Parallel Hessian Assembly for Seismic Waveform Inversion Using Global Updates

Scott French*, Yili Zheng†, Barbara Romanowicz‡, Katherine Yelick†§
*National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
†Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
‡Berkeley Seismological Laboratory, UC Berkeley, Berkeley, CA, USA
§Electrical Engineering and Computer Sciences Department, UC Berkeley, Berkeley, CA, USA

***Abstract*—We present the design and evaluation of a distributed matrix-assembly abstraction for large-scale inverse problems in HPC environments: namely, physics-based Hessian estimation in full-waveform seismic inversion at the scale of the entire globe. Our solution to this data-assimilation problem relies on UPC++, a new PGAS extension to the C++ language, to implement one-sided asynchronous updates to distributed matrix elements, and allows us to tackle inverse problems well beyond our previous capabilities.**

Our evaluation includes scaling results for Hessian estimation on up to 12,288 cores, typical of current production scientific runs and next-generation inversions. We also present comparisons with an alternative implementation based on MPI-3 remote memory access (RMA) operations, focusing on performance and code complexity. Interoperability between UPC++ and other parallel programming tools (e.g. MPI, OpenMP) allowed for incremental adoption of the PGAS model where most beneficial. Further, we note that this model of asynchronous assembly can generalize to other data-assimilation applications that accumulate updates into shared global state.

I. Introduction

High-quality models of the earth's interior properties, as seen by seismic waves, have a diverse range of applications, including basic science, geophysical exploration, environmental monitoring, and nuclear-test-ban treaty verification. Full-waveform seismic inversion techniques produce such models by minimizing the misfit between complete recordings (seismograms) of earthquakes or controlled sources and those predicted by numerical simulations, thereby fully exploiting the information content of the wavefield. However, such inversions remain computationally expensive, chiefly due to the large number of numerical simulations required.

One way to reduce the number of simulations is to use a rapidly converging Newton-like scheme to optimize the seismic model. While matrix-free Newton schemes are valuable in the limit of very large problems, direct Gauss-Newton is attractive when Hessian factorization is significantly cheaper than numerical simulation [1]–[3]. However, if the Hessian no longer fits in memory and thus must be distributed, assembly thereof from asynchronous parallel computations may require complex data-dependent communications patterns. This caveat reflects a common motif among scientific applications that assimilate observational data while updating shared global state, where spatially and temporally irregular access patterns can increase software complexity and impede performance if not carefully implemented using one-sided communications.

Here, we present the design of a scalable abstraction for distributed matrix assembly in HPC environments and its integration into an existing scientific application. Our solution uses UPC++ [4], a partitioned global address space (PGAS) extension to C++ that incorporates and builds on many popular features from other PGAS languages, enabling fully concurrent communication and computation while minimizing synchronization. Support for one-sided bulk communication and remote memory management, along with asynchronous remote task execution for custom matrix-update logic, proved fundamental to achieving high performance. Interoperability of UPC++ with other parallel programming models, as required by OpenMP and MPI-based application components (Hessian computation and parallel IO), allowed for incremental adoption of the PGAS model as needed.

The remainder of this paper is structured as follows: In Section II, we discuss our technique for global-scale full-waveform inversion, presenting the underlying problem and motivating our solution. In Section III, we focus on the design of our distributed matrix abstraction. Evaluation follows in Section IV, including scaling analyses for present- and next-generation scales of seismic inversion, as well as comparisons to an alternative MPI-based implementation. Finally, we present scientific results obtained using these techniques in Section V, and conclude in Section VI, where we summarize our contribution and discuss future directions.

II. Computational problem

Waveform inversion has long been a popular technique in the field of global seismology [5], [6], although historically limited to computationally light approximate treatments of wave propagation. As HPC resources have grown more powerful, the spectral element method (SEM) has emerged as a key tool for realistic numerical wavefield simulation from sub-km to planetary scale [7], [8]. Recently, adjoint methods, long popular in exploration seismology [9], have gained wider adoption and been used to implement SEM-based seismic inversion at regional to continental scales [10], [11]. However, adjoint inversions typically employ only first-order (gradient-based) optimization schemes, which converge slowly and require large numbers of wavefield simulations, precluding their use at the global scale where simulation is more expensive. Further, while popular in exploration for their enhanced convergence properties, matrix-free Newton schemes based on second-order adjoint methods [12], [13] and quasi-Newton schemes for first-order adjoint [14] have not yet been able to render global-scale adjoint inversion tractable. Instead, we employ a "hybrid" approach, combining SEM-based wavefield simulation with an efficient physics-based Hessian (and gradient) estimation method, to render global-scale full-waveform inversion tractable [1]–[3]. Here, we provide a detailed overview of our approach, introducing the underlying problem (distributed Hessian assembly) and motivating our solution.

A. *Hybrid full-waveform inversion*

Waveform inversion is an optimization problem: we seek a model $\mathbf{m}$ of Earth's interior properties that minimizes the misfit between observations of the wavefield (seismograms) and predictions given $\mathbf{m}$. We adopt the generalized least-squares criterion [15] and define a misfit function $\chi(\mathbf{m})$ as:

$$2\chi(\mathbf{m}) = \|\mathbf{d} - \mathbf{g}(\mathbf{m})\|^2_{\mathbf{C}_\mathbf{d}^{-1}} + \|\mathbf{m} - \mathbf{m}^p\|^2_{\mathbf{C}_\mathbf{m}^{-1}} \qquad (1)$$

where $\mathbf{d}$ are the observed data, $\mathbf{g}(\cdot)$ is the SEM-based forward operator which predicts $\mathbf{d}$ given $\mathbf{m}$, $\mathbf{C_d}$ reflects uncertainty in the data, and $\mathbf{m}^p$ and $\mathbf{C_m}$ characterize an assumed prior distribution in the model space ($\mathbf{m} \sim \mathcal{N}(\mathbf{m}^p, \mathbf{C_m})$).

Because $\mathbf{g}(\mathbf{m})$ is non-linear in $\mathbf{m}$, $\chi(\mathbf{m})$ must be minimized with either stochastic sampling or iterative optimization. Due to the expense of evaluating $\mathbf{g}(\mathbf{m})$, an iterative technique is invariably chosen. This may be achieved using the Gauss-Newton scheme following naturally from eq 1:

$$\mathbf{m}^{i+1} = \mathbf{m}^i + \left(\mathbf{C_m}\mathbf{G}^T\mathbf{C}_\mathbf{d}^{-1}\mathbf{G} + \mathbb{I}\right)^{-1} \left(\mathbf{C_m}\mathbf{G}^T\mathbf{C}_\mathbf{d}^{-1}\left[\mathbf{d} - \mathbf{g}(\mathbf{m}^i)\right] - \mathbf{m}^i + \mathbf{m}^p\right) \qquad (2)$$

where $\mathbf{G}$ is the wavefield Jacobian, that of the forward operator with respect to the model: $\mathbf{G}_{ij} = \partial\mathbf{g}_i(\mathbf{m})/\partial\mathbf{m}_j$.

