Efficient Parallel Streaming Algorithms for large-scale Inverse Problems

Hari Sundar
School of Computing
University of Utah
Email: hari@cs.utah.edu

***Abstract*—Large-scale inverse problems and uncertainty quantification (UQ), i.e., quantifying uncertainties in complex mathematical models and their large-scale computational implementations, is one of the outstanding challenges in computational science and will be a driver for the acquisition of future supercomputers. These methods generate significant amounts of simulation data that is used by other parts of the computation in a complex fashion, requiring either large in-memory storage and/or redundant computations. We present a streaming algorithm for such computation that achieves high performance without requiring additional in-memory storage or additional computations. By reducing the memory footprint of the application we are able to achieve a significant speedup ($\sim$3x) by operating in a more favorable region of the strong scaling curve.**

1. Introduction

The exponential increase in the quantity of measurements and data holds tremendous promise for data-driven scientific discoveries. However, much of data remains unused as extreme-scale inversion and UQ methods—as a systematic tool to infer knowledge from data—are unable to scale up to the amount of data being generated. While significant research has been done on various aspects of large-scale UQ problems, little work has been done on developing large-scale UQ algorithms that are both computationally as well as data scalable. Thus, there is a critical need to develop effective strategies to attack this crucial class of data-scalable UQ algorithms in order to continue the pace of scientific discoveries and to promote the progress of science.

Scalable algorithms for large-scale inversion and UQ for complex systems typically require high order information such as gradient and Hessian, which in turn can be computed efficiently using adjoint techniques. However, the major compute-bound bottlenecks for these methods are the forward and adjoint PDE problems. Contemporary approaches have focused on optimizing the computational efficiency for these PDEs. However, given the large amount of data generated during forward solves and the need to save and reuse it during adjoint solves to compute gradient and Hessian, it is equally important to develop scalable algorithms to manage *simulation data*. This is a problem on modern systems because the computational efficiency of the overall problem is dictated by the linearized pde solve or matrix-vector product as part of a single time-step. The overall memory footprint is however several orders of magnitude larger, leading to the computations operating in the absolute extreme (worst) strong scaling or in some cases being unable to compute on existing hardware without algorithmic changes. Currently, the most common strategy for dealing with this is to use checkpointing, which saves simulated data at only specific instances of time [1], [2]. from which the solution histories are regenerated as needed. However, these savings in memory-footprint and computational efficiency come at the cost of additional computations and limited data-scalability. Algorithmically the simulation-data generation and re-use for gradient and Hessian computation is serial in nature and therefore can be modified to a streaming algorithm. For scalability, the streaming has to performed to and from a node-local disk. Additionally, since SSD would be cost prohibitive, the streaming algorithm has to ensure that the (now ubiquitous) multiple cores effectively streaming to a single node-local disk. The development of such a streaming algorithm for the efficient computation of gradients and Hessians using adjoint techniques for large-scale inversion and UQ methods is the main contribution of this work. By efficiently overlapping the streaming in and out of simulation-data with simulations to produce and use new data, we are able to get the same performance as if we had all simulation data stored in memory. It is our expectation that this work would lead to additional research on application-controlled paging of data to improve the scalability of large-scale data-bound applications.

Organization of this paper: The rest of the paper is organized as follows: In §2 we present a simplified mathematical outline of an inverse problem, highlighting the cost of computing derivatives. In §3 we present our alternative approach to stream out and in simulation data in lieu of checkpointing along with experimental validation presented in §4. Finally we conclude in §5.

2. Cost of using derivative information

In this section, we will introduce the mathematical machinery neccessary for solving a PDE-constrained optimization problem using simple example problems. Consider a nonlinear least squares problem in which z are the uncertain parameters, and $\mathbf{s}_k$ is the state (typically the solution of a

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

partial differential equation). We will present two common scenarios, the first where we are interested in an initial condition based on a later observation and the second where the uncertain parameters are the material properties.

