array owned by the process: one for the previous time step and one for the current time step. We assume this array has been initialized with the initial values of $u$ stored in $\mathsf{data}[0]$. At each time step, the process initiates a ghost-cell exchange with its neighbors and then updates its inner region (cells $2 \ldots \mathsf{nxl} - 1$) concurrently with the exchange. Once the exchange completes, it updates the cells in positions $0$ and nxl (unless these are on the global boundary),

and proceeds to the next time step. The flipping of the bit $p$ ensures that on the next iteration, the current data just computed will become the previous data.

**ALGORITHM 1:** Classic nonblocking algorithm

**Uses** : data : array$[0..1][0..\mathsf{nxl}+1]$ of **real**; $p : \{0,1\}$; $t, j$ : **int**;
: $\mathsf{snd}_l, \mathsf{snd}_r, \mathsf{rcv}_l, \mathsf{rcv}_r$ : communication handles
**Requires** : $\mathsf{data}[0][j] = u^0_{g(\mathsf{pid},j)}$ for all $1 \le j \le \mathsf{nxl}$
**Ensures** : $\mathsf{data}[\overline{\mathsf{nsteps}}][j] = u^{\mathsf{nsteps}}_{g(\mathsf{pid},j)}$ for all $1 \le j \le \mathsf{nxl}$

```
1 p ← 0;
2 for t ← 1 to nsteps do
3     rcv_l ← postRecv(left, &data[0][0]); snd_l ← postSend(left, &data[0][1]);
4     snd_r ← postSend(right, &data[0][nxl]); rcv_r ← postRecv(right, &data[0][nxl + 1]);
5     for j ← 2 to nxl − 1 do data[1 − p][j] ← f(data[p][j − 1], data[p][j], data[p][j + 1]);
6     when complete(rcv_l) ∧ complete(snd_l) ∧ complete(snd_r) ∧ complete(rcv_r)
7         if g(pid, 1) ≠ 0 then data[1 − p][1] = f(data[p][0], data[p][1], data[p][2]);
8         if g(pid, nxl) ≠ nx − 1 then
          data[1 − p][nxl] = f(data[p][nxl − 1], data[p][nxl], data[p][nxl + 1]);
9         p ← 1 − p;
```

This solution is generally very effective. The overlap of communication (the ghost cell exchange) and computation (the interior update) often contributes significantly to performance. Moreover, since there is no explicit barrier in the loop, the algorithm allows for some—but not much—slack: at any time, the time steps of two neighboring processes can differ by at most 1. In an environment where every process is running at roughly the same speed, with no interruptions or delays, this slack is more than sufficient. However, if one process were to be delayed for many time steps, this would effectively delay all processes for nearly as many steps. As we will see, much more slack is possible.

### 2.3. The Resilient Algorithm

Algorithm 2 presents our new *resilient* algorithm. This algorithm has the same specification as the classic version: given the same inputs, both algorithms will eventually terminate with the same output. Both algorithms also use essentially the same amount of memory; in particular each stores $2(\mathsf{nxl}+2)$ solution values in each process. However they differ in fundamental ways. Most importantly, the classic version is *locally deterministic*: for a fixed initial condition, on any execution each process will follow the same sequence of state changes, regardless of how these are interleaved with actions from other processes. The resilient algorithm is *locally nondeterministic*: a process may take many possible paths from its initial to its final state.

In the classic version, the state of a process is determined by its current time step. At time step $i$, data holds the solution values for time steps $i$ and $i-1$. Each process simply moves through a sequence of states $s_0 \to s_1 \to \cdots \to s_{\mathsf{nsteps}}$ and then terminates.

In the resilient version, the state of a process is not determined by a single time step. Instead, the spatial domain is divided into three dynamically changing regions: the left, middle, and right regions. The state is specified by three integers: $t_l$, $t_m$, and $t_r$—the current time steps for the left-most cell, the middle region, and the right-most cell. At each state, there are up to 3 possible transitions to the next state.

We illustrate this organization with a concrete example in Figure 2. In the initial state (a), $t_l = 0 = t_r$ and $t_m = 1$. Cells $\mathsf{data}[0][1..\mathsf{nxl}]$ hold the initial values of $u$, exactly as in the classic version. The integer $x_l$ represents the border between the left and

middle regions, and $x_r$ the border between the middle and right regions. Both $x_l$ and $x_r$ are determined completely by $t_l$, $t_m$, and $t_r$, and are used only for convenience. Sends have been posted for the local boundary cells, and receives posted for the ghost cells.

Figure 2(b) shows an intermediate state that might arise in an execution. In this state, the left region encompasses three cells; the middle, four; and the right, two (including ghosts). The middle region, similar to the classic case, holds data for two consecutive time steps: times 2 and 3. The left region, on the other hand, holds data corresponding to two contiguous *diagonals* with slope 1 through the space-time plane. Specifically, position 0 holds the value at time step 0; position 1 holds steps 1 and 0; position 2 holds steps 2 and 1. Note that data$[0]$ always holds values for even time steps, data$[1]$ for odd time steps. The right region is dual and the data stored there corresponds to diagonals of slope $-1$.

From state (b), three transitions are possible: an update to the left region, an update to the middle region, or an update to the right region. The first and third are only enabled if the corresponding communication operations have completed. Assuming the left send and receive have completed, the result of a left update is illustrated in Figure 2(c). In the new state, the two left diagonals have moved up 1 unit, and the left-middle border has moved one unit to the left. The new values are computed by starting at the bottom left point on the diagonal (where the receive just completed) and moving up and to the right. At each point in this loop, a new value is computed using the three cells immediately below and to the left, immediately below, and immediately below and to the right.

Figure 2(d) shows the result of applying a right update to state (c), and (e) shows the result of applying a middle-update to (d). The final state is shown in 2(f).

To make the relation of the algorithmic data to the mathematical problem precise, we introduce some auxiliary state variables. Given a fixed state, define integers $t(j)$ for $0 \le j \le \mathsf{nxl}+1$, by

$$t(j) = \begin{cases} t_l + j & \text{if } 0 \le j < x_l \text{ ($j$ is in the \textit{left region})} \\ t_m & \text{if } x_l \le j < x_r \text{ ($j$ is in the \textit{middle region})} \\ \mathsf{nxl} + t_r + 1 - j & \text{if } x_r \le j \le \mathsf{nxl}+1 \text{ ($j$ is in the \textit{right region})} \end{cases} \tag{3}$$

Then $t(j)$ is the number of time steps completed for the cell with local index $j$. For example, $t(0) = t_l$ is the number of left ghost cells received and processed, and $t(\mathsf{nxl}+1) = t_r$ is the number of right ghost cells received and processed. In the initial state, $t(0) = 0 = t(\mathsf{nxl}+1)$, as no ghost cells have been received, and $t(j) = 1$ for all $j$ in the middle region ($1 \le j < \mathsf{nxl}+1$).

We will see that whenever control in process pid is at the top of the *while* loop (line 9 of Algorithm 2), the expressions given in Figure 4 must hold. The first four invariants in that list specify the relationship between the values held in data and the discrete solution values $u_j^t$. For example, in the initial state, these formulas imply

$$\mathsf{data}[0][j] = u^0_{g(\mathsf{pid},j)} \qquad (1 \le j \le \mathsf{nxl}),$$

i.e., each process holds the initial values for its cells in data$[0]$; nothing is known about any other values of data. In the final state, we will have $t(j) = \mathsf{nsteps}+1$ and therefore

$$\mathsf{data}[\overline{\mathsf{nsteps}}][j] = u^{\mathsf{nsteps}}_{g(\mathsf{pid},j)} \qquad (1 \le j \le \mathsf{nxl}),$$

which is the desired postcondition.

## 3. ALGORITHM CORRECTNESS

In this section, we sketch a proof of correctness of Algorithm 2. Specifically, we show that given any input, any execution of the algorithm will terminate normally with the