The Gauss-Newton approximation arises due to the linearization of $\mathbf{g}(\mathbf{m})$, yielding the Hessian estimate $\mathbf{G}^T\mathbf{C}_\mathbf{d}^{-1}\mathbf{G}$. An advantage of Newton-like schemes over gradient methods (e.g. non-linear CG) is rapid convergence, cutting down on the number of iterations of eq 2 and expensive computations of $\mathbf{g}(\mathbf{m})$. The Gauss-Newton scheme is especially appropriate when: (a) the total iteration count is too small for quasi-Newton methods to build up an accurate inverse-Hessian estimate; and (b) the Jacobian $\mathbf{G}$, or a sufficiently accurate estimate, may be calculated cheaply. We refer to our approach as *hybrid*, because it combines expensive but highly accurate numerical simulations for the forward computation $\mathbf{g}(\mathbf{m})$ and light-weight physics-based estimates of the Jacobian $\mathbf{G}$ to obtain the Gauss-Newton Hessian $\mathbf{G}^T\mathbf{C}_\mathbf{d}^{-1}\mathbf{G}$ (as well as the (negative) misfit gradient $\mathbf{G}^T\mathbf{C}_\mathbf{d}^{-1}\left[\mathbf{d} - \mathbf{g}(\mathbf{m}^i)\right]$).

B. *Estimation of the Hessian*

1) NACT: To derive an estimate of the Jacobian, we use non-linear asymptotic coupling theory (NACT) [6], a formalism based on perturbation theory of Earth's normal modes: the eigensolutions of the equations governing seismic wave propagation. To first order in small perturbations away from a reference spherically symmetric state, the governing equations may be linearized, and an expression for the wavefield in a *perturbed* earth derived. The result of this perturbation is couping between pairs of modes, the effect of which may be computed with a computationally heavy integration over the volume of the whole earth.

NACT provides a way to estimate both the perturbed-earth wavefield and the partial derivatives thereof with respect to $\mathbf{m}$, in a manner that is both accurate (albeit not as accurate as the SEM) and computationally light. Thus, NACT may be used to provide an estimate of $\partial\mathbf{g}(\mathbf{m})/\partial\mathbf{m}$, i.e. the elements of the wavefield Jacobian. Owing to the accuracy of the method, Jacobian estimates from NACT depend non-linearly on $\mathbf{m}$ and must be re-calculated as the iterative inversion proceeds and $\mathbf{m}$ evolves. Therefore, NACT takes additional steps to reduce cost by collapsing the coupling integration over the entire earth onto the great-circle plane joining each earthquake source and seismic receiver. Thus, the main computational kernel of Jacobian estimation is path integration: one for each choice of source-receiver path and mode pair. The cost scales as $O\left(N_{\mathbf{m}_r}\sqrt{N_{\mathbf{m}_{\theta\phi}}}N_{SR}f^4\right)$ where $N_{\mathbf{m}_r}$ and $N_{\mathbf{m}_{\theta\phi}}$ correspond to the radial (depth) and lateral (latitude-longitude) dimensions of $\mathbf{m}$ ($\dim\mathbf{m}$ is $N_\mathbf{m} = N_{\mathbf{m}_r}N_{\mathbf{m}_{\theta\phi}}$), N_{SR} is the number of source-receiver paths, and f is the maximum wavefield frequency (the number of mode pairs grows as f^4). A detailed review of mode perturbation theory and NACT may be found in [16].

2) Practical considerations: For realistically large numbers of data $\dim\mathbf{d} = N_\mathbf{d}$, where $N_\mathbf{d} \gg N_\mathbf{m}$, the Jacobian $\mathbf{G}$ is too large to form explicitly. Instead, we form the $N_\mathbf{m}\times N_\mathbf{m}$ Hessian estimate $\mathbf{G}^T\mathbf{G}$ and (negative) misfit gradient vector $\mathbf{G}^T\left[\mathbf{d} - \mathbf{g}(\mathbf{m})\right]$ directly (we absorb $\mathbf{C}_\mathbf{d}^{-1/2}$ into $\mathbf{G}$ and $\mathbf{d} - \mathbf{g}(\mathbf{m})$ for notational convenience). For each datum i, a particular source-receiver path and recorded seismogram, NACT yields a column-strided panel of $\mathbf{G}$, denoted $\mathbf{G}_{(i)}$. This irregular striding pattern arises from path integration, which limits non-zero elements of $\mathbf{G}$ to model parameters along the source-receiver great circle, and thus depends on the data (via the path geometry). For each datum i of size k (number of time samples), $\mathbf{G}_{(i)}$ is $k \times n$, where n is often much smaller than $N_\mathbf{m}$ ($n \propto N_{\mathbf{m}_r}\sqrt{N_{\mathbf{m}_{\theta\phi}}}$), while k varies independently from n and is ≥ 100 in practice. Thus, for each i, there is an $n \times n$ symmetric update $\mathbf{G}_{(i)}^T\mathbf{G}_{(i)}$ that must be merged into the full $\mathbf{G}^T\mathbf{G}$. The merge operation is the additive "augmented assignment" operator `+=`, and the mapping between elements is given by an indexed slicing operation: `GtG[ix,ix] += GtG_i[:,:]` where `ix` is an

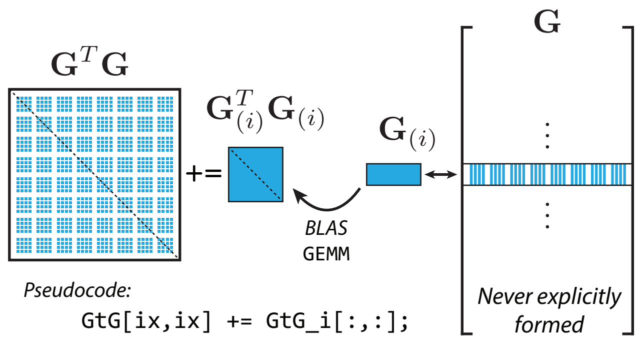


Figure 1. An illustration of the indexed strided-slice update, where $\mathbf{G}_{(i)}$ and $\mathbf{G}^T_{(i)}\mathbf{G}_{(i)}$ are stored in "compressed" form (without zeros).

indexing array (Fig. 1). Updates to the misfit gradient vector from datum i follow a similar pattern (in one dimension).

But why form the Hessian? In theory, one could cache the set of panels $\{\mathbf{G}_{(i)}\}$ and solve eq 2 with a Krylov method, requiring only matrix-vector products. However, relative size of the Hessian (even with $\{\mathbf{G}_{(i)}\}$ "compressed" as in Fig. 1), makes it significantly more practical for our use-case, which typically involves *a posteriori* analysis of the Hessian and repeated solution of eq 2 to test new formulations of $\mathbf{C_m}$. Further, while Krylov methods are clearly attractive when direct factorization of the operator in eq 2 is not feasible, this is not the case for present-day problem sizes, where the former is orders of magnitude cheaper than the SEM simulations that evaluate $\mathbf{g}(\mathbf{m})$ (Section II-C).