2.1. The parameter is the initial condition

Let $\{\mathbf{s}_k\}_{0,m} \in \mathbb{R}^n$ be the state given by:

$$\begin{aligned}
\mathbf{s}_0 &= \mathbf{z} && \text{(initial condition)} && (1)\\
\mathbf{s}_{k+1} &= \mathbf{A}_k\mathbf{s}_k + (\mathbf{1}^T\mathbf{s}_k)\mathbf{s}_k,\ k = 0,\dots,m-1, && \text{(evolution)} && (2)\\
f &= \mathbf{b}^T\mathbf{s}_m. && \text{(observations)} && (3)
\end{aligned}$$

This is a nonlinear discrete dynamical system resembling the (non adaptive) discretization of a PDE. Here $\mathbf{A}, \mathbf{b}$ are given and $\mathbf{1}$ is the vector of all ones. Therefore the forward map $\mathbf{F}$ is defined by the solution of the dynamical system to obtain $\mathbf{s}_m$ followed by extraction of quantity of interest f.

To compute the derivative $\mathbf{g} = \partial\mathbf{F}/\partial\mathbf{z}$, a D-dimensional vector, we can use either the sensitivity matrix, or the adjoint equation. Using the sensitivity formulation g_i can be computed by solving,

$$\begin{aligned}
\hat{\mathbf{s}}_0 &= \mathbf{e}_i\\
\hat{\mathbf{s}}_{k+1} &= \mathbf{A}_k\hat{\mathbf{s}}_k + (\mathbf{1}^T\hat{\mathbf{s}}_k)\mathbf{s}_k + (\mathbf{1}^T\mathbf{s}_k)\hat{\mathbf{s}}_k,\ k = 0,\dots,m-1,\\
g_i &= \mathbf{b}^T\hat{\mathbf{s}}_m.
\end{aligned} \tag{4}$$

No checkpointing is needed as we march s and $\hat{s}$ simultaneously in time. But we have to solve this for each i, so we have D linearized forward solves. For large D this is prohibitively expensive. While such an approach works for many large-scale machine learning problems (with small D), most large-scale UQ problems have $D > 10^6$ making the sensitivity formulation infeasible. For such problems, the adjoint formulation works better.

Using the adjoint formulation, we can compute $\mathbf{g}$, as follows. Assuming we have solved (1) and have stored the state, the adjoint problem is given by:

$$\begin{aligned}
\mathbf{w}_m &= -\mathbf{b} && \text{(terminal condition)} && (5)\\
\mathbf{w}_k &= \mathbf{A}_k^T\mathbf{w}_{k+1} + (\mathbf{1}^T\mathbf{s}_{k+1})\mathbf{w}_{k+1}\\
&\quad + (\mathbf{1}^T\mathbf{w}_{k+1})\mathbf{s}_{k+1}, && \text{(backward evolution)} && (6)\\
\mathbf{g} &= \mathbf{w}_0. && \text{(derivative)} && (7)
\end{aligned}$$

Notice that this requires only one backward solve (independent of D), but requires storing all $\mathbf{s}_k$. For large m this can be prohibitive, leading to the use of checkpointing strategies. Standard checkpointing strategies either (1) Store $\sqrt{m}$ states and recompute as we go backwards; in which case the cost of the adjoint solve is equivalent to two forward solves, or (2) store L states and use recursion until $L^q/m = 1$. In this case the total storage is $O(L\log(m)/\log(L))$ and the cost of the adjoint solve is $O(2\log(m)/\log(L))$. These can be significant if m, n are large. However, assuming we can store everything, the savings over checkpointing can be 2x or more.

If however, one uses the Hessian $\mathbf{H} = \partial^2\mathbf{F}/\partial\mathbf{z}^2$, checkpointing can lead to significant storage savings. A Hessian matrix vector multiplication $\mathbf{H}\hat{\mathbf{z}}$ is computed as follows.

$$\begin{aligned}
&\text{(Forward)}\\
\hat{\mathbf{s}}_0 &= \hat{\mathbf{z}}\\
\hat{\mathbf{s}}_{k+1} &= \mathbf{A}_k\hat{\mathbf{s}}_k + (\mathbf{1}^T\hat{\mathbf{s}}_k)\mathbf{s}_k + (\mathbf{1}^T\mathbf{s}_k)\hat{\mathbf{s}}_k,\ k = 0,\dots,m-1,\\
&\text{(Adjoint)}\\
\hat{\mathbf{w}}_m &= 0,\\
\hat{\mathbf{w}}_k &= \mathbf{A}_k^T\hat{\mathbf{w}}_{k+1} + (\mathbf{1}^T\hat{\mathbf{s}}_{k+1})\mathbf{w}_{k+1} + (\mathbf{1}^T\mathbf{s}_{k+1})\hat{\mathbf{w}}_{k+1}\\
&\quad + (\mathbf{1}^T\hat{\mathbf{w}}_{k+1})\mathbf{s}_{k+1} + (\mathbf{1}^T\mathbf{w}_{k+1})\hat{\mathbf{s}}_{k+1},\ k = m-1,\dots,0,\\
&\text{(Hessian matvec)}\\
\mathbf{H}\hat{\mathbf{z}} &= \hat{\mathbf{w}}_0.
\end{aligned} \tag{8}$$

