Asynchronous Two-Level Checkpointing Scheme for Large-Scale Adjoints in the Spectral-Element Solver Nek5000

Michel Schanen, Oana Marin, Hong Zhang, Mihai Anitescu
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, USA
$\{$*mschanen,oanam,hongzhang,anitescu*$\}$@*anl.gov*

***Abstract*—Adjoints are an important computational tool for large-scale sensitivity evaluation, uncertainty quantification, and derivative-based optimization. An essential component of their performance is the storage/recomputation balance in which efficient checkpointing methods play a key role. We introduce a novel asynchronous two-level adjoint checkpointing scheme for multistep numerical time discretizations targeted at large-scale numerical simulations. The checkpointing scheme combines bandwidth-limited disk checkpointing and binomial memory checkpointing. Based on assumptions about the target petascale systems, which we later demonstrate to be realistic on the IBM Blue Gene/Q system Mira, we create a model of the expected performance of our checkpointing approach and validate it using the highly scalable Navier-Stokes spectral-element solver Nek5000 on small to moderate subsystems of the Mira supercomputer. In turn, this allows us to predict optimal algorithmic choices when using all of Mira. We also demonstrate that two-level checkpointing is significantly superior to single-level checkpointing when adjoining a large number of time integration steps. To our knowledge, this is the first time two-level checkpointing had been designed, implemented, tuned, and demonstrated on fluid dynamics codes at large scale of 50k+ cores.**

***Keywords*-Two-Level Checkpointing; Adjoints; Gradient; Large Scale; Nek5000 ; CFD; PETSc**

I. Introduction

The computation of adjoints of time-dependent nonlinear systems of equations puts a huge strain on memory requirements. The goal of this paper is to enable adjoint-based optimization on partial differential equations (PDEs) written generically as

$$\mathbf{u}_t = \mathcal{L}[\mathbf{u}] \text{ in } \Omega \ , \mathbf{u}|_{\partial\Omega} = \mathcal{B}[\mathbf{u}] \ , \mathbf{u}|_0 = \mathbf{u}_0,$$

where the subscript t stands for time differentiation, the operator $\mathcal{L}$ accumulates spatial operators such as gradient and Laplacian, and $\mathcal{B}$ denotes the boundary conditions. The adjoint problem is derived by using perturbation theory to be a PDE given by

$$-\bar{\mathbf{u}}_t = \mathcal{L}^*[\mathbf{u}, \bar{\mathbf{u}}] \text{ in } \Omega \ , \bar{\mathbf{u}}|_{\partial\Omega} = \mathcal{B}^*[\bar{\mathbf{u}}] \ \ \bar{\mathbf{u}}|_T = f(\mathbf{u}_T),$$

where $\bar{\mathbf{u}}$ denotes the adjoint variable of $\mathbf{u}$, $\mathcal{L}^*$ is the adjoint operator, and $\mathcal{B}^*$ is the boundary conditions of the adjoint problem. Note that the adjoint problem depends on the solution of the forward one and is subject to time reversal from T to 0.

We assume that the time integration is performed by using a multistep scheme of order k and ignoring whether the scheme is implicit or explicit since this aspect bears no importance for our purposes. If we consider our scheme given by, say, a function F, we then have for the forward problem an update of the type

$$\mathbf{u}_{n+1} = \mathbf{u}_n + F(h, \mathcal{L}[\mathbf{u}_{n+1-i}]_{i \le k}),$$

where h is the time-step length. Note that here we intentionally omitted whether i starts from 0 or 1, which marks the difference between an implicit or explicit scheme, and also the exact formula for F, which is merely a linear combination of the given arguments and can be found in the literature for either Adams-type methods or backward differentiation.

Similarly for the reverse we have

$$\bar{\mathbf{u}}_{n+1} = \bar{\mathbf{u}}_n + F(h, \mathcal{L}^*[\mathbf{u}_{n+1+i}, \bar{\mathbf{u}}_{n+1-i}]_{i \le k}). \qquad (1)$$

The *adjoint run* will access $\mathbf{u}_{n+1+i}, i \le k$ available from the forward run. Notice that at the peak memory requirement, just after the forward run, all *checkpoints* $\mathbf{u}_{n+1+i}$ are stored in memory and $\bar{\mathbf{u}}_{n+1}$ is computed according to (1). This data-flow reversal problem of $\mathbf{u}_{n+1-i}$ is known as adjoint checkpointing in the field of algorithmic differentiation (AD) [1] or discrete adjoints. Ideally we want to checkpoint (store) all the states $\mathbf{u}$ in memory and reuse them whenever needed. In practice, however, because of the limit of memory size, it is feasible to checkpoint only selective states and recompute the missing states. Griewank and Walther [2] developed an offline algorithm, named `revolve`, to generate a checkpointing schedule that has been proven to minimize the number of recomputation time steps, given the total number of time steps and the number of allowed checkpoints in memory. To address the problems that an a priori number of time steps is not known (e.g., in the case of adaptive time stepping), researchers have proposed several online checkpointing algorithms [3], [4], [5]. Walther's algorithms [3] have been proven to be optimal when the number of time steps is no more than $\binom{s+2}{s}$, where s is the maximum number of checkpoints, and near-optimal after that bound and up to $\binom{s+3}{s}$ time steps. Wang's algorithm [5], complementary to Walther's, deals with an even larger number of time steps. Although it does not guarantee optimality in terms of recomputations, it is proven to be optimal in terms of repetition number, defined as the maximum number of times needed to compute a specific time step during the adjoint computation. The common feature for all these offline and online algorithms is that only RAM is considered to be the storage medium and the cost of storing and restoring checkpoints is negligible.

For limited RAM capacity and a large number of time steps, `revolve` may require a tremendous number of recomputation time steps which would hamper the performance. Using extra storage devices such as external disks for checkpointing may reduce the overall computational time despite the I/O cost, which may be considerable even on massively parallel systems with many nodes. To this end, Stumm and Walther proposed a multistage offline algorithm in [6] to minimize the overall access cost to checkpoints, which could be stored either in memory or on disk, with the choice depending on the frequency of the read-write checkpoint operations. This algorithm also uses the binomial approach and can be considered an extension of the `revolve` algorithm taking into account cheap and expensive checkpoints. The access cost of one disk checkpoint is assumed to be less than the cost for one time step, which is a strong restriction.