3) Parallel implementation with replication: The NACT calculation for each datum is independent of every other. Thus, NACT-based Hessian and gradient estimation is data-parallel and proceeds in two phases: (1) a map operation over the waveform data $\mathbf{d}$ and corresponding predictions from SEM simulations $\mathbf{g}(\mathbf{m})$, yielding per-datum Hessian and gradient contributions; and (2) a parallel reduction operation, yielding a single estimate of the full Hessian and gradient. Our implementation adopts a mixed MPI/OpenMP programming model. The outermost level of parallelism corresponds to MPI tasks, typically distributed among available compute resources one-to-one with NUMA domains (and ensuring local memory affinity). All MPI tasks are equivalent, with the exception that a root task spawns a Pthread responsible for work distribution (assigning data to MPI tasks).

Work is assigned to MPI tasks in data-subsets (more than one datum) that reflect locality of the seismic observation data and simulation output on disk. Subsets are distributed dynamically due to the data-dependent cost of NACT computations (via the frequency dependence). Each subset is processed in parallel by the OpenMP thread team associated with the MPI task (one thread per datum). Up to the limit that the full Hessian estimate can fit in memory, we adopt a replicated approach to reduction of updates, since the merge operation (addition) can be considered associative and commutative for our purposes. At the first reduction level, each MPI task maintains its own copy of the Hessian and gradient, to which the OpenMP thread team applies per-datum updates (Fig. 1). Once all subsets have been processed, the second level of reduction proceeds by summing all replicated Hessian and gradient copies across MPI tasks. Thereafter, the results are saved to disk, either by a single root task or a collective write via MPI-IO if large enough to warrant it.

C. Production hybrid inversion

In our recent global-scale imaging efforts [1]–[3], the dataset is composed of tens of thousands of time-discretized seismograms from hundreds of earthquakes distributed around the globe, yielding an $N_\mathbf{d}$ of $O(10^7)$. The model $\mathbf{m}$ characterizes 3D variations of seismic shear-wave velocity in Earth's mantle, which is expressed in a spline basis of $O(10^4-10^5)$ free parameters (see [1]–[3] for details). Given an iterative model estimate $\mathbf{m}^i$, we first use a the SEM to compute $\mathbf{g}(\mathbf{m}^i)$, after which we compute the Gauss-Newton Hessian and solve eq 2 for $\mathbf{m}^{i+1}$ (see Fig. 2). As noted in Section II-A, the use of a Newton-like optimization scheme means that this procedure is repeated for only a few iterations: often $\leq$ 10. While small, occupying 200-300 CPU cores, the SEM simulations yielding $\mathbf{g}(\mathbf{m})$ are numerous: requiring one simulation per earthquake, per inversion iteration. In contrast, the data assimilation and Hessian computation described in Section II-B is cheap, run once per iteration, and scales extremely well (it is data-parallel). Further, for a single iteration leading to the results featured in Section V and [3], factorization of the Hessian term in eq 2 requires a mere 2300 CPU hours, while SEM simulation requires over 700K hours – clearly demonstrating that factorization is much cheaper than simulation at present-day problem sizes (over two orders of magnitude).

Because we construct the Hessian estimate directly, the space complexity of the replicated approach described in Section II-B3 is independent of the number of data. Instead, this approach requires $O(N_\mathbf{m}^2)$ space per replica, where $N_\mathbf{m}$ is in turn dictated by the resolution of the model. In our recent work, where $N_\mathbf{m} \simeq 2.2\times10^5$, the Hessian now exceeds 90GB in size, even when storing only the upper triangular part at single precision. As such, the Hessian now exceeds available DRAM on a typical shared-memory compute node, rendering the replicated approach in Section II-B3 infeasible (though not for the gradient, which requires $O(N_\mathbf{m})$ space per replica). Further, in next-generation global-scale inversions currently in the planning stage, $N_\mathbf{m}$ is expected to grow by approximately a factor of 4, chiefly due to doubling model resolution in the lateral dimensions. This will yield an $N_\mathbf{m}$ exceeding 8.2×10^5, requiring over 1.3TB to store the Hessian. Thus, both our present-day and future needs clearly motivate the development of a scalable *distributed* dense matrix assembly abstraction tailored to this application.

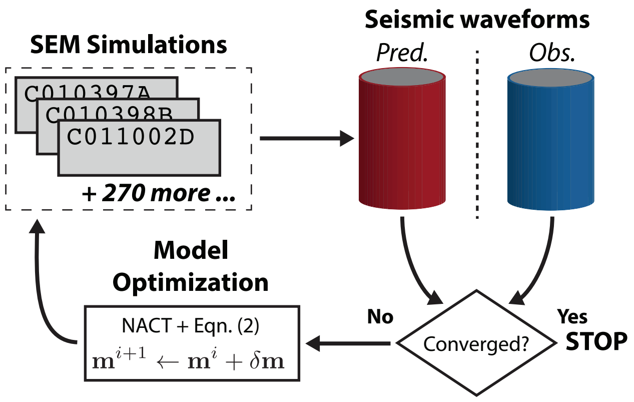


Figure 2. An overview of the iterative waveform inversion procedure. Spectral element (SEM) simulations are used to predict waveforms for the current model estimate $\mathbf{m}^i$, which are in turned compared to the corresponding observed data. If the waveforms agree sufficiently well, the model is considered to have converged with respect to the chosen dataset. Otherwise, the model is updated by solving eq 2 and the process repeats.

III. Abstraction design

A. Requirements

We carefully considered both the requirements intrinsic to our application, as well as those desirable from a usability or scalability perspective. Namely, the resulting abstraction should: **(1)** support distribution schemes common in parallel dense linear algebra (e.g. block-cyclic); **(2)** support updates on indexed slices of the distributed matrix, parameterized by associative-commutative operations; **(3)** provide clear consistency guarantees for updates: namely, that each update should take place atomically with respect to all others; **(4)** attain scalability through overlapped communication and computation with minimal synchronization; **(5)** provide a collective, barrier-like "commit" operation, guaranteeing that all preceding asynchronous updates have been applied; and **(6)** readily interoperate with existing OpenMP/MPI codes. While many of these requirements are constrained by the structure of our application (e.g. interoperation with MPI and OpenMP), others are based on anticipated needs. For example, the distribution scheme requirement reflects a desire to perform in situ parallel linear algebra operations on the resulting Hessian, such as with the ScaLAPACK [17].