Notice that we read $\mathbf{s}_k$, solve for $\hat{\mathbf{s}}_k$ and then read $\mathbf{s}_k, \hat{\mathbf{s}}_k, \mathbf{w}_k$ and solve for $\hat{\mathbf{w}}_k$. The cost of the Hessian matvec with storing everything is again two forward solves. But we need to store $\mathbf{s}_k, \mathbf{w}_k, \hat{\mathbf{s}}_k$. The checkpointing strategies in this case are: (1) Store $\sqrt{m}$ states and recompute as we go backwards resulting in the cost of the Hessian matvec being equivalent to six forward solves. (2) Store L states and use recursion until $L^q/m = 1$. The storage is $O(L\log(m)/\log(L))$ and the Hessian matvec is $O(6\log(m)/\log(L))$. Again, these can be significant if m, n are large. However, assuming we can store everything, the savings over checkpointing can be 3X or more. If we have $m = 10^5$ time steps and $L = 100$, the savings over nested checkpointing for the Hessian are almost 10X, an order of magnitude. We can also assume that we can store more frequently but we need to do more computation to reconstruct the full field. We can have a compression and decompression factor for the saved fields that can enter the calculation.

2.2. The case in which z is a material property

Now we consider the case in which the uncertain parameters are a material property. We compute derivatives of a functional with respect to parameters $\mathbf{z} \in \mathbb{R}^N$, which enter in the time stepping operator. This problem structure corresponds, for instance, to an inverse wave propagation problem, where first and second derivatives—computed through adjoints—are critical for seismic inversion, exploration and for quantifying the uncertainty in the inversion [3], [4], [5], [6]. Let us assume m time steps and denote by $\{\mathbf{s}_k\}_{k=0}^m \in \mathbb{R}^n$ the corresponding state vectors. Then,

$$\begin{aligned}
\mathbf{s}_0 &= \mathbf{0}, && \text{(initial condition)}\\
\mathbf{s}_{k+1} &= \mathbf{A}(\mathbf{z})\mathbf{s}_k + \mathbf{q}_k,\ k = 0,\dots,m-1, && \text{(evolution)}\\
f &= \mathbf{b}^t\mathbf{s}_m. && \text{(observations)}
\end{aligned} \tag{9}$$

Here, $\mathbf{q}_k$ is a forcing term that also may include information related to high-order time stepping. For simplicity, we assume that $\mathbf{A}$ is independent of k and that we have observations only in the final time. More observations can be used, but this does not change the overall cost of the adjoint and Hessian matvecs. Moreover, we assume that the

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

dependence of $\mathbf{A}$ on $\mathbf{z}$ is linear, so $\mathbf{A} = \sum_j \mathbf{A}_j \mathbf{z}_j$, where $\mathbf{A}_j = d\mathbf{A}/d\mathbf{z}_j$ and $\mathbf{z}_j$ is the j_{th} component of $\mathbf{z}$. To compute the derivative $df/d\mathbf{z}$ efficiently, we need to solve the adjoint problem:

$$\begin{aligned}
\mathbf{w}_m &= -\mathbf{b} && \text{(terminal condition)} \\
\mathbf{w}_k &= \mathbf{A}^T(\mathbf{z})\mathbf{w}_{k+1}, \ k = m-1, \ldots, 1, && \text{(backward evolution)} \\
\mathbf{g} &= \sum_{k=1}^{m-1} \sum_{j=1}^{L} \mathbf{e}_j (\mathbf{w}_{k+1}^T \mathbf{A}_j \mathbf{s}_k). && \text{(derivative)}
\end{aligned} \tag{10}$$

where $\mathbf{e}_j$ is the j_{th} basis vector. A Hessian matrix vector multiplication $\mathbf{H}\hat{\mathbf{z}}$ is computed as follows.

