Programmable In Situ System for Iterative Workflows

Erich Lohrmann[1], Zarija Lukić[2], Dmitriy Morozov[2], and Juliane Müller[2]

[1] Georgia Institute of Technology, Atlanta, GA;
elohrmann3@gatech.edu
[2] Lawrence Berkeley National Laboratory, Berkeley, CA;
{zarija,dmorozov,JulianeMueller}@lbl.gov

Abstract. We describe an in situ system for solving iterative problems. We specifically target inverse problems, where expensive simulations are approximated using a surrogate model. The model explores the parameter space of the simulation through iterative trials, each of which becomes a job managed by a parallel scheduler. Our work extends Henson [1], a cooperative multi-tasking system for in situ execution of loosely coupled codes.

1 Introduction

The growing gap between the speed of I/O and computation is widely recognized in the HPC community. Already on today's architectures the slow I/O is responsible for major bottlenecks; the problem will only get worse as we move to exascale, where data movement will dominate all design decisions.

The response to this problem is also well-known by now: in situ and in transit processing. If two codes (e.g., simulation and analysis) need to exchange data, they should do so directly, without going through disk. When running on the same nodes, they should share memory and access one another's data directly. When the rates of processing differ, it's logical to run different codes on different nodes, but they should send data directly to each other, without saving it to disk.

A number of in situ frameworks have been designed to address the I/O problem following this principle: ADIOS [2], DataSpaces [3], GLEAN [4], Damaris/Viz [5], ParaView's Catalyst [6], VisIt's libsim [7], Decaf [8], Henson [1], to name a few — we refer the reader to a community effort comparing four of these frameworks [9]. In all cases, the general pattern of the execution is the same: a simulation produces data and passes it to a chain of analysis codes that transform it, identify its salient features, visualize them, or save them to disk in significantly reduced, scientifically meaningful summaries.

Such chained pipelines support *direct simulations*, which have been invaluable tools in computational science: given a complete description of a physical system, they let a user predict an outcome of a measurement. The aim of our paper is to bring a different execution regime to the attention of the community and to describe our (partial) solution to the problems that it presents. We focus on the class of *inverse problems*, where the measurement result is known, and one tries to infer the values of parameters which characterize the underlying physical system. For example, in experiments we present in this paper, we try to reconstruct thermal parameters of the intergalactic gas in the universe using Lyman α power spectrum measurement [10].

As each simulation can be expensive, it is beneficial to explore parameter spaces by constructing surrogate models, i.e., computationally cheap approximations of the simulated phenomena. These models help identify those parameters that are likely to produce the most scientific insight. Crucially, these models facilitate automatic *iterative* parameter sweeps: the decisions about the input parameters to the simulations can be made automatically based on the results of the previous runs. Furthermore, the input data (either observational or finer simulation output) is usually shared between different parameters. Together, these features offer an opportunity for in situ automation, which we explore in this paper.

Related work. Besides the aforementioned in situ frameworks, we briefly note Swift/T [11], a system that allows the user to script complex workflows. It is much more advanced than our work, and we believe can be used to implement the kind of iterative execution described in this paper. We do emphasize one significant difference. To include user code, implemented in C or C++, into Swift/T, the code has to be organized and compiled into Swift modules (via SWIG wrappers). Because we rely on Henson [1], described in the next section, we are able to work with the separate executable directly.

2 Background

Henson. Our solution extends Henson [1], a cooperative multi-tasking system that lets multiple distinct executables run on the same node and share memory, without any changes to their memory management facilities.

Henson is built on two main ingredients: position-independent executables and coroutines. Individual codes are compiled as position-independent executables, making them simultaneously stand-alone executables and dynamic libraries. Henson loads multiple such codes as dynamic libraries, using `libdl` facilities. This puts them in the *same address space*, letting them access each other's memory directly.

The individual codes, referred by Henson as puppets, are treated as coroutines: each one gets its own stack. To coordinate execution, the codes call `henson_yield` function, which returns control to Henson (e.g., after every time step of a simulation). Crucially, when the control returns back to the puppet, its execution resumes exactly where it left off — all state is preserved. This way Henson provides low-overhead context switching and lets the user coordinate execution of multiple codes from an external script.

Henson includes facilities to help puppets exchange data. It provides a shared map of symbolic names to memory addresses to make it easy for puppets to identify important memory segments. For example, if simulation saves an address of an array by calling `henson_save_array("particles", &particles, ...)`, an analysis code can later access this array directly by calling `henson_load_array("particles", ...)` — the memory is shared, so only addresses are exchanged.

Henson also provides auxiliary facilities to help users work with MPI. Different codes can be organized into *execution groups*, which are given symbolic names (e.g., "producer" and "consumer") and are assigned to run on different processes. To exchange data between the two groups (e.g., to support in transit analysis), Henson provides `henson_get_intercomm` function that returns an MPI intercommunicator connecting the two groups.

Henson's major limitation is the domain-specific language used to express its scripts. It supports only while-loops and if-statements: both help express the order in which execution should alternate between the puppets, but are too limited in general. To support more complicated workflows, we have extended Henson to use ChaiScript,[3] a general purpose scripting language, implemented as a C++ header-only library. The interpreter, including its standard library, is compiled directly into Henson. This offers a major benefit over Python (another natural choice): the interpreter does not search for modules on the filesystem; this automatically obviates a major difficulty with using Python in an HPC environment.

Finally, the most significant addition to Henson, made as part of this work, is the addition of a scheduler that lets the user iteratively launch multiple jobs (that themselves can use in situ and in transit processing), depending on the decisions made by one of the puppets, in our case a surrogate model. We describe the scheduler in detail in the next section.

Surrogate models. Surrogate models are computationally cheap approximations of expensive simulations models [12]. They are widely used in derivative-free optimization, when objective function values are computed based on the output of computationally expensive black-box simulation models, and thus no analytic description of the objective function and its derivatives are available. In general, we use the representation $f(x) = s(x)+e(x)$, where $f(x)$ is the expensive objective function, $s(x)$ is the surrogate model, and $e(x)$ is the difference between the two. Surrogate model optimization algorithms start by generating an initial experimental design of size n_0, for example, using Latin hypercube sampling. The expensive function $f(x)$ is evaluated at the points in the initial design, and we fit the surrogate model $s(x)$ to the data pairs $\{(x_i, f(x_i))\}_{i=1}^{n_0}$. Then, in each iteration of the algorithm, we use the surrogate model $s(x)$ to select one or multiple new points x_*, at which we will do the next expensive evaluations. We update the surrogate model with the new data $(x_*, f(x_*))$ and iterate until the stopping criterion has been met. Typically used stopping criteria are a maximum CPU time or a maximum number of allowed expensive function evaluations.

Different surrogate model types have been developed in the literature. We focus here on radial basis function (RBF) models although other models may

[3] `http://chaiscript.com/`

work in our context. An RBF interpolant is defined as follows:

$$s(x) = \sum_{i=1}^{n} \lambda_i \phi(\|x - x_i\|_2) + p(x), \tag{1}$$

where n denotes the number of points for which we have already evaluated the objective function, $\phi(\cdot)$ is a radial basis function (we use the cubic, $\phi(r) = r^3$), and $p(\cdot)$ denotes the polynomial tail (here, $p(x) = a + b^T x, a \in \mathbb{R}, b \in \mathbb{R}^d$, d is the number of dimensions). The model parameters are determined by solving a linear system of equations.

Different strategies have been developed to iteratively select one or more new sample points. For example, Gutmann [13] uses a target value for the surrogate model and defines a merit function which he (cheaply) optimizes in order to determine the next sample point. Regis and Shoemaker [14] use a stochastic approach in which they create candidate points by perturbing the best point found so far and based on scoring criteria, the best candidate is selected for evaluation. Müller and Shoemaker [15] use a similar approach and in addition to candidates created by perturbation, they also create candidates by uniformly sampling points from the whole variable domain. More examples of surrogate model algorithms and their application to engineering design problems can be found in the literature [16].

3 Scheduler

To support iterative workflows, we have added a `Scheduler` class to Henson. Scheduler takes over a given execution group (a set of MPI ranks). It dedicates one process as a controller and the rest as workers. The controller loads a puppet in charge of the overall execution logic (the surrogate model in our case). That puppet generates a set of trials and, over time, receives results of expensive evaluations, updates the model, and generates new trials. Given the trial points from a surrogate model, a user can `schedule` a new job by specifying an arbitrary ChaiScript function to call, together with its arguments, how many processes it needs to execute, and how those processes should be partitioned into execution groups. The job is placed in the queue on the controller process.

The controller maintains the state of worker processes (whether they have a job assigned to them or whether they are available). If there are jobs in the queue and enough available workers to execute them, it sends out the job (the previously queued function) to the workers. When the workers are done with the job, one of them (e.g., the root) returns a value, which is sent back to the controller. The result of the execution (in our case, expensive evaluation $f(x_*)$) is placed in a results queue to be retrieved and processed by the surrogate model. Listing 1.1 illustrates a sample ChaiScript using the scheduler. Figure 1 illustrates a possible break down of processes between execution groups within and outside the scheduler.

We highlight some technical ingredients that are crucial for this system to operate properly. When a set of processes is selected to execute a job, we need to

```
var pm = ProcMap()
var nm = NameMap()

def world(args)
{
    var sim = load("./simulation ...", pm)
    var ana = load("./analysis   ...", pm)

    sim.proceed()
    while (sim.running())
    {
        ana.proceed()
        sim.proceed()
    }

    if (pm.local_rank() == 0)
    {
        var result = nm.get("result")
        return result
    }
}

var sched = Scheduler()
if (sched.is_controller())
{
    var surrogate = load("./surrogate-model ...", pm)
    surogate.proceed()

    // schedule jobs
    for (/* initial trials */)
    { sched.schedule("job-${i}", "world", args,
                     ["all" : 0], sched.workers()/2) }

    while (sched.control())
    {
        if (!sched.results_empty())
        {
            var x = sched.pop()
            // pass x back to the surrogate
            surrogate.proceed()
            // get new trials and schedule new jobs
        }
    }
    sched.finish()  // signal to workers
} else { scheduler.listen() }
```

Listing 1.1. A sample scheduler ChaiScript.

construct an MPI communicator on those processes — this communicator acts as the job's MPI_COMM_WORLD. Unfortunately, all (intra-)communicator creation functions provided by MPI are collective, meaning that all the workers, even those that are not assigned to the given job, have to execute them. In our case, this would mean synchronizing all the workers to create a communicator for a new job — clearly undesirable behavior.

To work around this problem, we use the algorithm of Dinan et al. [17] that allows for non-collective communicator creation — or, more accurately, it's collective only on the processes that participate in the newly created communicator. Unlike MPI_Comm_split that constructs a communicator by splitting a larger communicator, the non-collective algorithm builds a communicator from the bottom up. Starting from MPI_COMM_SELF, it alternates between intra- and inter-communicators, using MPI_Intercomm_create and MPI_Intercomm_merge functions, and merges the local communicators from the participating ranks into the desired communicator. We refer the reader to the original paper [17] for details.

Once the processes construct the communicator, they split it into sub-communicators corresponding to the execution groups specified by the user. Within the job, the user can access the inter-communicators between the groups by calling henson_get_intercomm, mentioned in the previous section. The advantage of this design is that the puppets become oblivious to whether they are running in a job inside the scheduler or over all the ranks. As a result, the user can take advantage of in transit analysis *inside a job*, where separate execution groups are responsible for data generation and analysis. As the earlier work on Henson [1] illustrates, such an execution regime can be beneficial when analysis is computationally expensive: the overhead of data movement pales in comparison to the gains of better strong scaling.

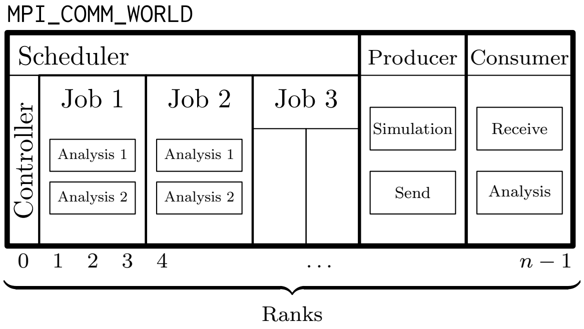

Fig. 1. Schematic partition of the processes with the scheduler.

4 Surrogate Model Experiment

In cosmology, spatial correlations of the Lyman α flux offer a promising route to measuring the cosmological and thermal parameters at high redshifts and small scales [18]. The measure we use is the Fourier-space analog of the two-point correlation function — the power spectrum. Given a cosmological model and a model for the ultraviolet background emission from galaxies, we can predict the resulting flux power spectrum, using the Nyx code [19]. The full parameter space of interest consist of 5 cosmological and 4 thermal parameters; to test our computational workflow system, we work with only 3 thermal parameters, thus significantly reducing the dimensionality of the problem. In addition, instead of running full Nyx simulation for every function evaluation (approximately 100,000 CPU hours), we use outputs of a single run and rescale 3 thermal parameters before calculating the power spectrum (approximately 100 CPU hours). This rescaling is an approximation of what a full Nyx run would yield, and is $\sim$10% accurate (Lukić et al. in prep.), which suffices for the purpose of this work.

We ran our experiments on NERSC's Edison, a Cray XC30 supercomputer with 5,576 nodes with 24 cores each. Each run requested wall clock time of thirty minutes and 3585 processors (150 nodes). This allocation is just enough to run seven simultaneous jobs of 512 processes each, with an extra process reserved for the controller. The input data, a snapshot of a cosmological simulation, is a 193 GB HDF5 file. The individual jobs consist of two separate executables: the power spectrum calculation for the given input parameters and comparison of the resulting spectrum to the given target in L^2-norm. The latter value is returned to the scheduler, which passes it to the surrogate model on the controller process to update its internal state and generate new trials.

Explicit caching. To avoid re-reading the input file, we implemented a standalone puppet that reads the data and stores it in memory. It's executed on all the worker processes before they come under the control of the scheduler. Because every job uses 512 processes, each process requires the same data for every job, and we can pre-load the data, save it in memory, and let individual jobs access it directly without re-loading it from disk. In two separate experiments, the average I/O time was 113.47 and 116.98 seconds. The subsequent calculation of the power spectrum took 322.39 and 423.16 seconds on average, respectively.

Implicit caching. Re-running the job without explicit caching, where each job re-loads the data directly from disk, we identified an implicit caching mechanism: Linux kernel's page cache. The average I/O time for the first seven jobs was 114.70 and 105.25 seconds, across two experiments. However, in the subsequent batch of jobs, executing on the same processes, the I/O time went down to 1.57 and 0.75 seconds. Forcing the kernel to drop its caches by allocating a sufficiently large array brought the average I/O time up to roughly 20 seconds, presumably the rest of the difference (115 vs 20 seconds) is due to other caching within the I/O subsystem.

Although in this case explicit caching offers virtually no benefit, it still has advantages. First, it gives the user explicit control over which data is stored

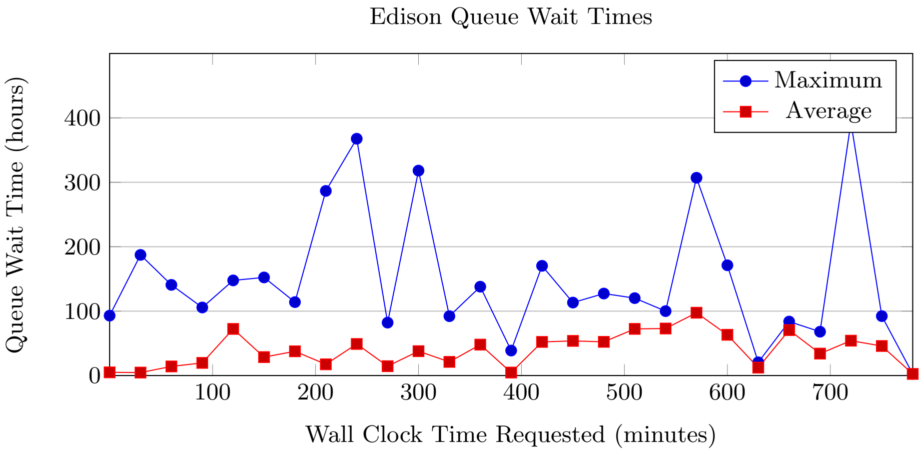

Fig. 2. A graph of queue wait times as a function of requested wall clock. The times tally all requests for 100 to 200 nodes, over six months (February through July 2016).

between job invocations — the kernel is far less predictable. Second, it allows the user to cache data coming not only from disk, but from any other source — in the full simulation pipeline (that we ultimately aim to implement), the data would come directly from the simulation, without being saved to disk first.

Alternatives. One could implement iterative job execution using the supercomputer's workload manager directly (SLURM in case of Edison). Submitting jobs into the queue, waiting for their execution and results, updating the surrogate model, and iterating would incur the extra overhead of queue wait times (and, of course, the extra I/O overhead). To get a fair comparison we consider the difference in queue wait times, when requesting the same number of nodes for a different amount of time.

Figure 2 shows the average and maximum queue wait times, in non-debug queues of Edison, as a function of requested wall clock, for 100 to 200 nodes, over the six months from February through July 2016. The salient point is that the average time grows sub-linearly. For example, requesting the nodes for 60 minutes gave an average queue wait time of 14 hours, while requesting the nodes for 600 minutes resulted in average wait time of 63 hours. In other words, it's advantageous to request more time and manage the jobs within the allocation.

It is possible to run multiple simultaneous jobs within the allocation directly, using MPI's MPMD mode. Besides again incurring I/O overheads, doing so would synchronize the jobs running simultaneously and would force all of them to take as much time as the slowest job. In our case, although the average calculation time was around 415 seconds, the fastest calculation finished in roughly 250 seconds, making the synchronization overhead unreasonable.

5 Conclusion

Iterative workflows — for example, for parameter search in inverse problems — are important in computational science, and we urge the community to not neglect them. The system presented in this paper takes the first step towards their support. In situ processing confers multiple advantages in this context. Besides the illustrated savings in I/O time, Henson lets us monitor the execution of the analysis codes by directly accessing their memory and thus avoiding unnecessary overheads. Such capability can be useful for terminating the code early. For example, if we are interested in an L^∞-norm of a time-varying measurement, we can stop computation once the maximum difference exceeds the current best guess.

Our system also supports more complicated experiments than presented in the previous section. For example, the full analysis we would like to run involves taking snapshots of a live Nyx simulation and fitting parameters to them. To do so, it's essential for the jobs managed by the scheduler to interact with execution groups outside of it. The necessary ingredients are already built into the system (`henson_get_intercomm` can access inter-communicators across execution group levels), and we plan to experiment with such more complicated execution regimes in the near future.

Acknowledgements

We are grateful to Jack Deslippe for providing us the raw data on Edison queue times. This work was supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-05CH11231, and by the use of resources of the National Energy Research Scientific Computing Center (NERSC).

References

1. Dmitriy Morozov and Zarija Lukić. Master of Puppets: Cooperative Multitasking for In Situ Processing. In *Proceedings of High-Performance Parallel and Distributed Computing*, pages 285–288, 2016.
2. Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.
3. Qian Sun, Tong Jin, Melissa Romanus, Hoang Bui, Fan Zhang, Hongfeng Yu, Hemanth Kolla, Scott Klasky, Jacqueline Chen, and Manish Parashar. Adaptive Data Placement for Staging-based Coupled Scientific Workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15, pages 65:1–65:12, New York, NY, USA, 2015. ACM.

4. Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E. Papka. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, pages 19:1–19:11, New York, NY, USA, 2011. ACM.
5. M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro. Damaris/Viz: A nonintrusive, adaptable and user-friendly in situ visualization framework. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pages 67–75, Oct 2013.
6. Andrew C. Bauer, Berk Geveci, and Will Schroeder. *The ParaView Catalyst User's Guide v2.0*. Kitware, Inc., 2015.
7. B. Whitlock, J.M. Favre, and J.S. Meredith. Parallel insitu coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, pages 101–109, 2011.
8. Matthieu Dorier, Matthieu Dreher, Tom Peterka, Gabriel Antoniu, Bruno Raffin, and Justin M. Wozniak. Lessons Learned from Building In Situ Coupling Frameworks. In *First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, Austin, United States, November 2015.
9. U. Ayachit et al. Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.
10. M. Viel, G. D. Becker, J. S. Bolton, and M. G. Haehnelt. Warm dark matter as a solution to the small scale crisis: New constraints from high redshift Lyman-α forest data. *Physical Review D*, 88(4):043502, August 2013.
11. Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian T. Foster. Swift/T: Large-scale Application Composition via Distributed-memory Dataflow Processing. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 95–102, 2013.
12. A.J. Booker, J.E. Dennis Jr, P.D. Frank, D.B. Serafini, V. Torczon, and M.W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. *Structural Multidisciplinary Optimization*, 17:1–13, 1999.
13. H.-M. Gutmann. A radial basis function method for global optimization. *Journal of Global Optimization*, 19:201–227, 2001.
14. R.G. Regis and C.A. Shoemaker. A stochastic radial basis function method for the global optimization of expensive functions. *INFORMS Journal on Computing*, 19:497–509, 2007.
15. J. Müller and C.A. Shoemaker. Influence of ensemble surrogate models and sampling strategy on the solution quality of algorithms for computationally expensive blackbox global optimization problems. *Journal of Global Optimization*, 60:123–144, 2014.
16. G.G Wang and S. Shan. Review of metamodeling techniques in support of engineering design optimization. *Journal of Mechanical Design*, 129:370–380, 2007.
17. James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective Communicator Creation in MPI. In *EuroMPI*, pages 282–291, 2016.
18. Z. Lukić, C. W. Stark, P. Nugent, M. White, A. A. Meiksin, and A. Almgren. The Lyman α forest in optically thin hydrodynamical simulations. *Monthly Notices of Royal Astronomical Society*, 446:3697–3724, February 2015.
19. A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel. Nyx: A Massively Parallel AMR Code for Computational Cosmology. *The Astrophysical Journal*, 765:39, March 2013.