B. Implementation

1) UPC++: To enable one-sided updates of distributed matrix elements, PGAS seemed to be an appropriate choice of programming model. Interoperability with existing MPI / OpenMP code made UPC [18], UPC++ [4], and Global Arrays (GA) [19] attractive options. While GA offers one-sided atomic updates of indexed matrix slices, we determined early on that we desired more direct control over how the latter are implemented. Further, UPC++ in particular offers specific features that led us to select it over UPC: **(1)** UPC++ combines one-sided remote memory management and one-sided bulk copies: convenient building blocks for synchronization-free transfer of update data when buffer size is known only on the initiating process; and **(2)** asynchronous remote tasks in UPC++ provide a powerful strategy for encapsulating bulk updates of remote matrix elements in a manner that can be serialized and executed in isolation on the target process (although serialization is not required by UPC++). Like GA, the desired functionality could also be achieved with MPI RMA (for example, `MPI_Accumulate` and `MPI_SUM` over indexed types), but again the generality of UPC++ stands in stark contrast. UPC++ permits the application programmer to explicitly control bulk data movement and custom update logic, and then offload the latter for execution on the target (ensuring consistency guarantees, maximizing data locality, taking advantage of known optimizations, etc.). We will return to this in Section IV-B.

2) Structure: Our abstraction is implemented as a C++ class, with one instance per UPC++ process, referred to as `ConvergentMatrix` (for the ability to "converge" to its final state asynchronously). Typical configurations associate a UPC++ process with a set of OpenMP threads (e.g. in a NUMA domain) that compute matrix updates, which `ConvergentMatrix` then applies. The matrix distribution scheme is modeled on the PBLAS, and parameters thereof (block size, process grid dimensions, etc.) are passed in as template parameters (as is the matrix data type, typically `float` or `double`). The `ConvergentMatrix` constructor takes as arguments the global dimensions of the distributed matrix, allowing the latter to be determined at run time. Further, the constructor is a collective operation, where local storage arrays are initialized and processes exchange configuration data needed in later asynchronous updates.

The abstraction's public interface consists primarily of three methods: `update`, `commit`, and `get_local_data`. In essence, `update` consumes indexed update slices (Fig. 1), bins the update elements by "owner" under the configured distribution, and enqueues their application (+=) on the target owners via UPC++ asynchronous tasks while also preparing for required data movement (see Section III-B3). The `commit` method is a blocking collective that ensures all preceding asynchronous updates have been applied. Thus, updates invoked with `update` are merged into the distributed matrix asynchronously and atomically at some point after their invocation and before a subsequent `commit` returns. Finally, `get_local_data` returns a pointer to matrix storage on the calling UPC++ process, for use with parallel linear algebra libraries or MPI-IO via a suitable `darray` type.

3) Data movement: How update data is moved to target processes deserves careful thought, as it will have a significant impact on execution time of the asynchronous update tasks. Two illustrative scenarios are described below: both employ the same functionality for data transfer, but *differ* in how remote memory is managed. UPC++ supports bulk one-sided RDMA transfers via `upcxx::copy`, which takes source and destination memory references as arguments (`upcxx::global_ptr` objects). This call is blocking, and successful return implies completion. UPC++ also provides `upcxx::allocate` and `upcxx::deallocate` for remote management of globally accessible memory. Both are blocking and one-sided, but unlike `upcxx::copy`, require runtime logic on the target side. With these semantics in mind, the two example scenarios are:

• **Push**: The initiating process could "*push*" the data to the destination process, *before* enqueuing the update task, through successive calls to `upcxx::allocate` and `upcxx::copy`. When the latter returns, buffers containing update data on the initiating side may be freed or reused. Here, `upcxx::global_ptr` references to the transferred data must be passed as arguments to the asynchronous task.

• **Pull**: The initiating process could store the update data to local (but remotely addressable) buffers and pass associated `upcxx::global_ptr` objects to the update task. The latter must then "*pull*" (via `upcxx::copy`) the update data *at execution* onto the target. Importantly, the asynchronous task is responsible for calling `upcxx::deallocate` in order to free the buffers on the initiating process.

Two key metrics to assess the proposed data movement schemes are: (a) safety guarantees for memory usage and (b) "weight" of the update task. Before discussing these points, it is valuable to draw the distinction between truly one-sided operations in UPC++ and those based on active messages (AM). UPC++ uses both the RDMA capabilities provided by GASNet [20], as well as GASNet's AM API to implement polling-based triggers for operations that require target-side logic (implemented via callback handlers). For example, `upcxx::copy` is based on the one-sided `gasnet_put` and `gasnet_get` primitives, while `upcxx::allocate` or `upcxx::deallocate` require AM-based handlers (to execute target memory management operations), as does `upcxx::async` (to enqueue tasks for later execution on the target). With this distinction in mind, a more detailed discussion of the proposed update implementations follows.

First, while the *pull* procedure requires no remote memory management operations within the `update` call on the initiating process, the *push* procedure will block in `upcxx::allocate` until the associated active-messages (AM) handler runs on the target and returns a reference to the receiving buffer. Either approach requires at least one immediate duplication of the storage associated with the binned updates: either a copy retained on the initiating process for asynchronous retrieval by the update task (*pull*), or a copy performed over the network (*push*). Due to the one-sided nature of `upcxx::copy`, there is no risk in the *push* case that a pause in AM handler progress on the target will prevent the next step completing: namely the binned-update buffer being freed or reused. Conversely, in *pull*, the `upcxx::deallocate` call is remote (freeing memory on the initiating side), and must be serviced by AM. Thus, the window during which multiple copies of the update data can consume memory in *push* is determined only by the throughput of `upcxx::copy`, while in *pull* this window could potentially be much larger (if AM handler execution is delayed). Second, to more easily reason about progress in executing the enqueued asynchronous update tasks (Section III-B2), the latter should be as light-weight as possible. This requirement is clearly satisfied more closely in the *push* case, where no communication occurs within the update task. Thus, the `update` method was designed to follow the *push* procedure.

Finally, it is important to note that UPC++ grants the programmer not only fine control over update logic, but also when and how resources (e.g. memory) are used. This stands in contrast, for example, to MPI, where these considerations are often internal to the runtime. This also highlights why PGAS (and UPC++ in particular) is a good fit for our problem, where communication patterns (buffer size, target process, timing) are known only on the initiating side.