$$\begin{aligned}
\hat{\mathbf{s}}_0 &= \mathbf{0} \\
\hat{\mathbf{s}}_{k+1} &= \mathbf{A}(\mathbf{z})\hat{\mathbf{s}}_k + \mathbf{A}(\hat{\mathbf{z}})\mathbf{s}_k, \quad k = 0, \ldots, m-1, \\
\hat{\mathbf{w}}_m &= \mathbf{0}, \\
\hat{\mathbf{w}}_k &= \mathbf{A}^T(\mathbf{z})\hat{\mathbf{w}}_{k+1} + \mathbf{A}^T(\hat{\mathbf{z}})\mathbf{w}_k, \quad k = m-1, \ldots, 0, \\
\mathbf{H}\hat{\mathbf{z}} &= \sum_{k=1}^{m-1} \sum_{j=1}^{L} \mathbf{e}_j (\hat{\mathbf{w}}_{k+1}^T \mathbf{A}_j \mathbf{s}_k + \mathbf{w}_{k+1}^T \mathbf{A}_j \hat{\mathbf{s}}_k).
\end{aligned} \tag{11}$$

If we drop the terms with $\mathbf{w}_k$, we obtain the Gauss-Newton approximation to the Hessian. Let us now summarize the basic computation and communication cost for the individual kernels in the above computations.

TABLE 1. *Complexity estimates for operatioons required for gradient computation and Hessian-vector application. We denote by N the overall number of state degrees of freedom, and by P the number of distributed-memory parallel processes. Moreover, cp is a constant modelling checkpointing. The value $cp = 0$ corresponds to the situation where no checkpointing is used and all forward steps are stored in memory; $cp = 1$ corresponds to a simple checkpointing strategy, where every $\sqrt{K}$th step is stored in memory, and the forward steps in between are recomputed as needed during the gradient computation. If checkpointing is used in the the Hessian-MatVec, we have assumed that checkpoints for the state $\mathbf{s}_k$ and the adjoint $\mathbf{w}_k$ fit into memory and are available. The Gauss-Newton Hessian $\hat{\mathbf{H}}$ does not require the adjoint variable $\mathbf{w}$, which reduces the complexity of a MatVec with $\hat{\mathbf{H}}$ compared to a MatVec with $\mathbf{H}$.*

	flops	comm
$\mathbf{A}(\mathbf{z})\mathbf{s}$, $\mathbf{A}^T(\mathbf{z})\mathbf{w}$	$M_f \sim \mathcal{O}(N/P)$	$M_c \sim \mathcal{O}((N/P)^{2/3})$
forward/adjoint solve	$S_f = KM_f$	$S_c = KM_c$
$\mathbf{g}$	$(2 + 1 + cf)S_f$	$(2 + 1 + cf)S_c$
$\mathbf{H}\hat{\mathbf{z}}$ (Hessian-MatVec)	$(6 + 5cf)S_f$	$(6 + 5cf)S_c$
$\hat{\mathbf{H}}\hat{\mathbf{z}}$ (GN-Hessian-MatVec)	$(4 + 3cf)S_f$	$(4 + 3cf)S_c$

3. Efficient streaming of simulation data

As discussed in §2, checkpointing partially mitigates the issue of the need to temporarily store large amounts of simulation data for subsequent use in the calculation of gradients and Hessian matvecs. Yet, this comes with an expense of carrying out an additional forward solve to compute the gradient and five, instead of two, forward/adjoint solves to compute a Hessian-vector product. This is illustrated in Figure 1. The whole inversion and UQ process are dominated by these additional forward/adjoint solves, and this prevents us from solving higher resolution seismic inversions to obtain better results. An alternate way to look at this is that given certain computing resources, we can estimate a higher resolution solution, if we are able to avoid these additional PDE solves arising from checkpointing.