More important, all these algorithms apply only one binomial strategy in the adjoint computation and hence can be classified as single-level schemes. Furthermore, none of them makes good use of extra storage devices other than RAM, leaving adjoint checkpointing still challenging for limited RAM capacity and a large number of time steps, a not-uncommon situation in large-scale scientific computations. Although having proven the optimality bounds for these algorithms, optimality proofs for two-level checkpointing are still limited. In [7], Aupy et al. prove an optimal scheme for synchronous as opposed to asynchronous two-level checkpointing with unlimited size and non-negligible latency for the disk.

In this paper, we propose a new two-level scheme that exploits the usage of disk to increase allowed checkpoints and at the same time takes advantage of the optimal results from `revolve` at one of the levels. Our scheme can handle both online and offline checkpointing for large scale simulations that run on petascale systems such as the IBM Blue Gene/Q system Mira, while allowing for multiple binomial strategies, each handling a subsequence of time steps. In our two-level checkpointing scheme we distinguish between storing a checkpoint to memory or to disk under the following assumptions:

A1 The total number of time steps is a priori unknown.

A2 Disk I/O is limited only by bandwidth and latency (not by size).

A3 There may be infinite disk checkpoints, writing to archive tapes is considered as disk.

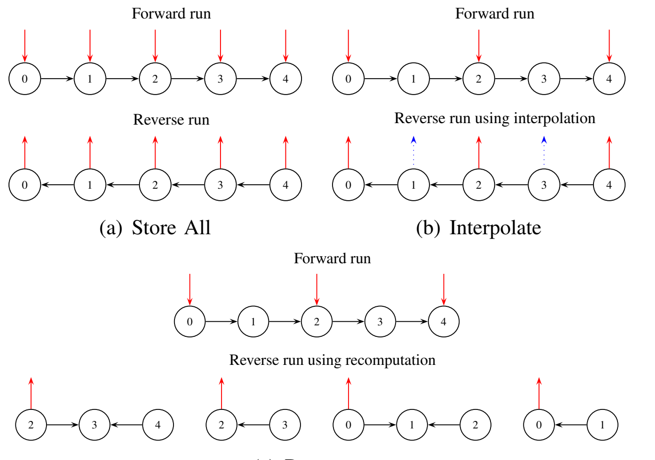
A4 Memory is bound by size.

A5 Memory bandwidth is infinite (writing/reading is 0 cost compared to the rest of the application).

Assumption A1 allows for a time stepper where the total number of time steps is unknown. In fluid dynamics, the physics targeted by the Nek5000 code, researchers commonly seek statistical convergence of important flow quantities (such as windowed energy spectra). The statistical convergence time is not known a priori. Assumptions A2 and A3 posit that the amount of data that can be stored is limited only by the bandwidth and thus by time. An application that is able to exploit the full I/O bandwidth is able to write during the entire execution time without being limited by storage size. Assumption A4 describes the limited RAM per core that is limited by the ever-increasing clock rate, also known as the *memory wall*. The evolution of floating-point operations per second and memory per core is increasingly diverging. Assumption A5 considers the writing and reading of checkpoints into RAM to be instantaneous. Obviously, in practice RAM has non-negligible latency. However, extensive measurements have shown that in comparison with the execution time of one time step, RAM can be neglected for the Nek5000 code.

The setup of our work differs from preceding approaches in significant ways. Single-level checkpointing schemes can accommodate a stringent Assumption A4 only by significantly increasing the number of recomputed steps for very long time horizons [2], [5]. The currently proposed multi-level checkpointing schemes [6], [7] cannot satisfy A1.

We aim to accommodate all five assumptions. In particular, our contributions are the following: (1) a novel asynchronous two-level adjoint checkpointing algorithm, (2) a performance model of this algorithm under the parametric limitations of Assumptions A2 and A4, (3) validation of this performance model on a large subsystem of the Mira supercomputer, and (4) prediction of the performance for running the largest possible adjoint computation instance for Nek5000 on Mira. In the numerical experiments section we will demonstrate that in our target regime of a large number of time steps, our two-level approach outperforms single-level checkpointing. To our knowledge, this is the first case of a two-level checkpointing scheme demonstrated for fluid dynamics adjoint computations on the scales described here. Moreover, an interesting feature of our solution is that it makes intensive use of all assets of the supercomputers—CPU, memory, disk, and archive—to improve overall time to solution.


Forward run
Reverse run
Forward run
Reverse run using interpolation
Forward run
Reverse run using recomputation

(a) Store All (b) Interpolate (c) Recompute

Figure 1. Three options for recovering the required states in the reverse run. The first (a) stores all states and restores all of them from memory. The second (b) interpolates the missing states (blue dotted up-arrow). This leads to approximated values for the adjoints due to interpolation errors. The third option (c) recomputes missing states from stored states. This is the method used in this paper. It has no approximation errors and is only a trade-off between recomputations and memory usage.

II. Adjoint Checkpointing

Without loss of generality we assume a time-stepping-based algorithm where checkpoints are stored or restored only between two time steps. Adjoint checkpointing is closely related to restart checkpoints in the domain of resilience. Resilience checkpoints also save the entire state of a program in the computational run [8]. Determining the state of a program is the main task when implementing restart capabilities. Despite these similarities, adjoint and restart checkpointing differ in the way the stored checkpoints are being accessed. Resilience checkpointing usually requires only the last checkpoint to be stored while previous ones are discarded according to a given resilience strategy. Adjoint computations, however, require access to all previously computed states, although not at the same time. Figure 1(a) shows a naive implementation of a *store all* adjoint checkpointing scheme. In the naive case all the checkpoints are stored in the forward run during the computation of the primal values, while the checkpoints are restored in the reverse run during the computation of the adjoints. All checkpoints are equally needed during the computational flow reversal. This request pattern never changes, is static, and is known a priori. The disadvantage of the naive implementation is that at nontrivial checkpoint sizes and numbers of time steps, the store to memory or to disk becomes infeasible. RAM does not provide enough space, and the disks do not provide enough bandwidth. The only solution is to skip the storing of a checkpoint, which is equivalent to a failure in terms of resilience. Two ways exist for recovering from this loss of information. One can interpolate (see Fig. 1(b)) using the previous and next checkpoint, or one can use *recomputations* (see Fig. 1(c)) to recompute the state of the lost checkpoint. Interpolation implies an approximation error that is not the subject of this paper. Recomputing a state introduces a higher runtime while decreasing the memory and/or bandwidth requirement. An adjoint checkpointing scheme tries to find a potentially optimal balance of recomputation cost and the amount of memory or bandwidth; this philosophy is the focus of this work.