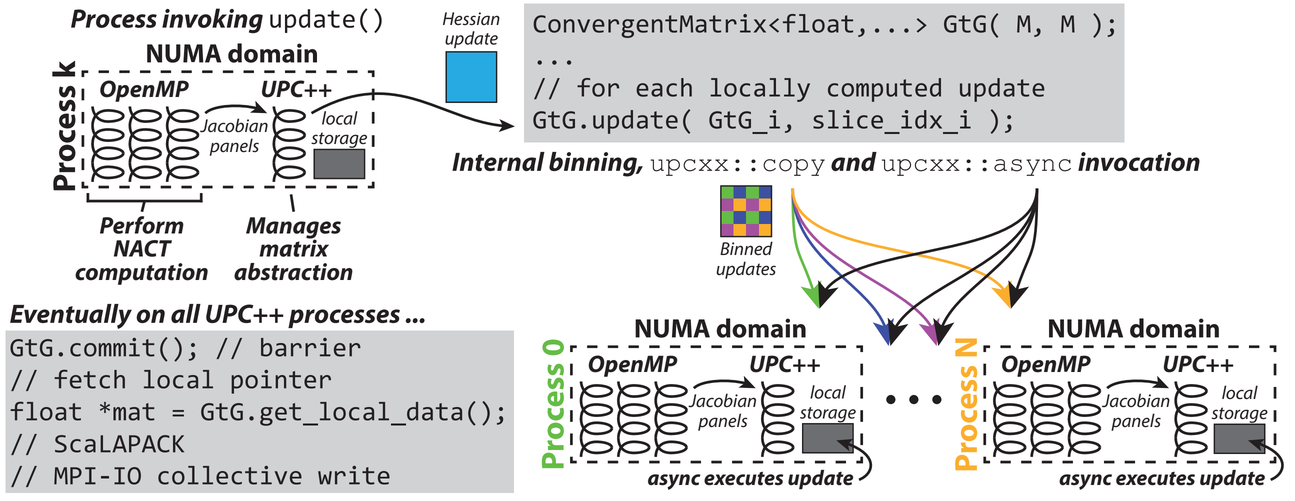
4) Integration: In Fig. 3, we illustrate our distributed Hessian assembly implementation (in contrast to the replicated approach; Section II-B3). Previously, Jacobian panels produced by each OpenMP thread team were consumed by a collocated MPI process, responsible for applying the resulting updates to the local Hessian (or gradient) replica. This MPI-based consumer has now been replaced with a UPC++ process, responsible for managing a `ConvergentMatrix` object to which updates from the OpenMP thread team are applied (replication is still used for the gradient). As before, MPI is used for distribution of work partitions, as well as for high-performance parallel IO. This latter functionality has become increasingly important as the distributed Hessian estimate has grown (soon exceeding 1TB, see Section IV). Further, we note that MPI may also be used indirectly through subsequent calls to the PBLAS / ScaLAPACK, in which case it may be possible to avoid IO before attempting to solve eq 2 (this approach is not currently used, however).

C. Challenges

Here, we discuss two challenges that arose during the development and deployment of `ConvergentMatrix`, along with the particular solutions we adopted.

1) Reasoning about progress: One of the more fundamental challenges we encountered was reasoning about progress: in terms of both execution of the asynchronous update tasks and remote memory management.

As noted above, asynchronous task invocation and remote memory management UPC++ require GASNet to poll the network for new messages on the target side and run the associated AM handlers. GASNet implicitly calls `gasnet_AMPoll` (which services the network) during message-sending operations [20], but reasoning about where, when, and if, additional calls to `gasnet_AMPoll` are necessary is non-trivial (in addition to those internal to



```
ConvergentMatrix<float,...> GtG( M, M );
...
// for each locally computed update
GtG.update( GtG_i, slice_idx_i );
```

```
GtG.commit(); // barrier
// fetch local pointer
float *mat = GtG.get_local_data();
// ScaLAPACK
// MPI-IO collective write
```

Figure 3. A schematic illustration of how the `ConvergentMatrix` abstraction is used in our production application, focused on the path taken by a single Hessian update and highlighting the roles of different coexisting parallel programming models / tools (UPC++, OpenMP, and MPI).

UPC++). Further, even when AM handlers for asynchronous tasks are run on the target process, UPC++ only enqueues these tasks for execution: each process must periodically ensure enqueued tasks execute (by calling `upcxx::advance`, which calls `gasnet_AMPoll` and services the queue).

Pausing to make progress on the enqueued update tasks has important implications for memory overhead, as the tasks are responsible for freeing their own receive buffers (see Section III-B3). Failure to do so can cause `upcxx::allocate` to fail due to memory exhaustion on the target process, halting (at least temporarily, if a wait-retry pattern is used) the movement of update data. While additional `upcxx::advance` calls may be invoked while a UPC++ process waits for new Hessian updates from the OpenMP thread team, there are other operations where such calls cannot easily be interleaved. To this end, we introduced an always-on "progress" thread, responsible for periodically invoking `upcxx::advance`. While this solution requires locks to prevent concurrent calls to UPC++ routines that alter the task queue, this critical region of calls within the `update` method is compact, and the additional code complexity is minimal (less than 30 SLOC). We found this approach to be effective at ensuring progress despite the asynchronous nature of update operations, thereby enabling both high update throughput (rapid execution of update tasks) and efficient memory management (requiring a smaller reserved fast segment for GASNet, as well as less chance of memory exhaustion due to other operations using the heap).

2) Network hotspots: Another challenge was only encountered during benchmarks (Section IV) at high levels of concurrency: network congestion due to simultaneous updates (namely, `upcxx::copy`) against the same target. This can arise when update load and rate are near perfectly balanced, as in the synthetic benchmarks below. Such tests establish a lower bound on performance by constructing a worst-case scenario, where the asynchronous patterns of updates distributed across the application become synchronous (more like all-to-all operations). An effective strategy to mitigate this issue is randomized target ordering, ensuring that (with high probability) no two `ConvergentMatrix` instances will initiate updates on the same set of targets in the same order. An analogous approach has been used to optimize PGAS-based parallel FFT [21], [22].

IV. Evaluation

A. Scaling

Here, we present scaling results for present-day and anticipated next-generation problem sizes, obtained using a synthetic benchmark modeled closely after the real application. This tool is a drop-in replacement for the OpenMP thread team in Fig. 3, yielding streams of artificial updates with realistic sizes, access patterns (indexing into the Hessian), and production rates. As noted in Section III-C2, we configure the benchmark to produce updates at a nearly uniform rate, thus inducing worst-case simultaneous communication volume and placing a lower-bound on performance.

These benchmarks are performed on *Edison*, a Cray XC30 at the National Energy Research Scientific Computing Center and our primary production platform. Each *Edison* compute node has 64 GB of memory among two NUMA domains, each associated with a 12-core Intel "Ivy Bridge" processor. There are 5,576 compute nodes in total, linked via a Cray Aries high-speed interconnect, yielding a peak performance of 2.57 PFLOPS. In our scaling experiments, we mimic the layout of processes / threads seen in the real application: one UPC++ process (and `ConvergentMatrix` instance) per NUMA domain and 8 OpenMP threads performing simulated work (in practice, separate work distri-

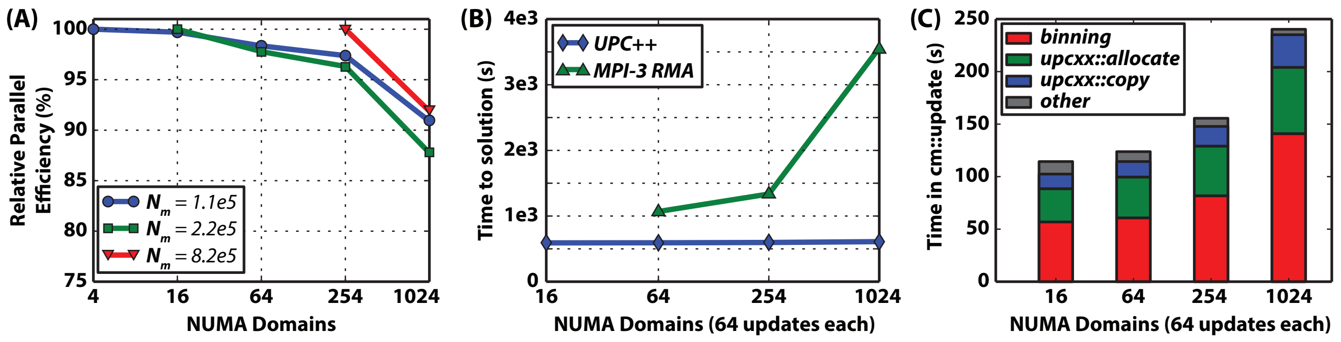


Figure 4. Strong (A) and weak (B) scaling results for the test runs discussed in the text. The UPC++ line in (B) is flat due to near-total overlap of computation and communication. Per-process aggregate walltime breakdowns (C) for the 64 `update` calls in (B), focusing on the three major internal operations (binning, remote allocation, and copying).