**ALGORITHM 2:** Resilient algorithm: code executed by each process.

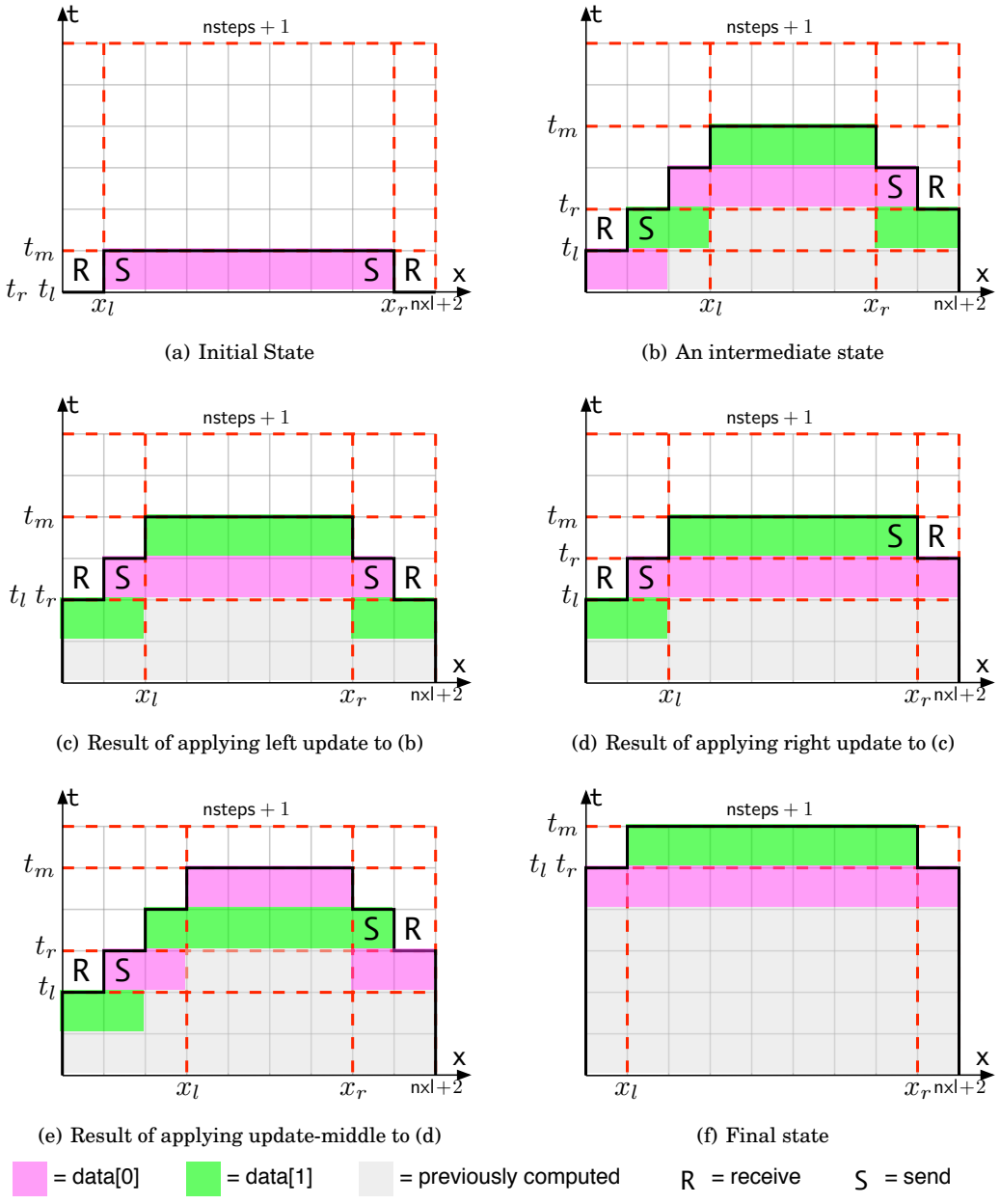
**Uses** : data : array$[0..1][0..\mathsf{nxl}+1]$ of **real**; $t_l, t_m, t_r, x_l, x_r, j$ : **int**;
: $\mathsf{snd}_l, \mathsf{snd}_r, \mathsf{rcv}_l, \mathsf{rcv}_r$ : communication handles
**Requires** : $\mathsf{data}[0][j] = u^0_{g(\mathsf{pid},j)}$ for all $1 \le j \le \mathsf{nxl}$
**Ensures** : $\mathsf{data}[\overline{\mathsf{nsteps}}][j] = u^{\mathsf{nsteps}}_{g(\mathsf{pid},j)}$ for all $1 \le j \le \mathsf{nxl}$

```
1  procedure update(p : {0,1}; j : int; v0, v1, v2 : real) is
2      if g(pid, j) ∈ {0, nx − 1} then return ;        /* don't update a global boundary cell */
3      data[p][j] ← f(v0, v1, v2);
4      if j = 1 ∧ t_l < nsteps ∧ (t_l < nsteps − 1 ∨ x_l ≠ 2) then snd_l ← postSend(left, &data[p][1]);
5      if j = nxl ∧ t_r < nsteps ∧ (t_r < nsteps − 1 ∨ x_r ≠ nxl) then
           snd_r ← postSend(right, &data[p][nxl]);
6  t_l ← 0; t_m ← 1; t_r ← 0; x_l ← 1; x_r ← nxl + 1;
7  rcv_l ← postRecv(left, &data[0][0]); snd_l ← postSend(left, &data[0][1]);
8  snd_r ← postSend(right, &data[0][nxl]); rcv_r ← postRecv(right, &data[0][nxl + 1]);
9  while t_m ≤ nsteps ∨ x_l > 1 ∨ x_r < nxl + 1 do choose
10     when t_l < t_m ∧ t_l < nsteps ∧ complete(rcv_l) ∧ complete(snd_l)    /* update left region */
11         rcv_l ← null; snd_l ← null; p ← 1 − t_l ;
12         if t_l < nsteps − 1 then rcv_l ← postRecv(left, &data[p][0]);
13         for j ← 1 to x_l − 1 do
14             update(p, j, data[1 − p][j − 1], data[1 − p][j], data[1 − p][j + 1]);
15             p ← 1 − p;
16         t_l ← t_l + 1; x_l ← x_l − 1 ;
17     when t_r < t_m ∧ t_r < nsteps ∧ complete(snd_r) ∧ complete(rcv_r)  /* update right region */
18         rcv_r ← null; snd_r ← null; p ← 1 − t_r ;
19         if t_r < nsteps − 1 then rcv_r ← postRecv(right, &data[p][nxl + 1]);
20         for j ← nxl downto x_r do
21             update(p, j, data[1 − p][j − 1], data[1 − p][j], data[1 − p][j + 1]);
22             p ← 1 − p;
23         t_r ← t_r + 1; x_r ← x_r + 1 ;
24     when t_m ≤ nsteps ∧ x_r ≥ x_l + 3                          /* update middle region */
25         p ← 1 − t_m ;
26         for j ← x_l + 1 to x_r − 2 do update(1 − p, j, data[p][j − 1], data[p][j], data[p][j + 1]);
27         t_m ← t_m + 1; x_l ← x_l + 1; x_r ← x_r − 1 ;
```

correct final values in data. To make this precise, we first present a formal execution model for programs using nonblocking communication. We then discuss the issue of *granularity* in this model, i.e., the question of which statements are considered atomic. We show that under reasonable assumptions, correctness is independent of the granularity chosen, so to simplify the remaining arguments we choose the coarsest model. We use this model to prove partial correctness of the resilient algorithm, i.e, we show if it terminates normally, the computed result is correct. Finally, we establish *total correctness* by showing it must terminate normally.

### 3.1. Execution model

We now describe an execution model for a parallel program using the nonblocking operations described above. A model is comprised of some fixed number of processes, each with its own set of variables and its own *program graph* [Baier and Katoen 2008]. The program graph is a directed graph in which the nodes are *locations* (corresponding to possible values of the program counter for that process) and the edges are *model transitions*. A model transition comprises a *model guard*—a boolean expression that




(a) Initial State

(b) An intermediate state

(c) Result of applying left update to (b)

(d) Result of applying right update to (c)

(e) Result of applying update-middle to (d)

(f) Final state

= data[0] = data[1] = previously computed R = receive S = send

Fig. 2. Illustration of Algorithm 2 with nsteps $= 5$ and nxl $= 7$

determines whether the transition is enabled—and a statement that transforms the state in a single atomic step. We assume the sequential semantics are understood, and focus here on the semantics of the communication operations.

An execution consists of a single sequence of atomic transitions from the different processes comprising the program. The state of such a program must record the state of the nonblocking operations (as these are not atomic). In addition a special *dæmon*

process is introduced to complete those operations; this process may be thought of as representing the runtime system.

*3.1.1. State.* We now make precise the definition of the state of a model.

*Definition* 3.1. A *state* of a model consists of

— the local state of each process, which includes the value of the program counter and the value of each variable
— for each $0 \leq i, j <$ nprocs, integers $\mathsf{sc}_{i,j}$ and $\mathsf{rc}_{i,j}$ which record the numbers of send requests and receive requests, respectively, issued by process $i$ for process $j$
— for each $0 \leq i, j <$ nprocs, a sequence $\mathsf{SQ}_{i,j}$ of *send records* for sends issued from process $i$ to process $j$, and a sequence $\mathsf{RQ}_{i,j}$ of *receive records* for receives issued from process $i$ to process $j$.

*Definition* 3.2. A *send record* $r$ comprises the following information:

— an integer ID number $r.\mathsf{id}$, which together with the source and destination ($i$ and $j$) uniquely identifies the record
— a buffer specification $r.\mathsf{buf}$ (e.g., $\&\mathsf{data}[a][b]$ for integers $a, b$)
— a boolean value $r.\mathsf{comp}$ indicating whether or not the operation is complete
— the message data $r.\mathsf{data}$, which in our case is always a single real number.

A *receive record* is similar, but only comprises the first three fields described above.

*Definition* 3.3. A *communication handle* is a symbol of the form $S(i, j, k)$ or $R(i, j, k)$. The letter $S$ or $R$ indicates whether the operation referenced is a send or receive, $i$ is the PID of the process that issued the operation, $j$ is the PID of the target of that operation (the destination or source, resp.), and $k$ is the ID number of the record for the operation. A special handle value of null indicating "no reference" is also allowed.

In the initial state, all $\mathsf{rc}_{i,j}$ and $\mathsf{sc}_{i,j}$ are $0$ and the sequences $\mathsf{SQ}_{i,j}$ and $\mathsf{RQ}_{i,j}$ are empty.

*3.1.2. Semantics of communication operations.* Execution of a *postSend* in process $i$ with destination $j$ and buffer $b$ creates a new send record $r$ with $r.\mathsf{id} = \mathsf{sc}_{i,j}$, $r.\mathsf{buf} = b$, $r.\mathsf{comp} = \mathit{false}$, and $r.\mathsf{data}$ is undefined. This record is appended to the end of $\mathsf{SQ}_{i,j}$ and $\mathsf{sc}_{i,j}$ is incremented. The value returned by *postSend* is the symbol $S(i, j, r.\mathsf{id})$. Execution of a *postRecv* is similar, adding a record to $\mathsf{RQ}_{i,j}$ and returning a handle of the form $R(i, j, r.\mathsf{id})$.

In addition to the events emanating from process statements, a *dæmon* process executes two additional types of transitions: *send completions* and *receive completions*.

A send completion can occur to any queued send request $r$ for which $r.\mathsf{comp}$ is *false*. This entails setting $r.\mathsf{comp} = \mathit{true}$ and $r.\mathsf{data} = *r.\mathsf{buf}$, i.e., the data is copied from the send buffer to the record in the queue.

A receive completion can occur to any queued receive request $r$ in $\mathsf{RQ}_{i,j}$ for which a *matching* complete send record $r'$ exists in $\mathsf{SQ}_{j,i}$. In our case, *matching* just means the ID numbers agree: $r.\mathsf{id} = r'.\mathsf{id}$. (This simple model suffices because there are no tags or wildcards, so operations can only be matched strictly by the order of issuance.) The completion entails setting $r.\mathsf{comp} = \mathit{true}$ and $*(r.\mathsf{buf}) = r'.\mathsf{data}$, i.e., copying the data from the send record into the receive buffer.

The function *complete* takes a handle. If that handle is non-null, it looks in the specified send or receive queue for the record $r$ with the specified ID, and returns $r.\mathsf{comp}$; $\mathit{complete}(\mathsf{null}) = \mathit{true}$.

*3.1.3. Executions.* The following define precisely the notion of *execution* of a model:

*Definition* 3.4. If $s$ and $s'$ are states, and $t$ is a transition, we write $s \xrightarrow{t} s'$ to mean that $t$ is enabled in $s$ and the state resulting from executing $t$ from $s$ is $s'$.

*Definition* 3.5. An *execution fragment* is a finite or infinite sequence $\xi$ of the form $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots$. We say $\xi$ is *initial* if its first state $s_0$ is an initial state; $\xi$ is *maximal* if it cannot be extended to a longer fragment, either because it is infinite or because no transition is enabled in its last state.

*Definition* 3.6. An *execution* is a maximal initial execution fragment satisfying the *weak fairness* property: a process cannot have a transition enabled forever without that process eventually executing.

The following deal with things that can go wrong in an execution:

*Definition* 3.7. Let $s$ be a state.

(1) We say $s$ is *deadlocked* if at least one process is not terminated in $s$ and there are no enabled transitions in $s$.
(2) We say $s$ is *potentially deadlocked* if at least one process is not terminated in $s$ and the only enabled transitions are send-completions for which no matching receive has been posted.