A. *Binomial checkpointing using revolve*

Under the assumptions presented in Sect. I, the memory checkpointing fulfills the optimality conditions of `revolve` [2]. Measurements will show that our revolve-based memory checkpointing will not have exactly zero cost for restoring and storing checkpoints from RAM, thus making it possibly suboptimal although still efficient. The role of `revolve` in our multi-level checkpointing scheme is reduced to a `revolve` interface that is called by our scheme introduced in the next section. It implements the function `revolve(` $state, q, adjstate, snaps$`)`, where $state$ is the forward state, $adjstate$ the adjoint state, q the stride size, and $snaps$ the number of available checkpoints. The function `whatodo` is implemented as described in [2].

III. Asynchronous Two-Level Checkpointing Algorithm

This section introduces a novel two-level checkpointing scheme that, as opposed to `revolve`, satisfies Assumption A1, in addition to A2–A5 in Sect. I. That is, it is an *online* algorithm: it works correctly and efficiently even without a priori knowledge of the number of time steps [5].

Previous two-level approaches consider disk usage only insofar as the checkpoints that are used the least should be dumped on disk; they do not envision disk as part of an optimal strategy, as we do here. In [6], for example, the authors assume that the number of disk checkpoints is limited in contrast, the algorithm presented here makes a stronger requirement A3: that the disk space is unlimited. On the other hand, as our scheme does not regard the disk as the end storage but only as a caching mechanism for the practically unlimited tape archive, A3 is not limiting. The only constraint is A2: that the bandwidth, given by the network where the data is being transfered down the memory hierarchy, is limited by a measurable and known value. This assumption will be verified by performance tests in Sect. V, Figures 8(a) and 8(b).

We present in Fig. 2 the hardware setup for our target system Mira. There are two storage devices for large files. One are the disks of Mira, connected to the compute nodes at a relatively high bandwidth. Storage size is still limited, however. Hence the archive, called the high-performance storage system (HPSS), is made available for storing large amounts of data. There is no official limit on its size, its

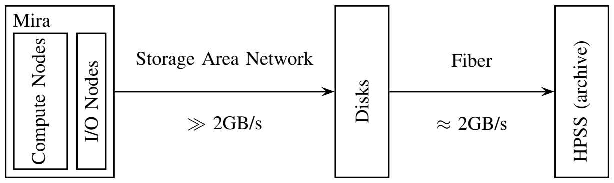


Figure 2. Mira storage hierarchy. The archive is referred to as *disk* in our algorithm. The disks on Mira are transparent and serve only as a buffer between archive and Mira.

main limiting factor being bandwidth of around 2 GB/s. We assume that the write bandwidth to Mira disk is negligible compared with that of the transfer to archive, which takes up a considerable amount of time and can be done asynchronously (see Sect. V for timings). Our top-level checkpointing layer uses disks only as caching mechanisms for transfer to archive, and not as resident storage, thus making the archive bandwidth transfer the practical limiting factor.

Our two-level checkpointing algorithm proceeds as follows. On the top level we carry out a bandwidth-limited checkpointing to disk/archive where no new checkpoint is generated until the current one has finished writing to disk/archive. As a result, the current checkpoint stride, q, is a function of the available bandwidth. On the lower level we use `revolve`, with checkpointing in memory, which is known to be optimal during computational flow reversal once q is prescribed. Notice that this is not a three-level checkpointing algorithm: the disks are used only as a buffer between the archive and the memory.

First, we introduce the first level with checkpoints to disk in Sect. III-A, followed by the memory checkpoints in Sect. III-B, which are applied in the reverse run of the disk checkpoints.

A. Checkpointing to disk (archive)

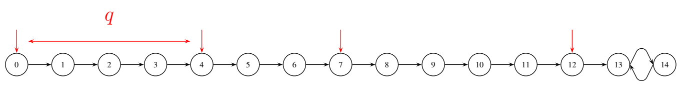


Figure 3. Forward run with a store (down-arrow) of disk checkpoints at every q-th time step. q changes at every stride and is dependent on the bandwidth to archive.

The first level checkpointing stores a checkpoint at every qth time step (see Fig. 3). This process can be asynchronous, and it is assumed so for the rest of this work. The larger the *stride* q, the more recomputations that will be required by `revolve` in the reverse run (see Sect. III-B). The reason is that the `revolve` memory checkpointing is restricted by a limited number of checkpoints c while covering this larger stride q The lower bound of q is given only by the bandwidth (network) and the checkpoint size. The checkpoint size itself is defined by the problem size and the wall clock time of one time step (a property of Nek5000 on Mira, which will be profiled in Sect. V).

Algorithm 1 Bandwidth-limited disk checkpointing: forward run and cached asynchronous push to disk

```
Require: Inital conditions: $state_0$
  $done \leftarrow false$
  apush($state_0$)
  $c \leftarrow 0$, $q \leftarrow 0$, $t \leftarrow 0$
  while $!last$ do
    $state_t, last \leftarrow$ forwardStep($state_{t-1}$)
    if $transfer\_done$ then
      apush($q$)
      apush($state_t$)
      $c \leftarrow c+1$, $q \leftarrow 0$
    end if
    $q \leftarrow q+1$
    $t \leftarrow t+1$
  end while
  apush($q$)
  apush($c$)
  return $state_t$
```

Algorithm 1 presents the algorithm for storing checkpoints. The checkpoints are stored using a stack interface by calling `apush`($state_t$), where $state_t$ is the state at time step t. `apush` is considered to be asynchronous: the push is initalizied and the function returns while the data is being copied. As a consequence of A2 and A3, we assume that `apush` never fails because of disk space limitations.

First the initial $state_0$ are stored via `apush`($state_0$), where c is the counter of all the stored disk checkpoints and q is the distance in time steps to the last checkpoint. The algorithm loops over all the time steps $t = 1$ to $t = n$, where the forward solver in `forwardStep()` computes state $state_t$ based on $state_{t-1}$. The application that implements `forwardStep()` has to return via the Boolean $last$ whether this was the last time step or not. Bandwidth is considered available if there is no checkpoint currently transferred via `apush` to the archive. This availability is represented by the boolean $transfer\_done$, being $true$ if `apush` is done with the transfer and $false$ otherwise. If the transfer is done, a new checkpoint is stored using an asynchronous `apush`($state_t$). This operation is preceded by storing the stride size q that corresponds to the just finished transfer. The q information is important for the offline revolve scheme that is being applied in the adjoint run (see Sect. III-B). The checkpoint counter c is then increased by one, and the stride size q is reset to 0. Finally, independently of whether a checkpoint is saved or not, the stride size q is increased by one. After the last time step $t = n$ has been computed, `forwardStep()` returns the $last$ set to $false$, and the while loop breaks. The last stride size q is being pushed together with the final number of checkpoints c.