In this work we propose an innovative streaming algorithm to avoid checkpointing and hence extra forward/adjoint solves. Our idea here is to exploit the causality of the time stepping scheme. For clarity in presenting the idea, ignore the spatial discretization and let us employ the forward Euler for time discretization so that the solution $u(x, t_{j+1})$ at the $(j+1)$th time step can be expressed as $u(x, t_{j+1}) = F(u(x, t_j))$, where $F(u(x, t_j))$ contains information at the jth time step. Clearly, in order to compute $u(x, t_{j+1})$ one needs information only from the immediate previous time step j. Thus, we start storing $u(x, t_j)$ as soon as it is available while advancing the solution to the next time step. Clearly, such a sequential access is amenable to streaming out to disk (write). The same is true during the adjoint solve, where we need to stream in (read) the forward solution from disk.

The *key challenge* is to ensure optimal performance and scalability on current and future architectures. Given that modern supercomputers consist of multicore processors and need to share a single local disk, the data being streamed out from each process needs to be interleaved so that the disk is still accessed sequentially. In this work, we evaluated our methods at the **Stampede** supercomputer at TACC, due to the availability of local disks. We have in the past [7] made the case for the need for local disks, and this work will further that cause. Streaming data to and from local-disk allows us to reduce the memory footprint without paying a computational penalty. Since IO is significantly slower than memory or network access, it is important that the cost of IO be hidden by effective overlap with computation.

For the gradient computation, we need to solve the adjoint equation. Using the backward Euler time discretization the adjoint solution $v(x, t_j)$ at the jth time step is given by $v(x, t_j) = G(v(x, t_{j+1}), u(x, t_{j+1}))$, where $G(v(x, t_{j+1}), u(x, t_{j+1}))$ contains information, including the forward solution as the forcing, at the $(j+1)$th time step. Therefore, while performing the computation for $v(x, t_j)$ we start loading the forward solution $u(x, t_j)$ so that it is already available before the adjoint solver begins the next time step computation. It is important to note that since we do backwards time stepping during the adjoint solve, we will need to convert the data access patterns to locally-forward access to ensure efficient access. Given that the solution data at a specific time step is still stored and accessed in a "forward" manner, this can be done efficiently and in a transparent manner. Note that since we accumulate the gradient in parallel with the adjoint computation [1], [2], the additional computation required for the accumulation will allow more flexibility in this regard.

For the Hessian-vector product computation, we need to solve (11). Similar to the gradient, Euler schemes for time discretization allow us to express the discrete version of (9) and (10) as $\overline{u}(x, t_{j+1}) = F(\overline{u}(x, t_j), u(x, t_j))$ and

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

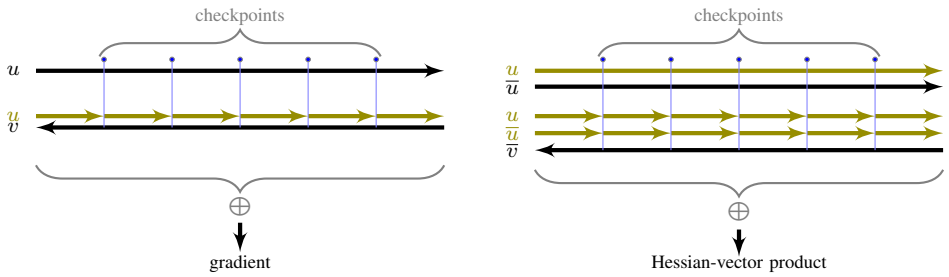

Figure 1. Illustration of the additional PDE solves required for checkpointing during the gradient computation and Hessian-vector products. The additional PDE solves are colored green. Note that 1 pde solve is saved during gradient computation and 3 pde solves are saved during Hessian-vector products. These savings amount to a 2x savings.

$\overline{v}(x,t_j) = G(\overline{v}(x,t_{j+1}), \overline{u}(x,t_{j+1}))$.

While similar to the gradient computation, there are *a few additional challenges to be overcome for performing the Hessian-vector product*. First, we need to compute and store the incremental forward and adjoint solutions at the observation points. Second, the incremental forward equation requires the complete forward solution at the same time step. Thus to *minimize massive data movement*, it is preferable that we solve the forward and the incremental forward PDEs simultaneously and store their solutions in an interleaved fashion. We first solve the forward PDE for a given time instance and then the incremental forward PDE before proceeding to the next time instance. On disk the solutions are saved in an interleaved fashion, i.e.,

$$u(x,t_0), \overline{u}(x,t_0), \ldots, u(x,t_j), \overline{u}(x,t_j), \ldots, u(x,t_T), \overline{u}(x,t_T).$$

This is followed by a simultaneous solve of adjoint and incremental adjoint PDEs, as well as the accumulation of the Hessian. It is important to note that *compared to checkpointing, our strategy reduces the memory footprint, the amount of computation as well as the overall data movement*. The overall speedup is a combination of savings from reduced computations, reduced data movement, as well as higher throughput due to operating in a more favorable region of the strong scaling curve as a result of the reduced memory footprint. We will now describe our implementation.