bution or IO tasks occupying the remaining cores). Similar to our production application, we use the GNU Compilers (4.8.2) in all of our tests (`-O3`). We store all matrix data in 32-bit `float`, again like the production application (limited by the precision of the seismic data, stored as `float` for compact representation on disk).

1) Strong Scaling: We examine three fixed problem sizes: two from recent inversions ($N_\mathbf{m} = 1.1 \times 10^5$ and 2.2×10^5) and one from a planned next-generation inversion (8.2×10^5) analogous to doubling the lateral resolution of the former two. These runs use a 2D block-cyclic distribution scheme (64×64 block size), occupying P UPC++ processes for $P \in \{2^2, 4^2, 8^2, 16^2, 32^2\}$, analogous to production calculations on up 1024 NUMA domains, or 12,288 cores. We quantify strong scaling in terms of relative parallel efficiency:

$$E_R(P) = \frac{T(P_{min}) \cdot P_{min}}{T(P) \cdot P}$$

where $T(P)$ is the time to solution using P processes (elapsed time from thread-team start to when `commit()` returns) and P_{min} corresponds to the *reference* run: the smallest P in the set above at which the problem size considered can be solved (due to memory limitations). We hold the total number of updates N_{up} initiated across all processes fixed at $N_{up} \in \{4096, 32768, 65536\}$, reflecting present-day and anticipated future inversions and allowing us to measure E_R across three orders of magnitude in core counts by extrapolation. Namely, T scales quasi-linearly with the number of updates initiated by each process, which allows us, for example, to infer $T(P = 4, N_{up} = 32768)$ from $N_{up} = 4096$ (the former takes too long to measure).

To elaborate, our application is partially pipelined: the thread team produces updates in parallel, which are buffered and consumed by `ConvergentMatrix`. There is a non-zero spin-up time at the beginning of each run while the thread team is working but has not yet produced updates. For small P and fixed N_{up} (many updates per instance), the fraction of T spent in spin-up is smaller than for larger P (fewer updates per instance). For example, the $T(P, N_{up})$ ratio $R(P) =$

Table I
STRONG SCALING FOR A RANGE OF $N_\mathbf{m}$ ON UP TO 12,288 CORES OF A CRAY XC30. GREEN VALUES: EXTRAPOLATED WITH $R(P = 64) = 7.88$; BLUE VALUES: $R(P = 64) = 7.80$ (SEE TEXT).

$N_\mathbf{m} = 1.1 \times 10^5$	$N_{up} = 4096$		$N_{up} = 32768$	
P / **Cores**	$T(P)$ s	$E_R(P)$	$T(P)$ s	$E_R(P)$
4 / 48	5070.59	100.0%	39948.20	100.0%
16 / 192	1271.40	99.7%	10016.61	99.7%
64 / 768	322.24	98.3%	2538.74	98.3%
256 / 3072	-	-	640.96	97.4%
1024 / 12288	-	-	171.68	90.9%

$N_\mathbf{m} = 2.2 \times 10^5$	$N_{up} = 4096$		$N_{up} = 32768$	
P / **Cores**	$T(P)$ s	$E_R(P)$	$T(P)$ s	$E_R(P)$
16 / 192	2318.57	100.0%	18079.84	100.0%
64 / 768	592.80	97.8%	4622.56	97.8%
256 / 3072	-	-	1173.27	96.3%
1024 / 12288	-	-	321.92	87.7%

$N_\mathbf{m} = 8.2 \times 10^5$	$N_{up} = 32768$		$N_{up} = 65536$	
P / **Cores**	$T(P)$ s	$E_R(P)$	$T(P)$ s	$E_R(P)$
256 / 3072	2399.96	100.0%	4703.16	100.0%
1024 / 12288	703.72	85.3%	1279.66	91.9%

$T(P, 32768)/T(P, 4096)$ will be $\simeq 8$ for $P = 4$, but less for $P = 64$ (due to the larger spin-up fraction). Here, we can use $R(P = 64)$ to extrapolate a *lower bound* on $T(4, 32768)$ from $T(4, 4096)$, which may in turn be used as the reference to establish a lower bound on E_R for larger P and N_{up}.

In Table I and Fig. 4A, we show T and E_R for the test runs described above. For all $N_\mathbf{m}$ and P considered, we observe impressive relative speedup and find that E_R remains consistently above 85% – indicative of nearly complete overlap of computation and communication. In our application, $N_\mathbf{m}$ is constrained *a priori* by the physics of wave propagation (namely, the attainable resolution) and held fixed for multiple inversion iterations. Thus, strong scaling is an important axis of evaluation for our application. Further, these tests clearly demonstrate that `ConvergentMatrix` readily scales to anticipated next-generation problem sizes.

2) Weak scaling: For our application, it is difficult to define a meaningful notion of weak scaling, tied to a nominal

Table II
WEAK SCALING FOR $N_{\mathbf{m}} = 2.2 \times 10^5$ ON UP TO 12,288 CORES OF A CRAY XC30: UPC++ AND MPI-BASED IMPLEMENTATIONS.

			UPC++	**MPI**
P	**Cores**	N_{up}	$T(P)$ s	$T(P)$ s
16	192	1024	591.18	fail
64	768	4096	592.50	1064.50
256	3072	16384	597.24	1345.66
1024	12288	65536	609.96	3467.65

fixed problem size *per process* while scaling global problem size by enlarging the number of processes. Two natural axes to scale global problem size are matrix dimension $N_{\mathbf{m}}$ and total quantity of data N_{up}. Growing $N_{\mathbf{m}}$ while retaining a fixed-size partition of the distributed matrix per process does not retain a fixed per-process problem size, as the dimension of each update must grow accordingly (Section II-B2). Holding $N_{\mathbf{m}}$ fixed while scaling N_{up} (adding processes, each performing a fixed number of updates), does not maintain the same matrix partition size, but does maintain the same update dimension and per-update communication volume.

Among these two options, we believe the second (scaling N_{up}) may be more informative. Importantly, though the per-update problem size is fixed, the total volume of concurrent communication increases with P, as does the cost of the binning operation. Further, unlike the fixed total N_{up} runs used in assessing strong scaling, these experiments are comparatively insensitive to the effect of spin-up time fraction (which is the same for all P). Thus, these runs should yield a more informative notion of problem size for evaluating the communication model. In Table II and Fig. 4B, we show weak scaling for an $N_{\mathbf{m}}$ of 2.2×10^5 in terms of time to solution T, for a range of P and fixed number of updates per-process (64). We find that T stays nearly constant over a wide range of core- (192–12288) and corresponding update-counts (1024–65535), gaining only 3% at the largest P (and problem size), indicative of near-total overlap of computation and communication. This is confirmed when we examine Fig. 4C, showing per-process time breakdowns for calls to the `update` method: the totals are considerably below computation time for the test case considered (64 updates will be computed by the OpenMP threads in ~ 570 s for this $N_{\mathbf{m}}$). Further, increases in aggregate `update` time are mostly driven by matrix-element binning, *not* the UPC++ update-movement operations.

B. Comparison with MPI-3 RMA

For comparison, we designed a second implementation, based on MPI-3 remote memory access (RMA) operations. While the shortcomings of MPI-2 for emulating PGAS-like functionality are well known [23], MPI-3 largely addresses these issues. The particular semantics of `MPI_Accumulate` fit well with our requirements: concurrent, element-wise atomic updates to remote memory using predefined commutative merge operations (e.g. `MPI_SUM`). These are *weaker* atomicity guarantees than the UPC++ version (which applies the *entire* update atomically), that we hoped could lead to performance advantages when using MPI-3 RMA.

In our re-implementation of `ConvergentMatrix`, we use a single global MPI window object to manage distributed-matrix storage arrays with minimal runtime overhead. Access is managed with the passive `MPI_Win_lock` / `unlock` pattern, requiring minimal synchronization and no target-side intervention. Exclusive locks are acquired for each accumulate call, after initial tests found performance advantages over shared locks, possibly due to implicit coordination between concurrent updates and poor locally when updates from distinct origins are interleaved. We also tried a single passive access epoch with shared-mode locks, opened by the first `update` and closed by `commit`, in conjunction with per-accumulate `MPI_Win_flush_local` calls, but found performance poorer than the exclusive-lock approach. Finally, because individual binned updates are arbitrarily structured, we define per-update MPI *indexed* derived types.

As noted in Section IV-A2, our weak-scaling tests are particularly sensitive to the volume of concurrent communication, and thus provide a useful framework for assessing different communication models. We repeated these tests for the MPI-based abstraction, using the same compiler configuration and Cray MPICH 7.0.3 (based on a heavily optimized MPICH 3.0.3). These *partial* results are shown in Table II and Fig. 4B. MPI's use of a 32-bit `int` for window indexing severely limits window size, placing a lower bound on P for a given $N_{\mathbf{m}}$ (here, $P = 16$ leads to overflow), while element indices may use any integer type in the UPC++ version (default: `long`). We find that time to solution for the MPI-based abstraction is significantly larger than for the UPC++-based one and weak scaling is comparatively poor, with 26% performance degradation between $P = 64$ and 256, and degradation by 226% of the $P = 64$ case at $P = 1024$ (possibly due to an error seen in the low-level communication library at the largest P, causing the pure-RDMA backend to be disabled in favor of a partially RDMA-based protocol). We believe the performance difference might be resolved with further tuning of the MPI implementation for our use case. However, there are semantic differences between the one-sided model in UPC++ and MPI-3, and we currently take advantage of UPC++-specific remote memory allocation and asynchronous tasks.

In terms of code complexity, UPC++ and MPI require analogous initialization steps (exchanging memory references vs. window creation), and similar quantites of code to implement one-sided updates (92 vs. 75 SLOC, respectively). Further, while UPC++ required additional care in reasoning about progress (Section III-C), it was easier to reason about performance (vs. MPI, where details relevant to debugging performance are hidden in the runtime). For example, in the `MPI_Accumulate` implementation in foMPI [24], a high-performance RDMA-aware MPI-3 RMA im-

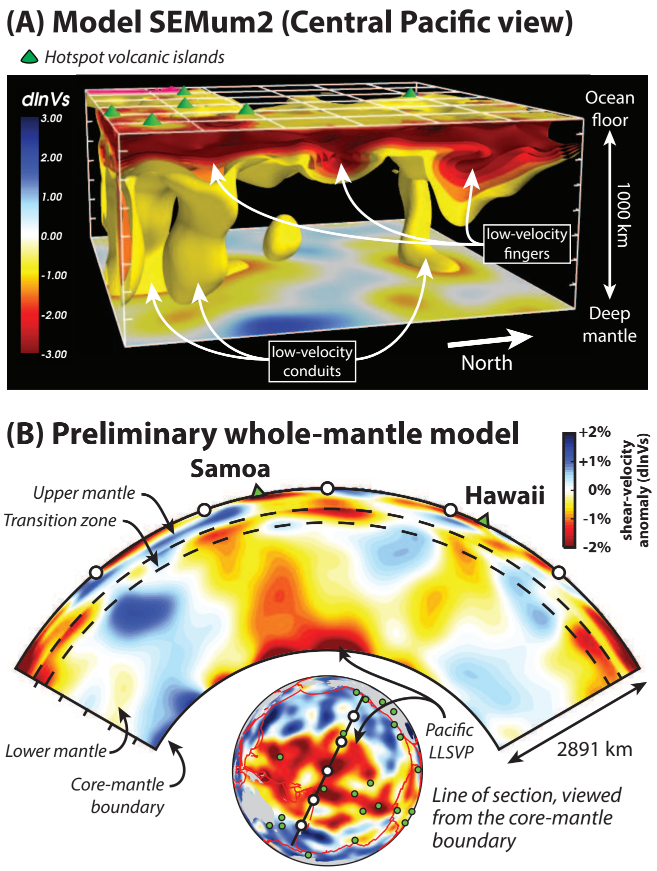


Figure 5. Full-waveform images of shear-wave velocity (V_S) structure in the earth's mantle. Our previous model SEMum2 [2] revealed finger-like low V_S anomalies in the upper mantle, accompanied by conduit-like structures extending below (A). Enabled by the matrix assembly abstraction described here, our preliminary whole-mantle model (B) sheds further light on the origins of these structures, and more broadly the interactions between different scales of convective phenomena – linking surface expression of these processes (hotspot volcanic islands like Hawaii), to the well known large low shear-wave velocity provinces (LLSVPs) in the deep mantle.

plementation, the style of bulk floating-point accumulate that we require involves multiple phases of data-movement (target-window lock, RDMA-get, origin-side accumulate, RDMA-put, unlock), in contrast to our explicit *single* phase. Indeed, this limitation may have its roots in design of the API itself, which encourages a truly passive target (e.g. avoiding extensive target-side buffering in the runtime, which can lead to memory contention). Thus, for certain types of accumulate-like operations (such as our one-sided matrix updates), the parallel programming model exposed by UPC++ has additional advantages owing to its generality.

V. Scientific results

Our new abstraction for scalable distributed matrix assembly has enabled us to solve problems not possible with our previous implementation (Section II-B3). In our earlier work [2], we focused on imaging seismic shear-wave velocity structure of the earth's upper mantle and transition zone ($\leq$ 800 km below the surface) – the limits of feasibility under the replicated approach. Our inversion technique revealed never before seen "fingers" of low seismic velocities in the upper-mantle beneath the world's ocean basins (Fig. 5a). While the images suggested a connection between these structures and columnar low-velocity features extending into the lower mantle, the depth range of our inversion was limited. To more fully examine the interactions between different scales of convective phenomena in the upper and lower mantle, with implications for the dynamics of the system as a whole, we have now moved on to whole-mantle imaging (to the core-mantle boundary at 2891 km depth). This most recent work has required three iterations (eq 2) at the whole-mantle scale ($N_{\mathbf{m}} = 2.2 \times 10^5$) using an enlarged dataset of higher-frequency seismic waveforms to attain better resolution [3]. The inversion completed quickly, owing to the rapid convergence of our Newton-like model optimization scheme, which is in turn enabled by the `ConvergentMatrix` abstraction. Our results are already yielding intriguing new images of coupling between different scales of convection in the earth's upper and lower mantle, as illustrated in Fig. 5b, and encourage us to explore next-generation, higher-resolution imaging characterized by the largest problem sizes discussed in Section IV-A1.

VI. Conclusions

Here, we presented the design and implementation of a distributed matrix abstraction for physics-based Hessian estimation that not only allows us to tackle previously intractable current-generation inversions, but also scales to anticipated next-generation problems. Our solution was enabled by the specific combination of PGAS features provided by UPC++, particularly remote memory management and asynchronous task execution, while still presenting a familiar language (C++) and allowing interoperation with MPI/OpenMP components of our code. We observed impressive scaling behavior based on synthetic benchmark experiments under scientifically meaningful configurations. We found that UPC++ was a natural fit for our one-sided assembly problem, where communication patterns (buffer size, timing, etc.) are known only on the origin side (and only at execution), and that UPC++ outperforms our MPI-3 RMA implementation. Further, this particular approach is quite general, and should be applicable to a broad range of data-assimilation problems arising in HPC applications.

In the near future, we will extend our whole-mantle imaging to higher-resolution inversions, similar in scale to the largest experiments discussed above. We also plan to explore usability improvements and optimizations to `ConvergentMatrix` as our use-case evolves. For example, update bins are currently implemented as append-only indexed arrays, while pre-communication sort / compaction may reduce communication volume if repeated hits to the same elements become more common in our workloads.

The UPC++ and MPI implementations are available from `https://github.com/swfrench/convergent-matrix` and `convergent-matrix-mpi` (commits `16c28d1` and `3338240` were used in our benchmarks, respectively).

Acknowledgment

This work was supported in part by the Department of Energy's Office of Science, Advanced Scientific Computing Research X-Stack Program under Lawrence Berkeley National Laboratory (LBNL) Contract DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the same LBNL contract.

References

[1] V. Lekić and B. Romanowicz, "Inferring upper-mantle structure by full waveform tomography with the spectral element method," *Geophys. J. Int.*, vol. 185, pp. 799–831, 2011a.

[2] S. French, V. Lekić, and B. Romanowicz, "Waveform Tomography Reveals Channeled Flow at the Base of the Oceanic Asthenosphere," *Science*, vol. 342, pp. 227–230, 2013.

[3] S. W. French and B. A. Romanowicz, "Whole-mantle radially anisotropic shear velocity structure from spectral-element waveform tomography," *Geophy. J. Int*, vol. 199, no. 3, pp. 1303–1327, 2014.

[4] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS extension for C++," in *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[5] J. Woodhouse and A. Dziewonski, "Mapping the upper mantle: three dimensional modeling of Earth structure by inversion of seismic waveforms," *J. Geophys. Res.*, vol. 89, pp. 953–5, 1984.

[6] X. Li and B. Romanowicz, "Global mantle shear velocity model developed using nonlinear asymptotic coupling theory," *J. Geophys. Res.*, vol. 101, pp. 22,245–22,272, 1996.

[7] D. Komatitsch and J. Tromp, "Spectral-element simulations of global seismic wave propagation - I. Validation," *Geophys. J. Int.*, vol. 149, pp. 390–412, 2002a.

[8] Y. Capdeville, E. Chaljub, J.-P. Vilotte, and J.-P. Montagner, "Coupling the spectral element method with a modal solution for elastic wave propagation in global earth models," *Geophys. J. Int.*, vol. 152, pp. 34–67, 2003.

[9] A. Tarantola, "Inversion of seismic reflection data in the acoustic approximation," *Geophysics*, vol. 49, pp. 1259–1266, 1984.

[10] C. Tape, M. Liu, A. Maggi, and J. Tromp, "Adjoint Tomography of the Southern California Crust," *Science*, vol. 325, pp. 988–992, 2009.

[11] A. Fichtner, B. Kennett, H. Igel, and H. Bunge, "Full seismic waveform tomography for upper-mantle structure in the Australasian region using adjoint methods," *Geophys. J. Int.*, vol. 179, pp. 1703–1725, 2009.

[12] I. Epanomeritakis, V. Akcelik, O. Ghattas, and J. Bielak, "A Newton-CG method for large-scale three-dimensional elastic full-waveform seismic inversion," *Inverse Problems*, vol. 24, p. 034015, 2008.

[13] L. Métivier, R. Brossier, J. Virieux, and S. Operto, "Full Waveform Inversion and the Truncated Newton Method," *SIAM J. Sci. Comput.*, vol. 35, pp. B401–B437, 2013.

[14] R. Brossier, S. Operto, and J. Virieux, "Seismic imaging of complex onshore structures by 2d elastic frequency-domain full-waveform inversion," *Geophysics*, vol. 74, no. 6, pp. WCC105–WCC118, 2009.

[15] A. Tarantola and B. Valette, "Generalized Nonlinear Inverse Problems Solved Using the Least Squares Criterion," *Rev. Geophys.*, vol. 20, pp. 219–232, 1982.

[16] B. Romanowicz, M. Panning, Y. Gung, and Y. Capdeville, "On the computation of long period seismograms in a 3-D earth using normal mode based approximations," *Geophys. J. Int.*, vol. 175, pp. 520–536, 2008.

[17] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[18] "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[19] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[20] D. Bonachea, "GASNet specification, v1.1," Computer Science Division (EECS), UC Berkeley, Tech. Rep. UCB/CSD-02-1207, 2002.

[21] R. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. A. Yelick, "Scaling Communication-intensive Applications on BlueGene/P Using One-sided Communication and Overlap," in *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[22] G. Almási, C. J. Archer, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng, "Optimization of MPI Collective Communication on BlueGene/L Systems," in *Proceedings of the 19th Annual International Conference on Supercomputing*, 2005.

[23] D. Bonachea and J. Duell, "Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations," *Int. J. High Performance Computing and Networking*, vol. 1, pp. 91–99, 2004.

[24] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling highly-scalable remote memory access programming with MPI-3 one sided," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 53.