If the bandwidth goes to 0, $transfer\_done$ is never set to $true$. Thus the entire computation consists only of one stride. Notice that in the case of multistep time steppers, where $state_t$ is dependent on $state_{t-i}$, $0 < i \leq k$, Algorithm 1 is still valid, as all the additionally required states are saved on disk; the checkpoint size grows by a factor of k. Once the forward run is completed, the computational flow reversal starts using the restore of a disk checkpoint and the application of `revolve` as a second-level checkpointing in order to compute the adjoints.

B. Restoring from disk and applying `revolve`

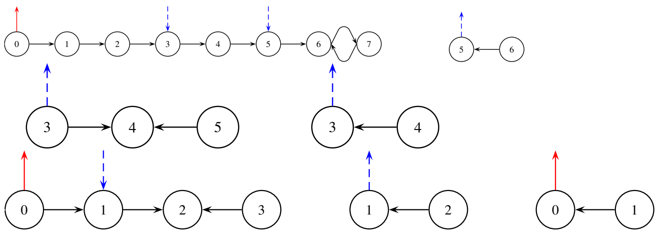


Figure 4. Single reverse run of one stride with $q = 7$ where one disk checkpoint is restored (up-arrow) and memory checkpoints are stored (dashed down-arrow) and restored (dashed up-arrow) according to revolve. The disk checkpoint at $t = 0$ is cached in memory after first restore.

This section describes the stride-by-stride restoring of the disk checkpoints followed by the application of `revolve` and thus the actual computation of the adjoints. For each stride with variable size q in Fig. 3, `revolve` is run according to Fig. 4. The disk checkpoint is restored (up-arrow), and the forward run begins where new memory checkpoints of state $state_i$ (dashed down-arrows) are placed according to the revolve scheme ($i = 3$ and $i = 5$). At the end of the stride (here $i = 7$), the adjoints are initialized by the initial condition of the adjoint state $adjstate_0$ or taken over by the adjoint state $adjstate$ of the last adjoined stride. After the adjoint of $i = 5$ is computed, $i = 3$ is restored. Then state $i = 4$ is recomputed and the adjoint of $t = 4$ is evaluated. Again, state $i = 3$ is restored and the adjoint of $i = 3$ computed. Now, the disk checkpoint (cached in memory) of $i = 0$ is restored, and state $i = 1$ is recomputed and stored. This step is followed by state $i = 2$ being recomputed and the adjoint of $i = 2$ being computed. State $i = 1$ is restored, followed by the adjoint evaluation of $i = 1$. The adjoint of $i = 0$ then is computed by reusing again the checkpoint of state $i = 0$. The additional number of recomputations compared with a store-all strategy is 3. The total memory checkpoints used is 2, whereas the store-all strategy would have used 5.

The interface for `revolve` used in Algorithm 2 is presented as a black box through the interface function `revolve(`$state, q, adjstate, snaps$`)`, where $state$ is the forward state and $adjstate$ is the $adjstate$ used to adjoin a stride of length q under the constraint of using at most $snaps$ number of checkpoints.

Algorithm 2 Bandwidth-limited disk checkpointing: reverse run and pop asynchronous ahead from disk

```
Require: Initial adjoints: $adjstate_0$
  $c \leftarrow$ pop()
  $q \leftarrow$ pop()
  $state \leftarrow$ pop()
  $q_c \leftarrow$ apop()
  $state_c \leftarrow$ apop()
  $adjstate \leftarrow$ revolve($state, q, adjstate_0$)
  for $c - 2$ to 0 do
      $q \leftarrow q_c,\ state \leftarrow state_c$
      $q_c \leftarrow$ apop()
      $state_c \leftarrow$ apop()
      $adjstate \leftarrow$ revolve($state, q, adjstate$)
  end for
  return $adjstate$
```

The revolve interface is called by Algorithm 2, while the disk checkpoints are restored. The checkpoints are restored with one prefetch buffer. The only requirement is the initial adjoint state $adjstate_0$. The checkpoint counter is popped by a call to a synchronous `pop`. Then the first stride size is popped, followed by the state $state$. All this has to be done synchronously because c, q, and $state$ have to be fully loaded before `revolve` is called for the first time. Now an asynchronous pop `apop()` is called to prefetch the next stride size q_c and state $state_c$. As opposed to `pop()`, the function `apop()` directly returns while doing the transfer in the background, leading to the first call to `revolve` that computes the updated adjoint state $adjstate$ of the first stride of size q based on the restored state $state$ and the initial adjoints saved in $adjstate_0$. During this computation `apop` continues to restore the second state in $state_c$. When the first call to `revolve` is done, we enter the checkpoint loop which will be repeated $c - 1$ times. The prefetched stride size q_c and state $state_c$ will be copied to q and $state$, respectively, followed by the next prefetch of the next stride in q_c and $state_c$ through a call to `apop`. Then, `revolve` is called on the current stride. After $c - 1$ steps, the final result is saved in $adjstate$. Again, notice that in the case of multistep time steppers the checkpoints are considerably larger.

IV. Algorithmic Analysis

Here we analyze the performance of our two-level checkpointing algorithm. As a performance metric, we use the number of time-step recomputations in the adjoint, lower being better. For brevity we consider the fixed-stride case, which can provide considerable insight into the overall performance. Our analytical tool is encapsulated in the following result.

Proposition 1. *Given C allowed checkpoints in RAM, the number of extra forward steps (recomputations) needed by the two-level checkpointing Algorithm 2 with a fixed stride size Q for the adjoint computation of M time steps (M is not known before the end of the computation is flagged) is*

$$N_r = M + n_s\, p(Q, C) + p\,(n_l, C) \tag{2}$$

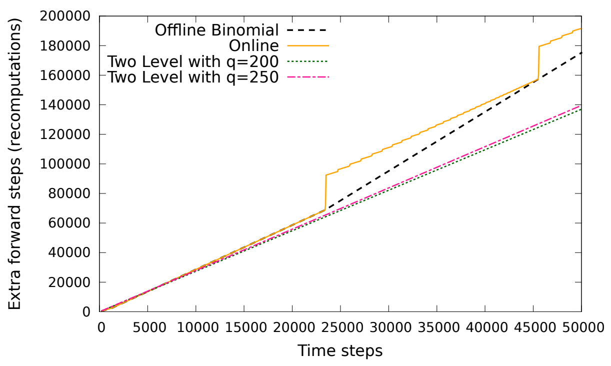
where $n_s = \lfloor M/Q \rfloor$ is the number of Q-sized strides and $n_l = M \bmod Q$ is the size of the last stride. Here $p(a, b) = t\, a - \binom{b+t}{t-1}$, where t is the unique integer (also known as repetition number [2]) satisfying $\binom{b+t-1}{t-1} < a \leq \binom{b+t}{t}$.

Proof: According to Proposition 1 in [2], $p(a, b)$ gives the number of extra forward steps using `revolve` to adjoin a sequence of a time steps storing up to b checkpoints. Thus, the overall number of forward steps needed for a time steps is $a + p(a, b)$. The first-level checkpointing of the two-level scheme storing data to disk requires integrating M steps. All the following forward steps should be considered as "extra" steps. Summing up the overall number of forward steps in each stride and substituting $M = n_s\, Q + n_l$ leads to (2). $\blacksquare$

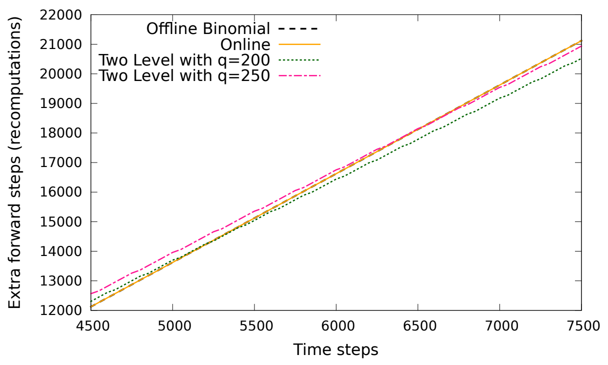
With this proposition, we can predict the actual number of recomputations of our online checkpointing algorithm for different M, C, and Q. Figure 5 plots, as a function of the total number of time steps M, the number of recomputations for the offline `revolve` algorithm [2]; the online algorithm combing Walther's algorithm [3] and Wang's algorithm [5]; and our two-level algorithm with stride sizes $Q = 200$ and $Q = 250$. The number of maximum checkpoints in RAM is set to $C = 50$. The combined algorithm in (2) using only checkpoints in RAM has been proven to be optimal when the number of time integration steps is less than $\binom{50+2}{2} = 1326$, and near-optimal until the number of time steps reaches $\binom{50+3}{3} = 23426$. After that point, it switches to Wang's algorithm [5], which has a proven minimal repetition number. The offline `revolve` algorithm of case (1) is included for comparison even if it is not applicable since M is not known a priori. Its performance is computed as if M were known a priori.

From Fig. 5(a) we can see that our algorithm needs fewer recomputations than does the combined online algorithm if M is sufficiently large (larger than approximately $5,300$ for $Q = 200$ and $6,550$ for $Q = 250$). For lower values of M, our algorithm may need more recomputations, but only slightly; see Fig. 5(b). Nevertheless the benefits in recomputations by using our algorithm significantly increase as the number M of integration time steps becomes large. For example, at time step $50,000$, the gap in the number of recomputations between our algorithm and the combined online algorithm is already over $50,000$. That is, *our two-level algorithm requires less than 75% of the number of recomputations of the best single-level online scheme.* We note that these number ranges are common for complex fluid dynamics simulations such as those carried out by Nek5000 [9].

An interesting observation deriving from Proposition 1 and also seen in Fig. 5(b) is that a smaller stride size Q results in an even small M, where our two-level checkpointing scheme presents a definite advantage. This suggests the benefit of using as small a stride Q as feasible. In the next section we will validate this observation and demonstrate the other benefits of our algorithm with profiling data from actual Nek5000 runs on Mira.



(a) 250-50000 time steps



(b) 4500-7500 time steps

Figure 5. Comparison with online and offline revolve algorithms in terms of time-step recomputations. There are up to 50 checkpoints available in RAM. For better visibility, (b) is extracted from a portion of (a).

V. Performance Analysis Based on Mira and Nek5000

In this section we carry out profiling experiments and analyze how our predicted performance results, which use Assumptions A1–A5, compared with actual performance as measured on Nek5000 runs on the Mira supercomputer. Furthermore, once we validate our performance model based on Proposition 1, we present a prediction framework that allows a user to extrapolate the runtime behavior from a small-sized test case to a full-scale simulation run.

A. Adjoint Nek5000 Calculations Validation

Nek5000 is a spectral-element thermo-hydraulics solver, here however we focus only on the flow part given by the incompressible Navier-Stokes equations. Neglecting the force terms and presuming the flow to be driven by initial/boundary conditions we have the adjoint Navier-Stokes equations given in (3).

$$\frac{\partial \bar{\mathbf{u}}}{\partial t} + (\nabla \mathbf{u})^\intercal \bar{\mathbf{u}} - \mathbf{u} \cdot \nabla \bar{\mathbf{u}} + \frac{1}{Re} \nabla^2 \bar{\mathbf{u}} + \nabla \bar{p} = 0, \ \nabla \cdot \bar{\mathbf{u}} = 0 \quad (3)$$

Our test case is based on the 2d lid driven cavity (see Fig. 6(a)) which consists of a box where the boundary conditions are set to 0, except for the velocity u in x direction of the top lid set to 1. Adjoint computations need a flow metric for computing the adjoint sensitivity. In our numerical experiments, we compute the sensitivity of the final kinetic energy $E_k = \frac{1}{2} < \mathbf{u}_T, \mathbf{u}_T > / < \mathbf{u}_0, \mathbf{u}_0 >$ normalized by the initial condition $\mathbf{u}_0$, T being the end time. E_k is here used as an objective function, and it is an important metric for flow sensitivity [10]. One adjoint computation gives us the gradient $\frac{dE}{d\mathbf{u}_0}$ by setting the initial condition of the adjoint (at final time, due to the computational flow reversal) to $\bar{\mathbf{u}}_T = \mathbf{u}_T$. Currently, Nek5000 implements the solvers of both the primal (forward run) and dual (adjoint run) equations. Adjoint implementations are complex and thus error prone. The adjoint of the fully nonlinear Navier-Stokes equations is a new topic which was not thoroughly addressed in the field of fluid dynamics, due to heavy computational expense. To validate the implementation the gradient of the objective function for one time step is compared using finite differences with the one accumulated by using an adjoint run (see Fig. 6) of the 2d lid driven cavity. This comparison requires a number of computations proportional to the number of degrees of freedom, whereas the adjoint is a constant of 2.2 slower than a forward step. This makes Figure 6(b) with only 1,296 degrees of freedom already roughly 600 times more expensive than Figure 6(a) (see below for a more elaborate estimate of the adjoint runtime). Some artifacts persist at element boundaries, most likely due to a mismatch between the low order finite-difference calculation and the way continuity is enforced across boundaries in the spectral element setup, however we have agreement qualitatively as well as in velocity magnitude

B. Nek5000 Performance Profile

Nek5000 has been shown to scale up to 250,000 cores. For a more detailed description of Nek5000, please refer to the user manual [11].

Two parameters quantify the problem size: the element number m and the polynomial order p. In three dimensions, the problem size n is $m \cdot p^3$, which is proportional to the size of a checkpoint. For double precision and a three-dimensional case it is equal to $n \cdot 8\ [bytes] \cdot 3$. The physical quantity that is being checkpointed is the velocity field $\mathbf{u}_i$ at a particular time step i. Nek5000 uses a multistep time integration scheme, i.e. time step $\mathbf{u}_i$ depends on k previous time steps $\mathbf{u}_{i-1}, \mathbf{u}_{i-2}, \ldots \mathbf{u}_{i-k}$, where k gives the accuracy of the method. In our test case k is always set to 1.

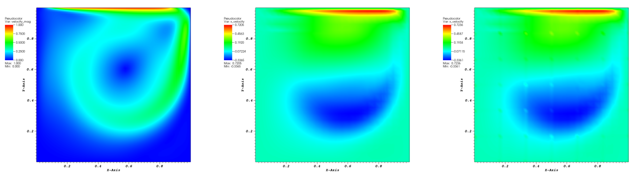

(a) Velocity $\mathbf{u}$ (b) Adjoint (c) Finite difference

Figure 6. Validation of the gradient in x direction computed of the energy E_k at time step t with respect to velocity $\mathbf{u}$ at time step $t-1$ via adjoint and finite difference in the 2d lid driven cavity.

A realistic saturation of Mira in terms of both memory and floating-point operations is crucial in order to obtain results that are indicative of the Nek5000 performance in a production run. We rely on empirical data of past experiments. The important parameter for the scaling of Nek5000 is n_P, the numbers of degrees of freedom per processor, roughly equal to n/P, where n is the problem size defined above and P is the number of processors on which the problem is run. In our experiments, the polynomial order was set to an average number of 12. In this regime, Nek5000 shows good scaling when run with two processes per core provided that it uses an n_P of at least $2,000$ to $4,000$ degrees of freedom per process, where communication overhead becomes significant. The smallest choice in this range for n_P gives the fastest solution. However, this may not be the most cost-efficient solution in terms of energy consumption or core hours.

Given that each node has 16 cores, we run with 32 processes on one node. One node has 16 GB of RAM, so we end up with 0.5 GB of maximum RAM per process. This gives us a maximum limit on n_P, the degrees of freedoms per process. To have a representative test case, we assumed that 0.25 GB of RAM per process should be used, leading to $50,000$ degrees of freedom per process. To be precise, the RAM usage is expected to be equal to $50,000 \cdot 8\ [bytes] \cdot 450 \approx 200MB$, which we validated in measurements throughout our tests. The factor 450 is a Nek5000-specific factor, defining the workspace required to carry out various internal operations, mapping from complex geometries to reference elements etc. On top of this additional space is needed for the checkpoints, as will be presented shortly. As a test case, we switched the same physical case from two to three dimensions, thus having a computationally realistic use case that allows us to scale the problem size to the number of computational nodes. That is, we can keep the degrees of freedom per node constant independent of the number of nodes used.

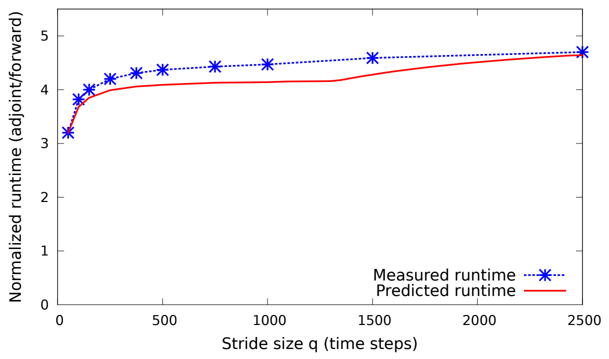


Figure 7. Normalized runtime consisting of the adjoint runtime divided by the forward runtime. The predicted ratio assumes that storing and restoring checkpoints has no cost and one adjoint and primal solve for one time step have the same runtime. The seeming discontinuity in the predicted time marks the point at $\binom{s+2}{s} = \binom{52}{50} = 1300$ mentioned in Sect. I. This point marks maximum reusage of the checkpoints and thus also maximum memory access, hinting at the largest difference at this point between measured and predicted curve in life experience. For `revolve` 50 checkpoints were used on 8,192 cores (512 nodes) at a problem size of $100,000$ degrees of freedom per core. The disk I/O bandwidth during the forward run determines the stride of time steps that revolve will be applied to. The predicted runtime was analytically derived and serves as a lower bound. Revolve neglects memory access time.

C. Memory Checkpoints Using Revolve

For the memory checkpoints we set a maximum number of 50. This translates into a reasonable memory consumption of $50 \cdot 3\ [dimension] \cdot 8\ [bytes] \cdot 50000\ [points\ per\ process] \approx 60MB$ per process. Thus Nek5000 requires a total of 260 MB of memory per node. Notice that with 5 step time stepper the RAM consumption increases to $200 + 5 * 60 = 500MB$. The already high number of 50 checkpoints is from a user's perspective fixed in Nek5000. Based on Proposition 1, increasing it further is not reasonable because the runtime gains are limited; decreasing them does not free any reasonable amount of memory. For a different architecture this choice may need to be revisited.

The first benchmarks were done on 512 nodes, amounting to $8,192$ cores and thus $16,384$ processes. Besides the number of checkpoints, `revolve` requires the total number of time steps, in our case the stride size. Based on the analysis in [2], we can produce the predicted additional effort of recomputation revolve needs after feeding in the information we have measured in our runs that the dual solve takes 2.2 times as long as the primal solve. The predicted performance result is displayed in Fig. 7 as normalized runtime, that is, the ratio of adjoint computation to forward computation.

In Fig. 7 we see also that the normalized runtime is already relatively flat after 500 time steps. Much of the runtime improvement can be done only if the strides get below 500 time steps. However, having a normalized runtime below 5 at a stride size of 1,000 time steps is still considered acceptable. Given the curve flatness, we decided that the reasonable range of strides should be no larger than 2,500, and we then ran numerical experiments on Mira with the stride in the range of 50–2,500. The results are displayed in Fig. 7. We see that the agreement between prediction and measurement is excellent; they are off by 8.5% in the worst case. This points out that the assumptions of `revolve`, also used in this work, and representing a subset of the A1–A5 assumptions are indeed reasonable to produce this performance model of our second-level checkpointing layer. The next section investigates larger settings, the disk/archive performance, and the impact on the first-level checkpointing layer of our approach.

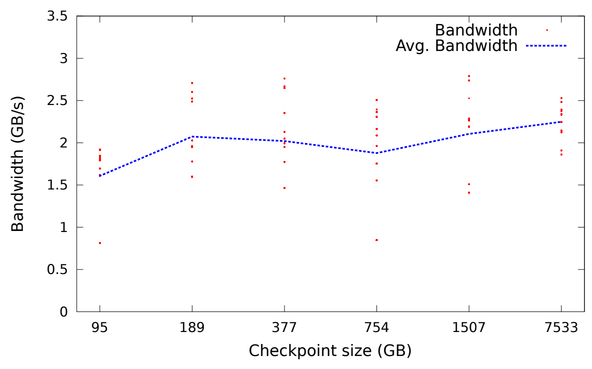
D. Disk Checkpoints Based on Revolve Performance and Bandwidth

We are now in a position to state our overall performance model. The final assumption, which is a design objective of Nek5000 and also extensively observed empirically for this code, is that the compute time of a forward time step can be assumed constant if the number of degrees of freedom per process, n_P, is fixed to some quantity τ_s which in this model can be measured with a small experiment. Since this is an indication of the scalability of the checkpointed code and not related directly to checkpointing, we keep it separate from A1–A5. Then the size $S(n)$ of a checkpoint can be computed as we did in Sect. V-C. The disk/archive bandwidth $B(S)$ can be estimated by running read/write experiments on Mira with files of the target checkpoint size. With this measured data, the expected stride size of our first-layer checkpointing scheme becomes $q = S(n)/(B(S(n))\tau_s)$. Subsequently, the performance of the reverse computation, a normalized adjoint/forward time-step calculation, can be computed from the `performance` model as we did in Sect. V-C and displayed in Fig. 7. This model now can produce an estimate of the usage of all Mira resources at any run parameter size.

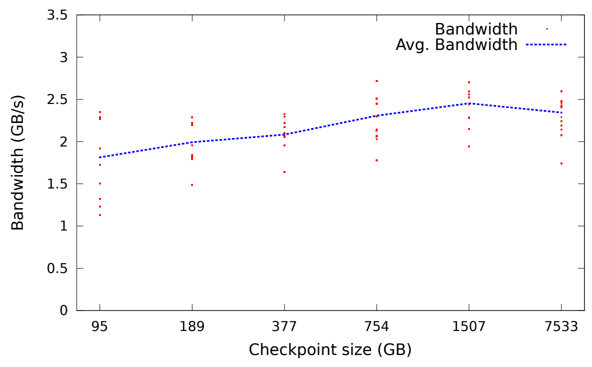
We present the results of this approach in Table I. We carry out Mira experiments to compute the bandwidth dependence on checkpoint size, $B(S)$. Then, using τ_s measured from previous runs, we compute the expected stride size q and the corresponding revolve adjoint/forward performance, which we display in Table I.

Concerning the accuracy of this estimate, an issue that may come into play is the latency of the archive. The archive has itself a level of disk storage where the data is eventually recorded on tapes according to an unknown caching strategy. However, it has been empirically shown that a storage to tape occurs only after about a week of storage in the archive. A primal run of Nek5000 usually does not run over a week. Nonetheless, if one chooses to do so, the algorithm may have to be adapted to a higher prefetch window, since the latency of the tapes is considerably higher.

Figure 8(a) and Fig. 8(b) give an overview of the variations in bandwidth to the archive while storing and restoring



(a) Store bandwidth



(b) Restore bandwidth

Figure 8. HPSS bandwidth with 320 measurements at 10 time intervals during 2 days for each checkpoint size.

data (see Table I for checkpoint sizes). The results are based on 320 transfers store and restore for each number of nodes at 10 time intervals of around 1-2 hours spread over 3 days. Only at very The adjoint run where the checkpoints are restored takes 2 to 5 times longer than the forward run where the checkpoints are stored (see Fig. 7). Only in very rare occasions the, if for example the adjoint run takes twice as long, the store was very fast (2.5 GB/s) and the restore very slow (1GB/s), it might lead to a problem. At worst, the asynchronous restore is not finished after a stride has been adjoined. However, this has not been observed in practice.

Next we validate our performance model and our contention that our two-level checkpointing scheme scales weakly for a given fixed stride size. For this experiment, the stride size was chosen to be fixed at $q = 250$. According to Table I this allows us to increase the number of nodes and thus the problem size up to 8,192 nodes, after which the stride size would have to be increased further. The primal and dual computation weakly scale equally well, with the predicted ratio of the normalized runtime being at a constant 4.2, deducible from Fig. 7. We see in Fig. 9 that the measured normalized runtime is within 2% of the predicted one for the entire range of 512–4,096 nodes. This both supports our contention that we have a valid performance

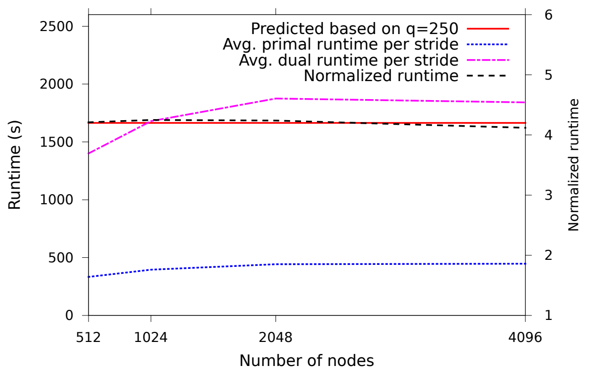


Figure 9. Runs conducted with a stride size of 250 and 50 memory checkpoints per stride. The average runtime of the forward run and adjoint run per stride represent the weak scaling behavior of Nek5000. The normalized runtime stays constant with increasing number of nodes.

Table I
ESTIMATED STRIDE SIZE FOR GIVEN PROBLEM SIZE

Measured		**Estimated**		
Size (GB)	**Avg. Bw. (GB/s)**	**Nodes**	**Stride Size (ts)**	**Norm. runtime**
95	1.61	2048	59	3.3
189	2.07	4096	91	3.7
377	2.02	8192	186	4.0
754	1.87	16384	403	4.3
1507	2.10	32768	717	4.4
7533	2.25	32768*5	3348	4.8

model for any problem size and shows the scalability of our approach. While more runs are desirable to strengthen the validation case, we point out that Fig. 9 took half a million CPU-hours on Mira, whereas the runs to calibrate the prediction took a mere 40,000 core-hours.

VI. CONCLUSION

We present a new online and scalable two-level checkpointing scheme for large scale time-dependent adjoint computations and we demonstrate it for the highly scalable fluid dynamics code Nek5000. The approach makes use of all the system's resources, from local node memory down the memory hierarchy to the archive. At the lower level, it uses the `revolve` algorithm which requires a given stride—a prescribed number of time steps. At the upper level, we use an algorithm that adapts the stride size to the disk/archive bandwidth and that can accommodate an a priori unknown number of total time steps.

In addition, we present a performance model for our approach that requires only a few inexpensive measurements on the target architecture for the same n_P (number of degrees of freedom per process) as the large-scale target run. With this model, we can predict the adjoint to forward run performance ratio and thus the total CPU time of a problem of arbitrary size with the same n_P. We validated our model with several experiments with a driven cavity example where the measured errors were within 2% in the entire parameter range, which included the use of $4096 * 32 = 131072$ parallel processes. Based on this performance model, we also estimate that our two-level checkpointing approach offers significant benefits compared with single-level checkpointing; for example, our approach needs only 75% recomputations compared with single-level checkpointing for simulations exceeding 50,000 time steps of Nek5000.

This paper has empirically verified that large-scale nonlinear adjoint computations are feasible with a predictable performance even in a high-latency environment with a deep memory hierarchy. We note that, as a result, the adjoint calculation needs about a factor of 5 CPU time overall more than a forward run, while producing in principle n times more information about a chosen metric of interest (in our case, total terminal flow energy), where n is the number of degrees of freedom.

VII. Outlook

This work started as an exploratory overview for a generic adjoint capability implementation in Nek5000. The presented work focuses on a continuous adjoint within our two-level checkpointing scheme. An alternative to continuous adjoints would be discrete adjoints, for which we would need to verify potential numerical and runtime differences. In particular, in coupled models or boundary conditions with no adjoint formulation, discrete adjoints are desirable. The third author has developed discrete adjoints as well as the two-level checkpointing scheme in PETSc [1] and more results will be presented in future publications.

Optimal checkpointing schemes remain an open issue. Clarification of the optimality of this algorithm or variations thereof with different assumptions is required. A first step would be the separate treatment disk and archive leading to a three-level scheme.

Specific issues linked to the large-scale setting also need to be addressed, such as increasing the efficient use of system resources. Compression of checkpoints is another means of trading disk memory for computational resources. Additionally, noise is a major issue on large-scale systems, since resources are rarely used solely by one application. Our adaptive stride approach is a first attempt at addressing that issue. We note, however, that if storing and restoring transfer times are not between specific bounds, the efficiency of this algorithm is dramatically reduced. More flexible approaches with respect to system noise need to be explored.

Acknowledgment

We thank Philipp Schlatter, KTH, Sweden for his invaluable suggestions and insights in the field of fluid dynamics. We also thank the entire Nek5000 group at Argonne for their continuous help on the inner workings of Nek5000 and Mira compute time. This material was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

[1]http://www.mcs.anl.gov/petsc/

References

[1] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. Philadelphia, PA: SIAM, 2008, no. 105.

[2] ——, "Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation," *ACM Transactions on Mathematical Software*, vol. 26, no. 1, pp. 19–45, Mar. 2000.

[3] P. Stumm and A. Walther, "New algorithms for optimal online checkpointing," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 836–854, 2010.

[4] V. Heuveline and A. Walther, "Online checkpointing for parallel adjoint computation in PDEs: Application to goal-oriented adaptivity and flow control," in *Euro-Par 2006 Parallel Processing*. Springer, 2006, pp. 689–699.

[5] Q. Wang, P. Moin, and G. Iaccarino, "Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 2549–2567, 2009.

[6] P. Stumm and A. Walther, "MultiStage Approaches for Optimal Offline Checkpointing," *SIAM Journal on Scientific Computing*, vol. 31, no. 3, pp. 1946–1967, 2009.

[7] G. Aupy, J. Herrmann, P. Hovland, and Y. Robert, "Optimal multistage algorithm for adjoint computation," Research Report RR-8721, LIP - ENS Lyon; Argonne national labortory, Tech. Rep., 2014.

[8] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*, ser. Computer Communications and Networks. Springer International Publishing, 2015.

[9] G. K. El Khoury, P. Schlatter, A. Noorani, P. F. Fischer, G. Brethouwer, and A. V. Johansson, "Direct numerical simulation of turbulent pipe flow at moderately high reynolds numbers," *Flow, Turbulence, and Combustion*, vol. 91, no. 3, pp. 475–495, 2013.

[10] A. Peplinski, P. Schlatter, P. Fischer, and D. Henningson, "Stability tools for the spectral-element code Nek5000: Application to jet-in-crossflow," in *Spectral and High Order Methods for Partial Differential Equations-ICOSAHOM 2012*. Springer, 2014, pp. 349–359.

[11] J. Lottes and P. Fischer et al., "Nek5000: User's manual," Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-351, 2015.