3.1. Implementation

Our large-scale seismic imaging code is written in `C` and `MPI`. Streaming in and out of memory was handled by the use of a memory mapped file (`mmap`) shared across all MPI tasks running on the node. The `mmap` file was created using the `MAP_SHARED | MAP_POPULATE | MAP_HUGETLB` options. `MAP_HUGETLB` was useful to ensure that sufficient interleaving of data from different tasks was possible, enabling smooth streaming to disks. Additionally, the best performance was obtained using `madvise` with the `MADV_SEQUENTIAL | MADV_DONTNEED` options. This was done to ensure that the file was streamed out immediately to disk. The sequential access might seem counterintuitive since we have multiple tasks (cores) writing to the same shared memory mapped file. This is because, the individual tasks write and read data in an interleaved fashion. Clearly, the effectiveness of the overlap depends on the IO bandwidth and the computational cost of the simulation. At the start of the application, we perform a quick profiling to assess the IO bandwidth for streaming data as well as the rate of generating simulation data. The granularity of overlap between the data generated by the different tasks is determined by this.

Since our code is `MPI` only, we need to determine which ranks are local to each node, so that their relative ordering can be determined for interleaving the data. While OpenMPI supports this via `OMPI_COMM_WORLD_LOCAL_RANK`, other implementations do not readily provide this. Therefore, we perform an initial synchronization using the shared memory mapped file during initialization and determine local ranks. This allows all processed to know the offset at which they should read or write in the interleaved data format.

4. Results

In this section, we present the evaluation for our methods and implementation. All experiments were carried out on the Stampede supercomputer at the Texas Advanced Computing Center, a linux cluster consisting of 6400 computes nodes, each with dual, eight-core processors for a total of 102,400 available cpu-cores. Each node has two eight-core 2.7GHz Intel Xeon E5 processors with 2GB/core of memory and a 3 level cache. Stampede has a 56Gb/s FDR Mellanox InfiniBand network connected in a fat tree configuration. On Stampede, each of the 6400 compute nodes has a single, 250GB commodity SATA hard drive that is used for OS provisioning, local HPC package installation support, and temporary per-job use by user applications via `/tmp`. The amount of `tmp` space available for application use is 69 GB/node; the IO rate to these commodity drives is measured to be 75 MB/sec for large block IO patterns.

Our first experiment was to ensure that effective overlap of IO is possible for our specific problem. In order to assess this, we ran a simple gradient computations with three variants, the first where the entire solution was saved in meory (in-RAM), second, was streamed to disk andthe third variant used uniform checkpointing to disk. We evaluated the performance of all three variants for a simulation where the forward solution that needed to be saved was 7.2GB

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE 2. *Strong scalability results for overlapping IO as an alternative to checkpointing during gradient computation. All runs were performed on the Stampede supercomputer at TACC. We report the runtime in seconds for each case. The (global) size of the forward solution that needed to be saved was 7.2GB.*

cores	in-RAM	Overlap IO	checkpointing
8	12.49	12.85	25.93
16	5.68	5.75	11.88
32	3.02	3.16	6.54
64	1.61	1.65	3.73
128	0.87	0.88	2.36

in size. We ran this experiment in a strong-scaling sense from 8-tasks (1 socket on a node) to 128-tasks (8 nodes) on Stampede. The results are presented in Table 2. We observe that the overlapped-IO variant has similar runtimes as the in-RAM variant and almost identical scalability, both of which are roughly $2\times$ faster than the checkpointed variant. Additionally, the scalability of the checkpointed version is poorer, ending up almost $3\times$ slower than the other two variants on 8 nodes. In larger full-scale UQ problems, the in-RAM variant would not be possible, but the use of the overlapped-IO variant allows us to get similar performance and scalability.