Clearly, if $s$ is deadlocked then $s$ is potentially deadlock. Avoiding potential deadlock is important because the MPI Standard does not *require* a send operation to complete unless a matching receive is available. Hence a potential deadlock may result in an actual deadlock when a program is executed in some circumstances.

*Definition* 3.8. An execution is *normal* if it is finite and free of potential deadlock. An execution is *abnormal* if it is not normal.

In particular, in a normal execution, every process terminates after a finite number of steps: the execution can't be extended (because it is maximal), and in the final state every process must have terminated (else it would have a deadlock).

### 3.2. Granularity

The concept of *program graph* provides a mathematically precise representation of a program, but there are still many ways in which code such as that of Algorithm 2 can be translated into a program graph. Of particular importance in the notion of *granularity*. For example, the evaluation of the program guard on lines 10–16 of Algorithm 2 could be represented by a single model guard, or it could be decomposed into several atomic steps, between which transitions from other processes may execute. The goal of this section is to show that under reasonable assumptions, these choices cannot make any difference.

*3.2.1. Coarse and fine-grained models.* On one extreme we have the *coarse* model. In this model, the execution of the *choose* statement in Algorithm 2 occurs as a single atomic step. That is, the evaluation of the three program guards, the selection of one which is true and the execution of the corresponding code block occur atomically. (If all three guards are *false*, then no transition is enabled in that process.) Moreover, all of the initialization statements from all processes (lines 6–8) happen as one atomic step, the first transition in any execution. This model is the easiest to reason about, but it may not accurately reflect what happens when the algorithm is implemented in a real programming language and executed on a real machine.

By contrast, in a *fine-grained* model, the evaluation of program guards and selection of one may require multiple atomic steps, as may the execution of the correspond-

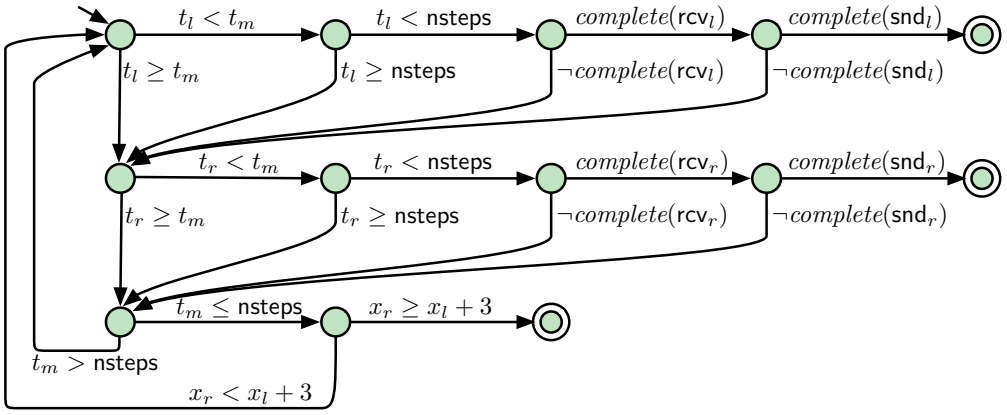



Fig. 3. A fine-grained guard evaluation routine

ing block. In between these atomic steps, other processes may execute. A fine-grained model will generally involve many more states and allow many more executions than the coarse model, and is therefore more difficult to reason about. On the other hand, they can be used to model actual implementations and architectural features with full accuracy.

While the results of this section deal specifically with Algorithm 2, the equivalence of fine-grained and course models holds for very general reasons and applies to a wide class of nonblocking programs. Such programs must be organized as a coarse guarded-transition system and satisfy the following: once a guard in process $p$ becomes *true*, it must remain *true* until a statement in $p$ executes; the *complete* function is used only in guards; and execution of each transition block must be deterministic and terminate in a finite number of steps.

*3.2.2. Guard evaluation and selection.* In a fine-grained model we must make some basic assumptions on the routine that is used to evaluate program guards and select an enabled transition:

(i) if the routine returns a transition $t$ at state $s$ then the program guard of $t$ must hold in $s$;
(ii) for any state $s$ in which control of process $p$ is in the routine and one of the program guards holds, the routine must return an enabled transition belonging to $p$ in a finite number of steps without blocking; and
(iii) the routine may use its own auxiliary variables but these variables are never accessed outside of the guard routine, and the routine may not modify any of the existing variables in the program.

Note that (ii) is a reasonable requirement since if the program guard becomes *true*, it will remain *true*.

An example of a fine-grained guard routine satisfying these properties is shown in Figure 3. This routine "spins" through the various steps required to evaluate the guards and breaks out as soon as it determines one holds.

*3.2.3. Transformation to intermediate model.* Given an execution $E$ in a fine-grained model, we describe how to transform $E$ to an execution $E'$ in an *intermediate model* where each guard evaluation/selection takes place as a single atomic step.

Given $E$, proceed as follows. For each process $p$, consider the projection onto $p$ of $E$. Suppose this sequence is finite and terminates inside the guard routine. It follows that the guard must have been *false* when $p$ entered the routine for the last time; otherwise, our assumptions dictate that the routine would complete in a finite number of steps without blocking, and by weak fairness (Definition 3.6), this would have to occur within $E$. The same holds if the projection is infinite and control stays within the guard routine forever. In either case, we simply remove from $E$ the subsequence of transitions from $p$ beginning with the transition in which $p$ enters the guard routine for the final time. In the resulting execution, $p$ ends in a state where no transitions are enabled in the intermediate model.

Next, for each process $p$, for each successful invocation of the guard selection routine (resulting in a guard evaluating to *true* and a transition selected), simply remove all steps in that routine from the execution and replace the final one with a single transition representing the evaluation of a guard to *true* and the selection of the corresponding transition. Since, by assumption, the guard must have held when the routine exited, the newly introduced single transition will be enabled. Since none of the variables (outside of possible auxiliary ones used exclusively in the guard routine) are modified by the guard routine, the state arrived at must be identical. Hence the result of this transformation is an execution $E'$ in the intermediate model.

We claim that if $E$ is normal then $E'$ is normal and ends in the same state as $E$. Indeed, the length of $E'$ is at most that of $E$, and all states other than the intermediate states in the guard routine calls are unchanged.

We also claim that if $E$ is abnormal then $E'$ is abnormal. To see this, assume $E$ is abnormal. Either (i) there is at least one process $p$ which executes non-guard transitions forever in $E$, or (ii) there is at least one process $p$ which either terminates inside the guard routine or runs forever staying within the guard routine. In case (i), $E'$ is infinite, since the transformation does not remove non-guard transitions. In case (ii), $p$ terminates in $E'$ at a location with at least one outgoing transition but for which no transition is enabled; in particular $p$ has not terminated normally. In either case, $E'$ is abnormal.

*3.2.4. Transformation to coarse model.* Out next goal is to transform an execution in the intermediate model to one in the coarse model. We say an execution in the intermediate model has *coarse form* if it has one of the following forms:

$$\mathit{initialization}*;\ (\mathit{completion}*;\ \mathit{guard};\ \mathit{block})*;\ \mathit{completion}* \tag{4}$$

$$\mathit{initialization}*;\ (\mathit{completion}*;\ \mathit{guard};\ \mathit{block})^{\omega} \tag{5}$$

That is, it consists of a sequence of initializations (lines 6–8 of Algorithm 2) from various processes, followed by some number of completion operations, followed by a guard transition followed immediately by the sequence of transitions from the code block corresponding to that guard; followed by some number of completion operations, followed by another guard transition followed immediately by its code block, and so on. If it is finite it may end with a sequence of completion operations.

Our basic approach follows Lipton's method of "reduction" [Lipton 1975]. The idea is to show that transitions in an execution can be re-ordered in certain ways that preserve properties of interest. To do this we will use the following lemma, which is proved in Appendix A:

LEMMA 3.9. *Suppose* $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2$ *is an execution fragment and one of the following holds:*

(*1*) $t_1$ *and* $t_2$ *are transitions from two different non-dæmon processes, or*
(*2*) $t_1$ *is a completion operation and* $t_2$ *is not a guard transition.*

$$\mathsf{data}[\overline{t(j)-1}][j] = u^{t(j)-1}_{g(\mathsf{pid},j)} \quad \text{if } t(j) \geq 1 \wedge 1 \leq j \leq \mathsf{nxl} \tag{6}$$

$$\mathsf{data}[\overline{t(j)-2}][j] = u^{t(j)-2}_{g(\mathsf{pid},j)} \quad \text{if } t(j) \geq 2 \wedge 1 \leq j \leq \mathsf{nxl} \tag{7}$$

$$\mathsf{data}[\overline{t_l}][0] = u^{t_l}_{g(\mathsf{pid},0)} \quad \text{if } \mathsf{pid} > 0 \wedge \mathit{complete}(\mathsf{rcv}_l) \wedge t_l < \mathsf{nsteps} \tag{8}$$

$$\mathsf{data}[\overline{t_r}][\mathsf{nxl}+1] = u^{t_r}_{g(\mathsf{pid},\mathsf{nxl}+1)} \quad \text{if } \mathsf{pid} < \mathsf{nprocs}-1 \wedge \mathit{complete}(\mathsf{rcv}_r) \wedge t_r < \mathsf{nsteps} \tag{9}$$

$$x_l = t_m - t_l \tag{10}$$

$$x_r = \mathsf{nxl} + 2 + t_r - t_m \tag{11}$$

$$1 \leq t_m \leq \mathsf{nsteps} + 1 \tag{12}$$

$$0 \leq t_l \leq t_m \tag{13}$$

$$0 \leq t_r \leq t_m \tag{14}$$

$$0 \leq x_l < x_r \leq \mathsf{nxl} + 2 \tag{15}$$

If $\mathsf{pid} \neq 0$ then $\mathsf{RQ}_{\mathsf{pid},\mathsf{pid}-1} = \langle r_0, \ldots, r_k \rangle$, where $k = \min\{t_l, \mathsf{nsteps}-1\}$ and $r_j.\mathsf{id} = j \wedge r_j.\mathsf{buf} = \&\mathsf{data}[\overline{j}][0]$ $(0 \leq j \leq k)$. If $\mathsf{pid} \neq 0 \wedge t_l < \mathsf{nsteps}$ then $\mathsf{rcv}_l = R(\mathsf{pid}, \mathsf{pid}-1, t_l)$; otherwise, $\mathsf{rcv}_l = \mathsf{null}$. (16)

If $\mathsf{pid} \neq 0$ then $\mathsf{SQ}_{\mathsf{pid},\mathsf{pid}-1} = \langle r_0, \ldots, r_k \rangle$, where $k = \min\{t_l, \mathsf{nsteps}-1\}$ and $r_j.\mathsf{id} = j \wedge r_j.\mathsf{buf} = \&\mathsf{data}[\overline{j}][1]$ and, if $r_j$ is complete, $r_j.\mathsf{data} = u^j_{g(\mathsf{pid},1)}$ $(0 \leq j \leq k)$. If $\mathsf{pid} \neq 0 \wedge t_l < \mathsf{nsteps}$ then $\mathsf{snd}_l = S(\mathsf{pid}, \mathsf{pid}-1, t_l)$; otherwise, $\mathsf{snd}_l = \mathsf{null}$. (17)

If $\mathsf{pid} \neq \mathsf{nprocs}-1$ then $\mathsf{RQ}_{\mathsf{pid},\mathsf{pid}+1} = \langle r_0, \ldots, r_k \rangle$, where $k = \min\{t_r, \mathsf{nsteps}-1\}$ and $r_j.\mathsf{id} = j \wedge r_j.\mathsf{buf} = \&\mathsf{data}[\overline{j}][\mathsf{nxl}+1]$ $(0 \leq j \leq k)$. If $\mathsf{pid} \neq \mathsf{nprocs}-1 \wedge t_r < \mathsf{nsteps}$ then $\mathsf{rcv}_r = R(\mathsf{pid}, \mathsf{pid}+1, t_r)$; otherwise, $\mathsf{rcv}_r = \mathsf{null}$. (18)

If $\mathsf{pid} \neq \mathsf{nprocs}-1$ then $\mathsf{SQ}_{\mathsf{pid},\mathsf{pid}+1} = \langle r_0, \ldots, r_k \rangle$, where $k = \min\{t_r, \mathsf{nsteps}-1\}$ and $r_j.\mathsf{id} = j \wedge r_j.\mathsf{buf} = \&\mathsf{data}[\overline{j}][\mathsf{nxl}]$ and, if $r_j$ is complete, $r_j.\mathsf{data} = u^j_{g(\mathsf{pid},\mathsf{nxl})}$ $(0 \leq j \leq k)$. If $\mathsf{pid} \neq \mathsf{nprocs}-1 \wedge t_r < \mathsf{nsteps}$ then $\mathsf{snd}_r = S(\mathsf{pid}, \mathsf{pid}+1, t_r)$; otherwise, $\mathsf{snd}_r = \mathsf{null}$. (19)

Fig. 4. Invariants. For each pid ($0 \leq \mathsf{pid} < \mathsf{nprocs}$), the expressions above hold on process pid whenever control is at line 9 of Algorithm 2.

*Then* $s_0 \xrightarrow{t_2} s_1' \xrightarrow{t_1} s_2$ *for some state* $s_1'$. *Furthermore, if* $\{s_0, s_1, s_2\}$ *contains a potentially deadlocked state, so does* $\{s_0, s_1', s_2\}$.

This lemma is used to show the following (also proved in Appendix A):

THEOREM 3.10. *Given any normal fine-grained execution of Algorithm 2, there exists a coarse execution terminating in the same state. If the Algorithm admits an abnormal fine-grained execution, it also admits an abnormal coarse execution.*

### 3.3. Partial Correctness

We next establish partial correctness: if a process terminates, it will satisfy its postcondition (the *Ensures* clause of Algorithm 2). By Theorem 3.10, we may restrict attention to the coarse model of the algorithm. We take the classic approach of showing that a set of formulas all hold in the initial state and are invariant under every (coarse) transition (cf. [Ashcroft 1975]).

The invariants are given in Figure 4. The first four establish the relationship between the values held in the array data and the actual discrete solution $u$ at each step. The next six define $x_l$ and $x_r$ in terms of $t_l$, $t_m$, and $t_r$, and express bounds on these variables. The final four specify the state of the communication data: the left receive and send operations followed by the right receive and send operations.

To show invariance, one must essentially consider each type of transition (the three updates, and the completion operations) and show each preserves each invariant. This is straightforward but tedious, and a sampling of the proof is in Appendix B.

Upon exiting the *while* loop, the loop condition must be *false* and all of the invariants hold. In particular $t_m > \mathsf{nsteps}$, $x_l \leq 1$, and $x_r \geq \mathsf{nxl} + 1$. By (12), $t_m = \mathsf{nsteps} + 1$. According to (3), $t(j) = t_m$ for $1 \leq j \leq \mathsf{nxl}$. Hence (6) becomes $\mathsf{data}[\overline{\mathsf{nsteps}}][j] = u^{\mathsf{nsteps}}_{g(\mathsf{pid},j)}$, which is the desired postcondition.

### 3.4. Total Correctness

In this section we show that every execution of Algorithm 2 is normal (Definition 3.8). The first step is the following (proved in Appendix A):

LEMMA 3.11. *Algorithm 2 is free of potential deadlock (Definition 3.7).*

Now suppose some process does not terminate normally. By Lemma 3.11, this process must execute an infinite number of iterations of the *while* loop. Now, the value $v$ of the expression $t_l + t_r + t_m$ increases by at least one on each iteration of the loop, and therefore $v$ increases without bound. But $t_l, t_r \leq t_m \leq \mathsf{nsteps} + 1$, hence $v \leq 3(\mathsf{nsteps} + 1)$, a contradiction.

## 4. NOISE MODELING

In order to test the resilient algorithm, a noise injection module was developed to simulate nonuniform runtimes across processes.

We borrow from Beckman et al. [2006] and Hoefler et al. [2010] who refer to each individual delay caused by an interruption as a *detour*. Furthermore, they use the term *noise* to refer to the general phenomenon of asynchronous OS level events that interrupt an application's execution. Conceptually, we take this generalization further and use the term *noise* to refer to any detour-causing phenomena which interrupts an application's execution, and which originates outside of the application code.

Previous related studies have focused on two broad classes of noise generation: constant frequency detours [Hoefler et al. 2010; Beckman et al. 2006], and detours which coincide precisely with noise profiles found on existing high performance machines [Hoefler et al. 2010]. In Agarwal et al. [2005], other distributions are introduced as well—specifically, the exponential, Pareto, and Bernoulli distributions. These distributions anticipate, respectively, that detours occur as a Poisson, heavily skewed, or infrequent bursting process. Here we assume as little as possible about noise characteristics. This is due in part to the goal of designing an algorithm that is robust in the presence of a broad range of noise distributions, and also because we still have little detail as a community regarding expected characteristics of exascale machines.

Specifically, noise is parameterized as a vector $\eta$ with components:

$$\eta \equiv [T, \mu, \sigma], \tag{20}$$

using a Gaussian distribution with

— $T \equiv$ detour duration: the (fixed) length of time a process is suspended during a single detour;
— $\mu \equiv$ mean detour period: the average time interval between detours on a process;
— $\sigma \equiv$ detour variation: the standard deviation of the time interval between detours on a process.

This parameterization essentially sets a fuzzy frequency of detours. Again, $T$ is constant, and while the mean, $\mu$, is fixed, for a Gaussian distribution $96\%$ detours occur with a time separation $\mu \pm 2\sigma$. The noise injection API described by Beckman et al. [2006] induces detours by periodically halting the execution of an application in order to force execution of a delay loop [Beckman et al. 2006]. The injection mechanism used in the present analysis essentially uses the same technique. A random number generator is seeded by each process's rank, and a timer is set by choosing the period before the first detour from a uniform distribution. Subsequently, the classic and resilient algorithms proceed. The timer is triggered, and its SIGALRM is processed by a handler function. This handler puts the algorithm to sleep for a time $T$. After waking up, it samples the timer reset period from the gaussian distribution. This process is repeated until the simulation completes after a fixed number of time steps.

## 5. RESULTS

### 5.1. Experimental Methodology

We perform a number of simulations comparing the resilient to the classic algorithm for a variety of choices for $\eta$. The simulations are carried out on the Argonne Leadership Computing Facility's Cetus and Mira machines; an IBM BG/Q with 1600 MHz PowerPC A2 cores, 1 GB RAM per core, 16 cores per node, and a 5D Torus Proprietary Network interconnect. All source code and experimental artifacts are available at http://vsl.cis.udel.edu/downloads/resilient_experiments.tgz.

The algorithms in Section 2 were realized as C programs using MPI. Functions *postSend* and *postRecv* mapped to MPI_ISend and MPI_IRecv, respectively. The guard selection routine in the resilient algorithm uses MPI_Test and MPI_Waitany. It first checks the left and/or right update guards, favoring whichever side has completed fewer receives. In the case of a tie, it favors one or the other according to a bit which flips at each iteration. Only if neither of those transitions is enabled, does it then try the middle-update. If that is also not enabled, it blocks at an MPI_Waitany until one of the four communication requests completes, and tries again. (Note that this routine avoids continuously "spinning," checking guards over and over again, through the use of MPI_Waitany.) The motivation behind the heuristic is to keep $t_l$, $t_r$, and $t_m$ as close as possible. Other heuristics are possible and will be explored in future work.

Using present state-of-the-art machines and synthetic noise to draw general conclusions about stencil computations for an arbitrary supercomputer is not a straightforward task. Specific choices for the values of $\eta$, for example, might constitute a fraction of a time step for a highly complex system of equations and potentially hundreds of time-steps for a much simpler one. To cast our results as generally as possible, then, the experimental parameters, $T$, $\mu$, and $\sigma$, are selected relative to the computational time it takes to complete a single iteration of the classic algorithm for a given problem size (denoted as $C$).

Formally, our test simulations use a dimensionless form of $\eta$, $\eta^*$, defined as:

$$\eta^* \equiv [T^*, \mu^*, \sigma^*]$$

where

$$T^* \equiv \frac{T}{C}$$
$$\mu^* \equiv \frac{\mu}{C}$$
$$\sigma^* \equiv \frac{\sigma}{C}.$$

For all the experiments presented here, a constant problem size of 10,000 points per process core is used, which yields a $C$ of $\approx 0.00207$ seconds on Cetus and Mira. Additionally, each simulation is terminated after the execution of a fixed 10,000 timsteps.

Our experiments measure two quantities—the *resilience speedup* and *resilience efficiency*—for a range of values of $\eta^*$. Specifically, the resilience speedup $S$ is defined as

$$S = \frac{\tau_{classic}}{\tau_{resilient}},$$

the ratio of runtimes for the classic and resilient algorithm for a fixed number of time steps, where the runtime of the classic algorithm is given by $\tau_{classic}$, and the runtime of the resilient algorithm is given by $\tau_{resilient}$. The resilience efficiency $E$ is defined as

$$E = \frac{\tilde{\tau}_{classic}}{\tau_{resilient}},$$

the percent of the runtime of the classic algorithm without simulated noise to the runtime of the resilient algorithm with simulated noise for a fixed number of time steps, where $\tilde{\tau}$ denotes the total runtime in the absence of noise. This quantity provides a measure of the performance of the resilient algorithm in absolute terms. It indicates the efficiency of the resilient algorithm relative to an ideal (i.e. noiseless) machine, and it quantifies the net loss in algorithm performance due to simulated noise.

### 5.2. Experiments

*5.2.1. General observations.* We begin with a few general observations that are discussed in greater detail below. First, we observe that, in the absence of noise, the resilient algorithm is consistently 1-2% slower than the classic algorithm across a wide range of test parameters. This is expected given the extra bookkeeping present in the resilient algorithm. In the presence of noise, however, we observe a broad range of performance regimes. For values of $\eta^* \ll 1$, as we discuss further below, resilience speedup is generally small ($0.9 - 1.5$). Values of of $\eta^* \gg 1$ on the other hand exhibit large resilience speedup in the range of $2 - 30$. The resilience efficiency ranges from $0.2 - 0.8$, and is only affected significantly by variations in $T^*$, $\mu^*$, and the number of processes used. Details supporting these summary observations are discussed below.

*5.2.2. Effects of the Variation of $T^*$ and $\mu^*$.* The first set of experiments varies the non-dimensional mean interrupt period, $\mu^*$, for a number of different values of $T^*$. The resulting resilience speedup and efficiency curves are shown in Figures 5 and 6.

Note first Figure 5. Both the resilience efficiency and resilience speedup are affected by $T^*$ and $\mu^*$ to roughly an equivalent extent. The resilience speedup in this figure additionally shows that the resilient algorithm consistently outperforms the classic algorithm by a small margin. Despite this out-performance, the resilient algorithm remains significantly affected by the noise vectors illustrated in Figure 5. This effect is illustrated by the resilience efficiency which is consistently low (between $0.2$ and $0.5$ in most cases).

Note Figure 6. Observed resilience speedups reach up to 30 for this set of experiments. The runtimes underlying these resilience speedups (not illustrated) indicate that while the classic algorithm is slowed significantly by increases in $T^*$, the resilient algorithm remains largely unperturbed. Indeed, we expect this trend to continue until $T^*$ approaches the local problem size $\mathsf{nxl} = 10,000$. Recall that the resilient algorithm can relax its barriers as long as there is local work remaining on a given process (i.e. $x_r - x_l \geq 3$).

What Figure 6 shares with Figure 5 is the observation that as $\mu^*$ gets larger, the resilience speedup gets smaller, and the resilience efficiency gets larger. This is easily explained by noting that the frequency of detours goes down with larger $\mu^*$ and therefore, both the classic and resilient algorithm are less affected by the otherwise equivalent noise vector. This effect is even more accented in Figure 6 where resilience speedups of different detour sizes go so far as to approach one another as $\mu^*$ increases.

The differences between Figures 5 and 6 illustrate some important mechanisms through which noise affects application performance. Note from Figure 6, that relative to $\mu^*$, $T^*$ has a disproportionate effect on both the resilience speedup in this set of experiments, whereas for the experiments of Figure 5 they had roughly equivalent effects. This could have been predicted, given previous studies on the effects of noise. Petrini et al. [2003] introduced the idea of resonant noise in his analysis of performance bottlenecks. An application was said to enter into *resonance* with computational noise, when the frequency of its barriers, and the granularity of its local work shared similar values with the frequency and duration of its detours. Further, Petrini et al. [2003] concluded, the effects of noise are most detrimental to an application's performance when said noise *resonates* with the application. This idea has proven compelling for many researchers [Tsafrir et al. 2005; Beckman et al. 2006; Hoefler et al. 2009; Hoefler et al. 2010]. However, Beckman et al. [2006] observed that this particular notion of resonance was limited, and updated the idea accordingly. "Fine-grained noise will have little effect on a course-grained application, as it simply will not be able to desynchronize the processes in any significant way... However we see no reason why course-grained noise should not effect a fine-grained application." Perhaps because of this distinction made by Beckman et al. [2006], Hoefler et al. [2010] described the effects of *noise propagation*, as being amongst the most damaging noise events with respect to an application's performance. This term refers to when noise on one processor has the effect of slowing the execution of another processor which depends on it.

Given that $C$ corresponds to the granularity of the classic algorithm, any value of $T^*$ less than a small multiple of $C$ should affect the performance of the classic algorithm minimally. Likewise, any value of $T^*$ which is a large multiple of $C$ should have the effect of propagating to dependent processors. For our model problem, every processor is dependent on each other within a small window of slack.

The resilience efficiencies shown in Figures 6 and 5 are more difficult to understand. Note that the resilience efficiency in Figure 6 is much better than that observed in Figure 5 for smaller values of $T^*$. However, for large enough values of $T^*$, and small enough values of $\mu^*$, the resilience efficiency is equal to that observed in Figure 5. The mechanisms behind these fluctuations in efficiency within very distinct pockets of the domain $\eta^*$ are not entirely clear and require further research.

*5.2.3. Weak Scaling Behavior.* We examine the behavior of resilience speedup and resilience efficiency as process counts are increased while maintaining the same local problem size. A number of researchers have pointed out that noise can have an increasingly negative impact with increasing process counts [Agarwal et al. 2005; Beckman et al. 2006; Hoefler et al. 2010]. To test this hypothesis, we perform a weak scaling

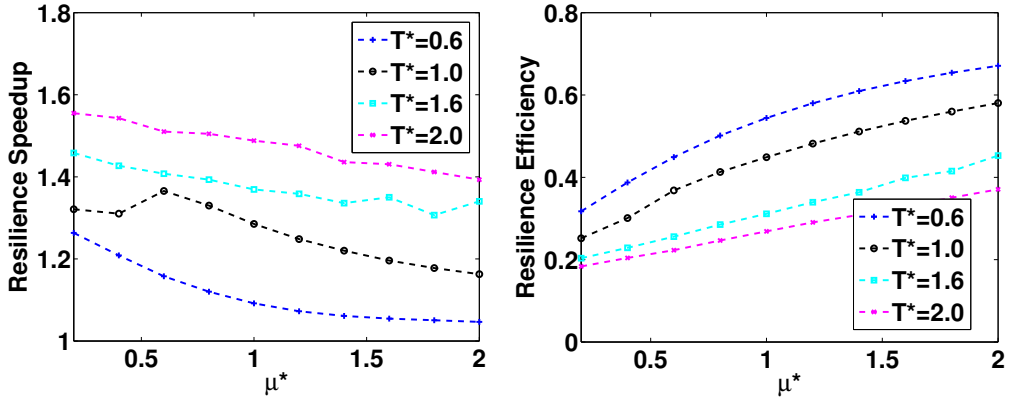



Fig. 5. The resilience speedup and its corresponding resilience efficiency vs. the non-dimensional mean interrupt period, $\mu^* = \frac{\mu}{C}$ for a number of detour durations, $T^* = \frac{T}{C}$. The experiment was performed on a single full node (16 processes), with a problem size of 10,000 points per process; $\sigma^* = T^*$ throughout.

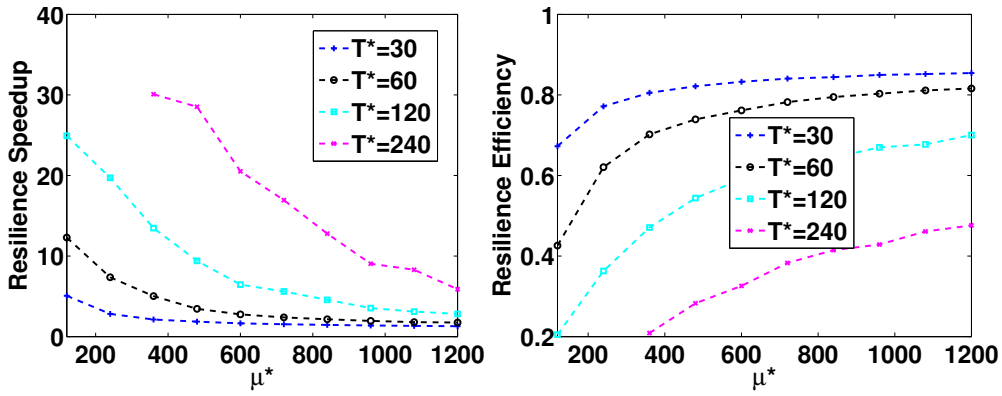



Fig. 6. The resilience speedup and its corresponding resilience efficiency vs the non-dimensional mean interrupt period, $\mu^* = \frac{\mu}{C}$ for a number of detour durations, $T^* = \frac{T}{C}$. The experiment was performed on a single full node (16 processes), with a problem size of 10,000 points per process; $\sigma^* = T^*$ throughout.

experiment for both the classic and resilient algorithms (Figure 7). In the reported experiments, $T^* \gg 1$ and $\mu^* \gg 1$. Experiments with smaller $T^*$ and $\mu^*$ ($\eta^* \ll 1$) were performed as well (not shown). These reveal only that both algorithms perform roughly equivalently under such perturbations, and that there is no significant degradation in the performance of either.

Figure 7 shows that the resilience speedup is significant—a value of approximately 12 for the 32 process case. This value remains roughly constant with increasing process count, peaking at a value of 13 for 1,000 processes. At about 15,000 processes this trend changes, and a significant decline in resilience speedup is exhibited, and continues up to 32,000 processes. At this point, the resilience speedup is 7.5.

This drop in speedup initially indicates that scale degrades the performance of the resilient algorithm faster than that of the classic algorithm. However, this figure ignores the respective increase of the inherent (slight) non-scalability of both approaches due to increased communication costs. Moreover, it masks the fact that the magnitude of increased runtime for the classic algorithm is greater than that for the resilient al-

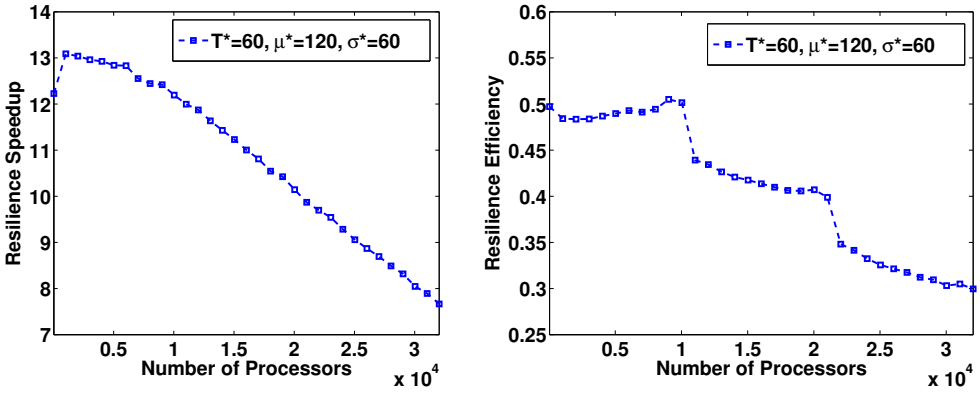



Fig. 7. The resilience speedup and its corresponding resilience efficiency vs. the number of processes. The problem size grew with the number of processes at a rate of 10,000 points per process. The noise vector, $\eta^*$ was held constant.

gorithm, (runtime increases about 100 seconds for the classic algorithm versus about 42 seconds for the resilient algorithm). The drop in speedup occurs because adding 42 seconds roughly doubles the resilient algorithm's runtime, while adding 100 seconds to the classic algorithm increases its runtime by only one sixth. Resilience speedup in Figure 7 therefore drops dramatically, not because the performance of the resilient algorithm degrades more quickly relative to the classic algorithm, but rather because the relative performance of the resilient algorithm degrades more quickly than the relative performance of the classic algorithm.

Note that the resilience efficiency is similar to the 16 process experiments for an equivalent noise vector (shown in Figure 6). Not until approximately 20,000 processes does the resilience efficiency in 7 fall below this value.

*5.2.4. Effects of Varying $\sigma^*$ .* Finally, we examine the behavior of the resilience speedup and resilience efficiency as we vary $\sigma^*$. With the present noise model, $\sigma^*$ essentially controls the non-uniformity of the time spacing of detours on a given process. It should further affect the extent to which detours between adjacent processes are unsynchronized. Beckman et al. [2006] simulate asynchronous noise between processes by injecting constant frequency detours at different start times for each process. Our noise injection approach staggers start times for each process by sampling from a uniform distribution seeded by the rank of each process. A larger standard deviation should decrease the probability of simultaneous detours on neighboring processes. A number of researchers have observed that asynchronous noise is far more detrimental to the performance of bulk synchronous algorithms than is synchronized noise [Petrini et al. 2003; Agarwal et al. 2005; Beckman et al. 2006; Hoefler et al. 2010]. We aim to further probe the effects of asynchronous noise with this set of experiments.

A variation in $\sigma^*$ may have secondary effects of increasing or decreasing the number of detours experienced by a process in the course of a simulation. Because these effects have already been observed in the experiments varying $T^*$ and $\mu^*$, we aim to eliminate them as much as possible in order to isolate the specific effects of $\sigma^*$ on the performance of the classic and resilient algorithms. To achieve this, we limit the number of detours that each process can experience to an arbitrary fixed value, adequate to collect reliable statistics (500 in this case). The results of this experiment can be found in Figure 8.

Figure 8 shows that the resilience speedup is significant (greater than 3), and that it increases with increasing $\sigma^*$ before saturating at a value $S \approx 4.25$. Underlying these

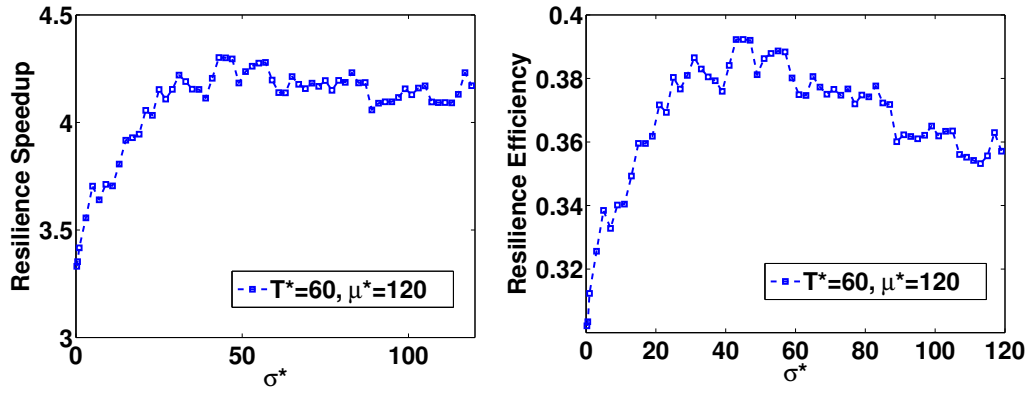



Fig. 8. The resilience speedup and its corresponding resilience efficiency vs. the non-dimensional standard deviation of the interrupt period, $\sigma^* = \frac{\sigma}{C}$. The experiment was performed on a single full node (16 processes), with a problem size of 10,000 points per process. $T^*$ and $\mu^*$ remained at $60$ and $120$ for the course of the experiment. Interruptions where limited to occur 500 times only, so as to isolate the effects of asynchrony introduced by adjustments in $\sigma^*$.

speedups is an increase in the runtime of the classic algorithm combined with little variation in the performance of the resilient algorithm. Understanding the trend and saturation of $S$ with increasing sigma is not straightforward and requires deeper analysis. Given that $S$ saturates when $\sigma^* \approx T^*$ [2], we have some confidence that $\sigma^*$ controls precisely the extent of asynchrony between detours on adjacent processors. Verification of this, and the precise mechanisms by which this control is enforced requires further research.

Other figures, not included in this paper, explore the regime $\eta^* \ll 1$ using a wide range of values of $\sigma^*$. The results reveal that both algorithms perform roughly equivalently, and that there is no significant degradation in the performance of either.

### 5.3. Discussion

For each experiment described above, the dominant factor in the resilience speedup is not a reduction in the runtime of the resilient algorithm, but rather increases in the runtime of the classic algorithm. The general trend of the impact of noise on the classic algorithm can be explained as follows: the longer the detours of each noise event, the greater the negative impact of the noise on the performance of the classic algorithm; the higher the frequency of detours, the greater the negative impact of the noise on the classic algorithm; the more processes used, the greater the impact of the noise on the classic algorithm.

The non-dimensional detour duration, $T^*$, is the dominant parameter in degrading the performance of the classic algorithm. This observation has also been made by Beckman et al. [2006] and Hoefler et al. [2010]. Above a certain threshold of $T^*$, variation in the non-dimensional mean period between detours, $\mu^*$, can also greatly impact to the performance of the classic algorithm. Relative to these two parameters, scaling and the non-dimensional period variation, $\sigma^*$, can have a small yet non-trivial effect on the performance of the classic algorithm. However, observing such effects depends on $\mu^*$ and most importantly on $T^*$ being small and large enough (respectively) to degrade the performance of the classic algorithm in the absence of other effects.

[2] it can be shown that this is a threshold at which $\eta^*$ has maximized the extent to which non-overlapping detours are possible

When detours, $T^*$, are small and have little effect on the classic algorithm's performance for experiments with a small number of processes or a moderate $\sigma^*$ value, the performance is not affected significantly when the number of processes or $\sigma^*$ is increased. Choosing a non-dimensional noise vector, $\eta^*$, from Figures 5 or 6 which does or does not significantly degrade the performance of the classic algorithm determines whether or not increasing $\sigma^*$ or the number of processes exacerbates this degradation. While these general trends are straightforward, the exact mechanisms through which they manifest themselves are not yet clear and require further study.

## 6. CONCLUSIONS

We have introduced a barrier relaxation technique for explicit stencil computations. The motivation for this resilient algorithm is to reduce global synchronization points in the presence of non-uniformities in process execution times anticipated on future supercomputers. We have implemented this technique and demonstrated its application to a simple model problem. Our results show that for a wide range of runtime non-uniformity, resilience speedups range from $0.9 - 1.5$ for high frequency, short detour noise, and $2 - 30$ for lower frequency, longer detour noise. Experimental parameters in which the resilient algorithm outperformed its classic counterpart are those for which previous studies observed major degradation in parallel application performance [Petrini et al. 2003; Agarwal et al. 2005; Beckman et al. 2006; Hoefler et al. 2010]. In brief, longer, less frequent detours negatively impact application performance much more than frequent detours of a shorter duration, asynchronous noise has far greater impact on application performance than synchronous noise, and increasing the number of processes exacerbates the effects of noise. The resilient algorithm provides a significant improvement in performance over the classic algorithm in each of these categories.

The resilient algorithm presented here does not provide an either/or choice with respect to other techniques in fault tolerance or stencil optimization. It is reasonable to think that any of the single process optimizations cited in the introductory section could be coupled to the resilient algorithm, resulting in even greater performance on current state of the art and next generation supercomputers. For example, communication-avoiding algorithms, such as those described in Demmel et al. [2008], could also be combined with the present approach. The method presented here could also be coupled with diskless checkpointing [Plank et al. 1998], dynamic task queueing [Berzins et al. 2010], or to the myriad of techniques for numerical defect correction.

Finally, the results presented here may provide insight to hardware vendors regarding questions faced by application developers in dealing with computational noise. Specifically, how much performance is lost in accounting for runtime non-uniformities between processes? Will we be able to make up for that with the levels of parallelism achievable at exascale? What range of values of $\eta*$ are most easily tolerated algorithmically? The analysis presented here potentially provides insight into these important questions.

## REFERENCES

Saurabh Agarwal, Rahul Garg, and Nisheeth K. Vishnoi. 2005. The impact of noise on the scaling of collectives: a theoretical approach. In *Proceedings of the 12th international conference on High Performance Computing (HiPC'05)*. Springer-Verlag, Berlin, Heidelberg, Article 29, 10 pages. DOI:http://dx.doi.org/10.1007/11602569_31

Gabrielle Allen, Thomas Dramlitsch, Ian Foster, Nicholas T. Karonis, Matei Ripeanu, Edward Seidel, and Brian Toonen. 2001. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (Supercomputing '01)*. ACM, New York, NY, USA, Article 52, 1 pages. DOI:http://dx.doi.org/10.1145/582034.582086

S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K.Hill, J. Hiller, S. Karp, C. Koelbel, D.Koester, P. Kogge, J.Levesque, D. Reed, V. Sarkar, R.Schreiber, M. Richards, A. Scarpelli, J.Shalf, A.Snavely, , and T. Sterling. 2009. *Exascale computing study: Software challenges in achieving exascale systems*. ECSS Report 101909. Georgia Institute of Technology.

Thomas J. Ashby, Pieter Ghysels, Wim Heirman, and Wim Vanroose. 2012. The impact of global communication latency at extreme scales on Krylov methods. In *Proceedings of the 12th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I (ICA3PP'12)*. Springer-Verlag, Berlin, Heidelberg, Article 31, 15 pages. DOI:http://dx.doi.org/10.1007/978-3-642-33078-0_31

E. A. Ashcroft. 1975. Proving assertions about parallel programs. *J. Comput. Syst. Sci.* 10, 1, Article 7 (Feb. 1975), 26 pages. DOI:http://dx.doi.org/10.1016/S0022-0000(75)80018-3

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.* 1, 1, Article 4 (Jan. 2004), 23 pages. DOI:http://dx.doi.org/10.1109/TDSC.2004.2

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press, 55 Hayward Street Cambridge, MA 02142-1315 USA.

Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures (SPAA '12)*. ACM, New York, NY, USA, 193–204. DOI:http://dx.doi.org/10.1145/2312005.2312044

P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. 2006. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In *Cluster Computing, 2006 IEEE International Conference on*. IEEE Computer Society, Washington, DC, USA, 1–12. DOI:http://dx.doi.org/10.1109/CLUSTR.2006.311846

Anna Beletska, Wlodzimierz Bielecki, Albert Cohen, and Marek Palkowski. 2010. Synchronization-Free automatic parallelization: beyond affine iteration-space slicing. In *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing (LCPC'09)*. Springer-Verlag, Berlin, Heidelberg, 233–246. DOI:http://dx.doi.org/10.1007/978-3-642-13374-9_16

Anna Beletska, Wlodzimierz Bielecki, Albert Cohen, Marek Palkowski, and Krzysztof Siedlecki. 2011. Coarse-grained Loop Parallelization: Iteration Space Slicing vs Affine Transformations. *Parallel Comput.* 37, 8 (Aug. 2011), 479–497. DOI:http://dx.doi.org/10.1016/j.parco.2010.12.005

Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. 2008. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. ECSS Report TR-2008-13. The Department of Energy.

Martin Berzins, Todd Harman, Justin Luitjens, Charles A Wight, Qingyu Meng, and Joseph R. Peterson. 2010. *Dynamic Task Scheduling for Scalable Parallel AMR in the Uintah Framework*. SCI Technical Report UUSCI-2010-001. SCI Institute, University of Utah.

Martin Berzins, Qingyu Meng, John Schmidt, and James C. Sutherland. 2012. DAG-Based software frameworks for PDEs. In *Proceedings of the 2011 international conference on Parallel Processing (Euro-Par'11)*. Springer-Verlag, Berlin, Heidelberg, 324–333. DOI:http://dx.doi.org/10.1007/978-3-642-29737-3_37

George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. 2009. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.* 69, 4 (April 2009), 410–416. DOI:http://dx.doi.org/10.1016/j.jpdc.2008.12.002

David L. Brown, Paul Messina, Pete Beckman, David Keyes, Jeffery Vetter, Mihai Anitescu, John Bell, Ronald Brightwell, Brad Chamberlain, Donald Estep, Al Geist, Bruce Hendrickson, Michael Heroux, Rusty Lusk, John Morrison, Ali Pinar, John Shalf, and Mark Shephard. 2010. *Cross Cutting Technologies For Computing At the Exascale*. Technical Report. U.S. Department of Energy (DOE) Office of Advanced Scientific Computing Research and the National Nuclear Security Administration.

D. Chazan and W. Miranker. 1969. Chaotic relaxation. *Linear Algebra Appl.* 2, 2 (1969), 199 – 222. DOI:http://dx.doi.org/10.1016/0024-3795(69)90028-7

Zizhong Chen and Jack Dongarra. 2006. Algorithm-based Checkpoint-free Fault Tolerance for Parallel Matrix Computations on Volatile Resources. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, Washington, DC, USA, 97–97. http://dl.acm.org/citation.cfm?id=1898953.1899028

A. T. Chronopoulos and C. W. Gear. 1989. s-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.* 25, 2 (Feb. 1989), 153–168. DOI:http://dx.doi.org/10.1016/0377-0427(89)90045-9

Kaushik Datta. 2009. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-177.html

J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. 2008. Avoiding communication in sparse matrix computations. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE Computer Society, Washington, DC, USA, 1–12. DOI:http://dx.doi.org/10.1109/IPDPS.2008.4536305

Chris Ding and Yun He. 2001. A ghost cell expansion method for reducing communications in solving PDE problems. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM) (Supercomputing '01)*. ACM, New York, NY, USA, 50–50. DOI:http://dx.doi.org/10.1145/582034.582084

Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rde, and Christian Weiss. 2000. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis* 10 (2000), 21–40.

E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408. DOI:http://dx.doi.org/10.1145/568522.568525

John Daly et. al. 2012. Inter-Agency Workshop on HPC Resilience at Extreme Scale. (February 2012).

Amr Fahmy and Abdelsalam Heddaya. 1996. *Management of Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk*. Technical Report. Boston University, Boston, MA, USA.

Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 361–366. DOI:http://dx.doi.org/10.1145/1088149.1088197

B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. 2000. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal Supplement Series* 131, 1 (2000), 273. http://stacks.iop.org/0067-0049/131/i=1/a=273

Hajime Fujita, Robert Schreiber, and Andrew A. Chien. 2013. It's Time for New Programming Models for Unreliable Hardware. (18 March 2013). https://engineering.purdue.edu/~milind/waci/fujita.pdf Conference Talk Given At: ASPLOS 2013 Provocative Ideas session.

Al Geist and Christian Engelmann. 2002. Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors. (2002). Submitted 2002.

Evangelos Georganas, Jorge González-Domínguez, Edgar Solomonik, Yili Zheng, Juan Touriño, and Katherine Yelick. 2012. Communication avoiding and overlapping for numerical linear algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 100, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389132

P. Ghysels, T. Ashby, K. Meerbergen, and W. Vanroose. 2013. Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines. *SIAM Journal on Scientific Computing* 35, 1 (2013), C48–C71. DOI:http://dx.doi.org/10.1137/12086563X

William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (2nd ed.). MIT Press, Cambridge, MA, USA.

J.A. Gunnels, D.S. Katz, E.S. Quintana-Orti, and R.A. Van de Gejin. 2001. Fault-tolerant high-performance matrix multiplication: theory and practice. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE Computer Society, Washington, DC, USA, 47–56. DOI:http://dx.doi.org/10.1109/DSN.2001.941390

Rinku Gupta, Pete Beckman, Byung-Hoon Park, Ewing Lusk, Paul Hargrove, Al Geist, Dhabaleswar Panda, Andrew Lumsdaine, and Jack Dongarra. 2009. CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP '09)*. IEEE Computer Society, Washington, DC, USA, 237–245. DOI:http://dx.doi.org/10.1109/ICPP.2009.20

Michael A. Heroux and Al Geist. 2013. Resilience: Past, Present and Future. (2013). Sandia National Laboratory.

Torsten Hoefler, Timo Scheider, and Andrew Lumsdaine. 2009. THE EFFECT OF NETWORK NOISE ON LARGE-SCALE COLLECTIVE COMMUNICATIONS. *Parallel Processing Letters* 19, 04 (2009), 573–593. DOI:http://dx.doi.org/10.1142/S0129626409000420

Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International*

*Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. DOI:http://dx.doi.org/10.1109/SC.2010.12

Kuang-Hua Huang and J.A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *Computers, IEEE Transactions on* C-33, 6 (1984), 518–528. DOI:http://dx.doi.org/10.1109/TC.1984.1676475

Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2006. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness (MSPC '06)*. ACM, New York, NY, USA, 51–60. DOI:http://dx.doi.org/10.1145/1178597.1178605

Jin-Soo Kim, Soonhoi Ha, and Chu Shik Jhon. 1998. Relaxed Barrier Synchronization for the BSP Model of Computation on Message-passing Architectures. *Inf. Process. Lett.* 66, 5 (June 1998), 247–253. DOI:http://dx.doi.org/10.1016/S0020-0190(98)00061-1

Fredrik Berg Kjolstad and Marc Snir. 2010. Ghost Cell Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP '10)*. ACM, New York, NY, USA, Article 4, 9 pages. DOI:http://dx.doi.org/10.1145/1953611.1953615

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721. DOI:http://dx.doi.org/10.1145/361227.361234

Yudan Liu, R. Nassar, C.B. Leangsuksun, N. Naksinehaboon, M. P?un, and S.L. Scott. 2008. An optimal checkpoint/restart model for a large scale high performance computing system. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE Computer Society, Washington, DC, USA, 1–9. DOI:http://dx.doi.org/10.1109/IPDPS.2008.4536279

Guoming Lu, Ziming Zheng, and Andrew A. Chien. 2013. When is multi-version checkpointing needed?. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale (FTXS '13)*. ACM, New York, NY, USA, 49–56. DOI:http://dx.doi.org/10.1145/2465813.2465821

John Mccalpin and David Wonnacott. 1999. *Time Skewing: A Value-Based Approach to Optimizing for Memory Locality*. Technical Report. In http://www.haverford.edu/cmsc/davew/cache-opt/cache-opt.html.

Jiayuan Meng and Kevin Skadron. 2009. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 256–265. DOI:http://dx.doi.org/10.1145/1542275.1542313

Fabian Oboril, Mehdi B. Tahoori, Vincent Heuveline, Dimitar Lukarski, and Jan-Philipp Weiss. 2011. Numerical Defect Correction As an Algorithm-Based Fault Tolerance Technique for Iterative Solvers. In *Proceedings of the 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing (PRDC '11)*. IEEE Computer Society, Washington, DC, USA, 144–153. DOI:http://dx.doi.org/10.1109/PRDC.2011.26

Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. 2003. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC '03)*. ACM, New York, NY, USA, 55–. DOI:http://dx.doi.org/10.1145/1048935.1050204

J.S. Plank, K. Li, and M.A. Puening. 1998. Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on* 9, 10 (1998), 972–986. DOI:http://dx.doi.org/10.1109/71.730527

Gabriel Rivera and Chau-Wen Tseng. 2000. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (Supercomputing '00)*. IEEE Computer Society, Washington, DC, USA, 32. http://dl.acm.org/citation.cfm?id=370049.370403

Even J. Rosser. 1998. *Fine-Grained Analysis of Array Computations*. Ph.D. Dissertation. Dept. of Computer Science, The University of Maryland, Maryland.

A. Roy-Chowdhury, N. Bellas, and P. Banerjee. 1996. Algorithm-based error-detection schemes for iterative solution of partial differential equations. *Computers, IEEE Transactions on* 45, 4 (1996), 394–407. DOI:http://dx.doi.org/10.1109/12.494098

L. Ridgway Scott, Terry Clark, and Babak Bagheri. 2005. *Scientific Parallel Computing*. Princeton University Press, Princeton, NJ, USA.

Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 2001. Rescheduling for Locality in Sparse Matrix Computations. In *Proceedings of the International Conference on Computational Sciences-Part I (ICCS '01)*. Springer-Verlag, London, UK, UK, 137–148. http://dl.acm.org/citation.cfm?id=645455.653905

Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. 2005. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 303–312. DOI:http://dx.doi.org/10.1145/1088149.1088190

Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. DOI:http://dx.doi.org/10.1145/79173.79181

D. Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE Computer Society, Washington, DC, USA, 171–180. DOI:http://dx.doi.org/10.1109/IPDPS.2000.845979

## A. PROOFS

PROOF OF LEMMA 3.9. (1) If $t_1$ and $t_2$ modify only local variables it is clear they commute. They also commute if either or both are posts of sends or receives, since they modify two distinct queues. A guard transition can only depend on local variables and the result of the *complete* function, and only a (dæmon) completion operation can affect the result of that function.

(2) If $t_2$ belongs to an ordinary process, it cannot involve the *complete* funtion, and hence can be moved before any completion operation. If $t_2$ is a completion operation it commutes with any other completion operation since they must operate on distinct records.

If $s_1$ is potentially deadlocked, then so is $s_2$: by definition, $t_2$ must be a send completion, so executing $t_2$ only reduces the set of enabled transitions. □

PROOF OF THEOREM 3.10. First transform to an execution with atomic guards in the intermediate model. Find the first initialization transition in this execution, and move it all the way to the left using repeated applications of Lemma 3.9. This is justified since an initialization is neither a guard nor a dæmon transition. Find the next initialization and move it left until it is just to the right of the first one. Repeat this process until all initializations occur at the beginning of the execution. Let $t_0$ be the last initialization transition.

Next, find the first non-dæmon transition $t$ after $t_0$; $t$ must be a guard transition. Say $t$ belongs to process $p$. Find the first transition $t'$ after $t$ belonging to $p$. Then $t'$ is not a guard; all of the transitions between $t$ and $t'$ belong to other processes or are dæmon transitions. By Lemma 3.9, $t'$ can be moved to the left to the point just after $t$. Repeat this for each transition belonging to $p$ until the last transition in the block is reached.

Now let $t_0$ be the last transition in the block just completed and repeat the paragraph above. Continue in this way. If the execution is finite, this will yield an execution in the form of (4) and ending in the same state as the original execution. If infinite, it defines a sequence of finite sequences, each a prefix of the next, and hence defines an infinite sequence of the form (5). Finally, if the original execution passes through a potenetially deadlocked state, so does the final one, since it was obtained by repeated applications of Lemma 3.9. □

PROOF OF LEMMA 3.11. Assume a reachable state $s$ is potentially deadlocked. In $s$, at least one process has not terminated, and every process that has not terminated is at line 9 with its local state satisfying all of the following:

$$t_m \leq \mathsf{nsteps} \vee x_l > 1 \vee x_r < \mathsf{nxl} + 1 \tag{21}$$

$$t_l \geq t_m \vee t_l \geq \mathsf{nsteps} \vee \neg \mathit{complete}(\mathsf{rcv}_l) \vee \neg \mathit{complete}(\mathsf{snd}_l) \tag{22}$$

$$t_r \geq t_m \vee t_r \geq \mathsf{nsteps} \vee \neg \mathit{complete}(\mathsf{snd}_r) \vee \neg \mathit{complete}(\mathsf{rcv}_r) \tag{23}$$

$$t_m > \mathsf{nsteps} \vee x_r < x_l + 3. \tag{24}$$

It can be shown that (21)–(24), together with the invariants (10)–(15), imply

$$(t_r < t_m \wedge t_r < \mathsf{nsteps}) \vee (t_l < t_m \wedge t_l < \mathsf{nsteps}). \tag{25}$$

Furthermore, the only enabled transitions can be send completions for which no matching receive request has been posted.

Let $i$ be the PID of a non-terminated process in $s$ with minimal $t_m$. We will assume $t_r < t_m \wedge t_r < \mathsf{nsteps}$ and obtain a contradiction. A dual argument (omitted) shows that

assuming $t_l < t_m \wedge t_l <$ nsteps leads to a contradiction. It will therefore follow from (25) that no potentially deadlocked state $s$ can be reached in any execution.

So assume $t_r < t_m \wedge t_r <$ nsteps. We must have $i <$ nprocs $- 1$, since if $i =$ nprocs $- 1$, $\mathsf{snd}_r$ and $\mathsf{rcv}_r$ are always complete, contradicting (23). So consider process $i+1$, writing $x'$ for the value of any variable $x$ on process $i + 1$. Either (i) $t_r \leq t'_l$ or (ii) $t_r > t'_l$.

In case (i), it follows from (16) and (17) for process $i + 1$ that there are records in $\mathsf{SQ}_{i+1,i}$ and in $\mathsf{RQ}_{i+1,i}$ with ids $t_r$. By (18) and (19) for process $i$, those records match $\mathsf{rcv}_r$ and $\mathsf{snd}_r$, respectively. Therefore $\mathit{complete}(\mathsf{rcv}_r) \wedge \mathit{complete}(\mathsf{snd}_r)$ must hold on process $i$, contradicting (23).

In case (ii), there are records in $\mathsf{SQ}_{i,i+1}$ and in $\mathsf{RQ}_{i,i+1}$ with ids $t'_l$. Now process $i+1$ is not terminated in $s$ (as $t'_l < t_r <$ nsteps). Furthermore, $\mathit{complete}(\mathsf{rcv}'_l) \wedge \mathit{complete}(\mathsf{snd}'_l)$ must hold, else the dæmon process would be able to complete the requests. By (22) on process $i + 1$, $t'_l \geq t'_m$, and therefore $t'_m \leq t'_l < t_r \leq t_m$, contradicting the assumption that process $i$ has minimal $t_m$. □

### B. INVARIANT PRESERVATION

We consider here two kinds of transitions. The other cases are similar.

*Left update.* Assume the guard holds (line 10) and the invariants hold (figure 4). We wish to show that the invariants hold after executing the update of the left region.

Since $\mathsf{rcv}_l$ is complete, the value of $\mathsf{data}[\overline{t_l}][0]$ is given by (8). This is not affected by the posting of the receive at line 12, which is to $\mathsf{data}[1 - \overline{t_l}][0] = \mathsf{data}[\overline{t_l + 1}][0]$. There are no other accesses to $\mathsf{data}[\overline{t_l}][0]$ in the transition, and at the end $t_l$ is incremented. Since (16) held upon entering, the id of the new record posted must be $t_l+1$. This shows that (16) continues to hold after the transition is executed.

We now describe a loop invariant for the loop which begins after the assignment $j \leftarrow 1$. The loop invariant is the conjunction of two formulas. The first formula, the frame condition, is invariant for the obvious reason that none of the state components referenced in the formula are modified in the loop body. This part of the loop invariant is the conjunction of (7)–(17). The second part is the conjunction of the following:

$$1 \leq j \leq x_l \tag{26}$$

$$p = \overline{t(j)} \tag{27}$$

$$\forall k(0 \leq k < j)\ .\ \mathsf{data}[\overline{t(k)}][k] = u^{t(k)}_{g(\mathsf{pid},k)} \tag{28}$$

$$\forall k(j \leq k \leq \mathsf{nxl})\ .\ \mathsf{data}[\overline{t(k)}][k] = u^{t(k)-2}_{g(\mathsf{pid},k)} \tag{29}$$

Let's see that these hold upon reaching the loop (when $j = 1$). Formula (26) follows from the assumption $x_l > 1$ in the guard. Formula (27) follows from the assignment to $p$, equation (3) and the fact that

$$\overline{t(1)} = \overline{t_l + 1} = 1 - \overline{t_l}\,.$$

To prove (28) we just consider the case $k = 0$, where we have

$$\mathsf{data}[\overline{t(0)}][0] = \mathsf{data}[\overline{t_l}][0] = u^{t_l}_{g(\mathsf{pid},0)} = u^{t(0)}_{g(\mathsf{pid},0)}$$

by (8). Formula (29) follows from (7).

Assume now that loop invariant holds and $j \leq x_l - 1$. We must show the loop invariant holds after executing the loop body. The call to *update* performs an assignment (ignoring for now the additional code for handling special cases and posting a send).

The expression on the right hand side of this assignment satisfy the following:

$$\mathsf{data}[1-p][j-1] = \mathsf{data}[\overline{t(j-1)}\,][j-1] = u^{t(j-1)}_{g(\mathsf{pid},j-1)} = u^{t(j)-1}_{g(\mathsf{pid},j)-1} \tag{30}$$

$$\mathsf{data}[1-p][j] = \mathsf{data}[\overline{t(j)-1}\,][j] = u^{t(j)-1}_{g(\mathsf{pid},j)} \tag{31}$$

$$\mathsf{data}[1-p][j+1] = \mathsf{data}[\overline{t(j+1)-2}\,][j+1] = u^{t(j+1)-2}_{g(\mathsf{pid},j+1)} = u^{t(j)-1}_{g(\mathsf{pid},j)+1}. \tag{32}$$

Indeed, (30) follows from (28) for the case $k = j-1$. Equation (31) follows from (7). Equation (32) follows from (29) for $k = j+1$. It follows that the right-hand side of the assignment evaluates to

$$f(u^{t(j)-1}_{g(\mathsf{pid},j)-1}, u^{t(j)-1}_{g(\mathsf{pid},j)}, u^{t(j)-1}_{g(\mathsf{pid},j)+1}) = u^{t(j)}_{g(\mathsf{pid},j)}$$

and that this is assigned to

$$\mathsf{data}[p][j] = \mathsf{data}[\overline{t(j)}\,][j]$$

Hence, after $j$ is incremented at the end of the loop body, (28) will continue to hold. In the special case where *update* is called and $g(\mathsf{pid}, j) \in \{0, nx-1\}$ the function returns without doing anything and trivially maintains loop invariants.

Finally, we'll show that (17) holds after the transition has executed. Note that a send is only posted once (when $j = 1$), and a record, $r$ will be defined and added to the set of records, $\{r\}$ as $r.\mathsf{id} = t_l$, $r.\mathsf{comp} = \mathsf{false}$, and $r.\mathsf{buf} = \&\mathsf{data}[p][1] = \&\mathsf{data}[1-\overline{t_l}\,][1] = \&\mathsf{data}[\overline{t_l}\,][1]$. This satisfies (17). The other terms in the loop invariant follow for trivial reasons.

*A receive completion.* Assume the invariants hold and receive record $r$ in $\mathsf{RQ}_{\mathsf{pid},\mathsf{pid}-1}$ completes. The only invariant possibly affected is (8), because of its use of $\mathit{complete}(\mathsf{rcv}_l)$. If the execution of the completion affects this formula, it must change the value of $\mathit{complete}(\mathsf{rcv}_l)$ from $\mathsf{false}$ to $\mathsf{true}$. In this case we must have $\mathsf{pid} \neq 0$, so by (16), $\mathsf{rcv}_l = R(\mathsf{pid}, \mathsf{pid}-1, t_l)$ and hence the ID of the matching send is $t_l$. By (19) on process $\mathsf{pid}-1$, the data in that message is $u^{t_l}_{g(\mathsf{pid}-1,\mathsf{nxl})} = u^{t_l}_{g(\mathsf{pid},0)}$. By the semantics of receive-completion, this data will be stored in the receive buffer, which by (16) is $\mathsf{data}[\overline{t_l}\,][0]$, establishing (8) in the new state.

Received November 2013; revised XXX; accepted XXX