Our second evaluation was using our large-scale seismic imaging application, that is an optimization problem constrained by the acoustic-elastic wave propagation problem. This requires both the gradient as well as Hessian compuations and therefore has a significantly larger memory footprint. This was run on 1024 nodes on Stampede using both the checkpointing as well as overlapped-IO variants. Averaged over 10 runs, the overlapped variant was $2.92x$ faster than the checkpointed variant. We believe this is a combination of reduced computation as well as improved runtime due to reduced memory footprint. We were also able to run the overlapped-IO variant on 512 and 256 nodes, $1.8x$ and $1.04x$ faster than the checkpointed variant on 1024 nodes confirming our initial hypothesis on the effects of poor strong scaling. We used 16 tasks per node for all the runs. These results are plotted in Figure 2.

5. Conclusion

We presented new streaming algorithms for the computation of gradients and Hessians needed for large-scale PDE-constrained optimization problems that are efficient in both in-memory storage as well as the computation compared with standard checkpointing approaches. We demonstrated that we were able to solve a large inverse problem using a quarter of the resources compared to the standard approach of using checkpointing. We believe this will allow us to solve larger problems more efficiently on our largest machines. We also expect this research to pave the way of additional research into application controlled paging and the availability of large node-local storage on high-performance machines.

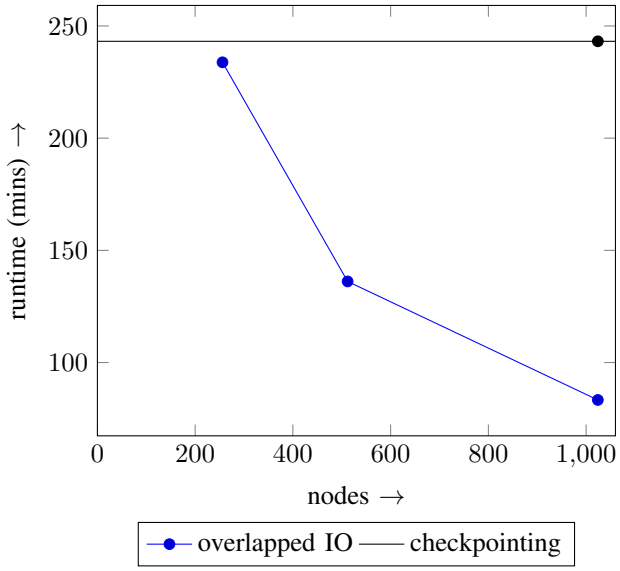

Figure 2. The effectiveness of the overlapped IO technique in improving the performance over checkpointing. Note that due to the memory requirements of checkpointing, it can only be run on 1024 nodes.

References

[1] T. Bui-Thanh, C. Burstedde, O. Ghattas, J. Martin, G. Stadler, and L. C. Wilcox, "Extreme-scale UQ for Bayesian inverse problems governed by PDEs," in *SC12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012, Gordon Bell Prize finalist.

[2] T. Bui-Thanh, O. Ghattas, J. Martin, and G. Stadler, "A computational framework for infinite-dimensional Bayesian inverse problems Part I: The linearized case, with application to global seismic inversion," *SIAM Journal on Scientific Computing*, vol. 35, no. 6, pp. A2494–A2523, 2013.

[3] V. Akçelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. R. O'Hallaron, T. Tu, and J. Urbanic, "High resolution forward and inverse earthquake modeling on terascale computers," in *SC03: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2003, Gordon Bell Prize for Special Achievement.

[4] V. Akçelik, G. Biros, and O. Ghattas, "Parallel multiscale Gauss-Newton-Krylov methods for inverse wave propagation," in *Proceedings of IEEE/ACM SC2002 Conference*, Baltimore, MD, Nov. 2002, SC2002 Best Technical Paper Award.

[5] T. Bui-Thanh, C. Burstedde, O. Ghattas, J. Martin, G. Stadler, and L. C. Wilcox, "Extreme-scale UQ for Bayesian inverse problems governed by PDEs," in *SC12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[6] C. Tape, Q. Liu, A. Maggi, and J. Tromp, "Seismic tomography of the southern California crust based on spectral-element and adjoint methods," *Geophysical Journal International*, vol. 180, pp. 433–462, 2010.

[7] H. Sundar, D. Malhotra, and K. W. Schulz, "Algorithms for high-throughput disk-to-disk sorting," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 93.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE