Dataflow Coordination of Data-Parallel Tasks via MPI 3.0

Justin M. Wozniak,* Tom Peterka,* Timothy G. Armstrong,†
James Dinan,* Ewing Lusk,* Michael Wilde,* Ian Foster*

*Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL USA
wozniak,tpeterka,dinan,lusk,wilde,foster@mcs.anl.gov

†Dept. Computer Science
University of Chicago
Chicago, IL USA
tga@uchicago.edu

ABSTRACT

Scientific applications are often complex collections of many large-scale tasks. Mature tools exist for describing task-parallel workflows consisting of serial tasks, and a variety of tools exist for programming a single data-parallel operation. However, few tools cover the intersection of these two models. In this work, we extend the load balancing library ADLB to support parallel tasks. We demonstrate how applications can easily be composed of parallel tasks using Swift dataflow scripts, which are compiled to ADLB programs with performance comparable to hand-coded equivalents. By combining this framework with data-parallel analysis libraries, we are able to dynamically execute many instances of a parallel data analysis application in support of a parameter exploration workload.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent Programming Structures*

General Terms

Design, Performance

Keywords

MPI, ADLB, Swift, parallel tasks, dataflow

1. INTRODUCTION

Many application workloads consist of multiple types of computation, as well as a significant mixture of procedures, such as I/O and validation. In the case of scientific simulation, computation is often followed by analysis and visualization, which may indicate that subsequent simulation is necessary. Large collections of tasks may be constructed to perform parameter exploration studies, Monte Carlo runs, optimization, uncertainty quantification, and other ensemble computations. Such applications may be elegantly expressed as a *dataflow* execution, in which a task is eligible to run when its requisite input data is ready. This model allows for implicit concurrency because data dependencies are explicit.

MPI enables the concurrent execution of multiple cooperating multiprocessing codes, each of which can have a separate communication context shared with only the MPI processes executing that code. MPI represents these contexts with *communicators* that typically form a tree hierarchy, starting from an initial *world* communicator, that encompasses all processes. Given a communicator, new child communicators can be created and passed to libraries for their exclusive use, allowing an application to be constructed through composition of existing parallel libraries and codes.

Communicators also provide a grouping capability that enables applications to form MPI process teams to process parallel subcomputations. Traditional MPI communicator creation was collective on a full parent communicator, including those processes that would not be members of the new communicator. This additional synchronization limited the ability of applications to form new process teams dynamically. To address this gap, the MPI 3.0 standard [14] added a new group-collective communicator creation routine [5] that is collective over only those processes that will be members of the new communicator. Group-collective communicator creation relaxes the synchronization involved in team formation, enabling the development of composite applications that dynamically generate teams to tackle individual computational tasks.

This core technology enables many new application composition models, as the low-level capabilities exposed are extremely powerful. However, the highly dynamical model enabled by group-collective communicator creation makes communicator management and data management a more significant programming challenge. In particular, application programmers must consider: 1) allocating processes to new communicators, and 2) moving data between locations in the parent space to the child library space. Note that while previous MPI communicator management also required data management, the dynamic nature of the new call requires more flexible data access mechanisms. A higher-level framework for composition of MPI libraries into a logical application could make the benefits of the new communicator creation calls accessible to a larger range of applications.

In this work, we extend a master-worker system and a

EuroMPI '13, Sept. 15–18, 2013, Madrid, Spain.

http://dx.doi.org/10.1145/2488551.2488561

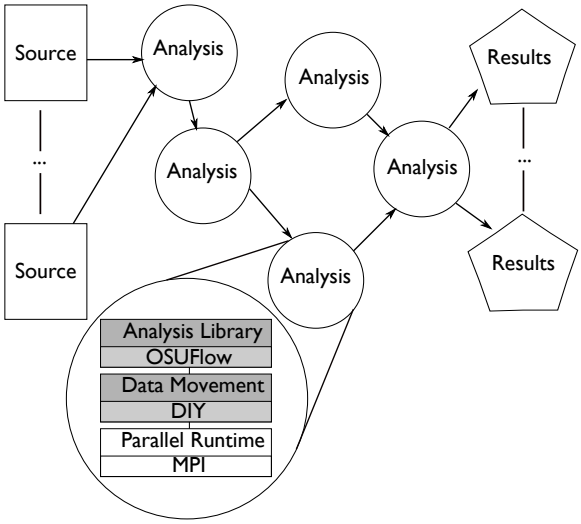

Figure 1: Dataflow program consisting of data-parallel tasks.

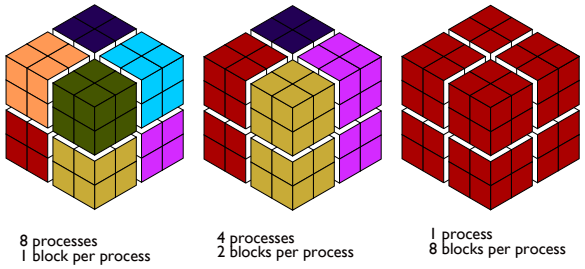

Figure 2: Allocation of eight blocks to processors. Blocks are colored by the process to which they belong.

high-level programming model to address these challenges in a reusable way. This model is an ideal fit for master-worker systems that allocate tasks (work units) to worker processes as workers become available. It also motivates the use of dataflow programming to coordinate control from one library call to the next. The Asynchronous Dynamic Load Balancer (ADLB) [8] is a master-worker system that provides multiple high level features; in this work, we extend it to use noncollective communicator creation for user-specified parallel tasks. Swift [17] is a scalable dataflow language and runtime engine; in this work, we extend it to use the ADLB parallel tasks feature. In this framework users may compose multiple parallel and/or sequential library calls in a high-level script that compiles into an ADLB/MPI program that manages a global data store and performs data-dependent processing.

The remainder of this paper is organized as follows. In Section 2, we provide background on the technologies used here and consider related work. In Section 3, we describe the implementation of the Swift parallel tasks feature and its novel use of MPI 3.0. In Section 4, we evaluate the implementation through synthetic benchmarks and an application in data-parallel particle tracing application. In Section 5, we offer summarizing remarks and briefly discuss future work.

2. BACKGROUND AND RELATED WORK

The effective combination of dataflow processing with data-parallel libraries integrates multiple concepts, including motivating requirements in *parallel data analysis and visualization* and *other application areas*, the ability of Swift to support *high-performance dataflow processing*, and the use of new MPI 3.0 *routines for dynamic communicator creation.*

2.1 Parallel Data Analysis and Visualization

As Figure 1 illustrates, the analysis and visualization of data computed from large-scale numerical simulations is a complex process. Multiple analysis steps are interconnected in a dataflow graph in which scheduling and execution are dynamic and asynchronous. Each analysis node in the graph itself is both complex and data-intensive and may require large-scale cluster or HPC resources and parallel libraries. The increasing complexity of analysis algorithms along with the need for scalability and performance has led to the composition of several libraries to accomplish a single task.

Figure 1 spotlights two such libraries, OSUFlow [13] and DIY [12, 11], that are used in our performance evaluation in Section 4. Together with MPI, this software stack provides a data-parallel infrastructure for the parallel advection of particles through a vector field for the purpose of visualizing streamlines or pathlines through the field. These libraries efficiently perform static SPMD data-parallelism at very large scales, but they do not include any mechanisms for the load balancing or dynamic scheduling needed to execute the overall flow graph. We use Swift for this purpose.

2.2 Other Relevant Parallel Applications

Many similar parameter exploration applications could also be approached with this model. Many parallel computations are configurable in one or more run time parameters, perhaps the most common being the number of MPI processes. In spatially decomposed data-parallel computations, the number of subdomains (blocks) into which the domain is decomposed is another common parameter. Some applications do not require the number of blocks to equal the number of processing elements, with several blocks residing on each processing element. (See Figure 2.)

Running a parameter exploration or search in this space allows investigators to find the best MPI and decomposition parameters to use later in bigger ensemble runs. Other applications that require good configuration parameters that may be approached by our model include: 1) selecting the optimal radix in the Radix-k image composting algorithm [10], 2) finding the best MPI-IO tuning hints [7], or 3) testing code transformations for linear algebra kernels [3].

2.3 High-performance Dataflow Processing

Swift is an implicitly parallel language for functional dataflow processing. Swift has C-like syntax and many features that have made it successful for many scientific application use cases [15]. Swift was originally implemented as a workflow language for distributed computing on grids [18] and clouds [6]. It has recently been reimplemented from scratch [17] to generate an MPI program that uses a scalable dataflow library, Turbine [16], to manage data dependencies and the ADLB load balancer [8] to distribute tasks.

In Swift, all variables are single-assignment futures and

may be of many primitive types, such as `int`, `string`, or `blob` (arbitrary bytes). These are stored in a globally-accessible data store. A statement `z=f(x,y);` triggers the execution of `f` when variables `x` and `y` are set, then sets `z`. Swift can call into functions implemented in external code libraries or as command-line applications. For example, `f` could refer to a function in an external library. Swift programs are composed of these *leaf* functions as well as traditional programming constructs such as functions, `if` statements, and `for` loops. The novel feature presented in this paper is support for parallel external functions that are executed cooperatively by many processes as a *parallel task*, implemented by using MPI 3.0.

2.4 Dynamic Communicator Creation in MPI

The MPI 3.0 standard introduces the `MPI_Comm_create_group` (CCG) routine for group-collective communicator. Dinan et al. demonstrated that this functionality can be implemented on top of existing MPI 2.0 functionality through recursive intercommunicator creation and merging [5]. While this approach can provide the same functionality, it requires $\log(P)$ communicator creation and merging steps. Direct implementation of the functionality provides greater efficiency than do existing MPI-2 communicator creation operations [4]. The impact of dynamic team formation on top of CCG was demonstrated in the context of load balancing a Markov chain Monte Carlo simulation [1].

In addition to relaxing synchronization, CCG defines a new semantic with respect to threads and collective routines. Previously, on a given communicator, collective operations could be called simultaneously only by a single thread per process. CCG relaxes this restriction by requiring an additional tag argument to distinguish the calls made by different threads. The ability to perform multiple CCG calls concurrently from different threads is critical to enabling flexible, dynamic team formation for MPI applications that utilize threads (e.g. via OpenMP) within a node.

3. IMPLEMENTATION

The CCG feature is an ideal fit for master-worker systems that allocate tasks (work units) to worker processes as workers become available. In many applications, it is not known how long a given task will take or how many processes will be requested for each task, so it is not possible to schedule work units in advance. With noncollective communicator creation, a master may allocate a subset of available workers and instruct them to assemble into a communicator to operate on a user task. We implemented features in Swift to support such use.

At runtime, a Swift program operates as a normal MPI program and does not use threads. As shown in Figure 3, the Swift program is translated [2] into a portable representation called "Turbine code," which includes references to requisite supporting tools and user libraries. This code is executed everywhere in the normal MPI SPMD manner; however, since it is an ADLB program, the first step is to segregate the ADLB services from workers. A number of ADLB workers are allocated as "engines", which process Swift dataflow logic. The ability of Swift to operate over multiple engines is a critical feature for scalability in terms of logic processing, task transmission to multiple ADLB servers, and memory space. The ADLB servers and Swift engines typically make

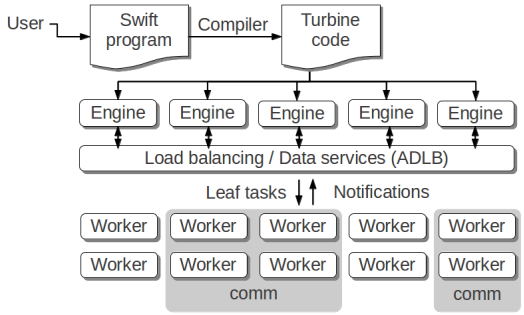

Figure 3: Runtime configuration of Swift program with parallel tasks.

up about 1% of the system. Finally, Swift "workers" simply execute leaf tasks as they are produced by Swift dataflow processing - this is where bulk user processing is performed. As diagrammed, workers may be dynamically grouped into communicators to execute parallel tasks.

ADLB is a master-worker system that provides multiple high level features, including task priorities, the ability to specify task location, task types, etc. We extended the ADLB task *Put* call to support a *parallelism* field, indicating the number of processes to allocate for the task. ADLB allocates the required number of workers and assembles them into a communicator before launching the task. Workers perform the task *Get* call, which has been extended by this work to additionally return an output communicator. For traditional single-process ADLB tasks, this is simply `MPI_COMM_SELF`; for parallel tasks, it is the communicator created by `MPI_Comm_create_group`.

We extended the Swift leaf function definition statement with the `@par` annotation, which declares that the function may be called as a parallel task. The caller applies the annotation with the desired number of processes for the new communicator, for example, `z = @par=8 f(x,y);`. If the definition of `f` did not have the `@par` annotation, a compile-time error occurs.

4. PERFORMANCE

To demonstrate the utility of the new features, we carried out synthetic benchmarks and an application case study. All measurements were performed on the Argonne Leadership Computing Facility *Eureka* visualization system, which contains 100 nodes, each containing two 64-bit 4-core Intel Xeon E5405 processors at 2 GHz, with 32 GB RAM, running on a GPFS filesystem shared with a large Blue Gene/P system. *Eureka* is designed to perform graphics processing on results from the Blue Gene/P. *Eureka* was chosen because we were readily able to use an MPI 3.0 implementation from source. We ran with MPICH 3.0.3; performance analysis was performed using MPE 1.3.0, Jumpshot, and various MPE log processing tools.

4.1 Synthetic Measurements

Our synthetic measurements directly measure the overhead by executing short-lived 0-second and 1-second tasks that strain our system. Such short tasks far exceed the performance requirements of our real application case, in which tasks execute for hundreds of seconds. We compare three im-

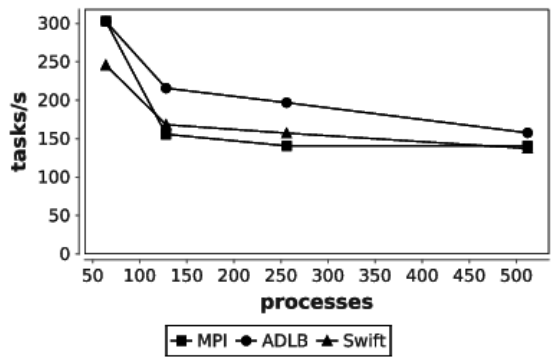

Figure 4: Parallel tasks task rate result for 0-second, fixed-parallelism tasks.

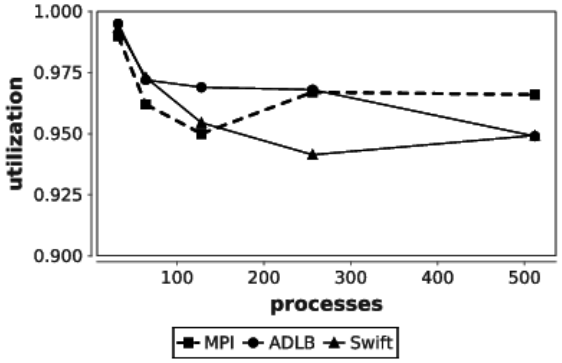

Figure 5: Parallel tasks utilization result for 1-second, fixed-parallelism tasks.

plementations: a hand-coded MPI benchmark written in C, a hand-coded ADLB benchmark, and a Swift benchmark. The MPI implementation is a simple for loop around the communicator create/free calls; the ADLB implementation is a simple for loop of tasks puts, with all workers executing parallel tasks; and the Swift implementation is a simple parallel foreach loop around calls to the task.

We use two metrics to evaluate the implementations: task rate and utilization. Task rate is number of parallel tasks that may be executed per unit time. This incurs the cost of repeatedly constructing the task in the given programming model and producing the new communicator with MPI. Utilization is the amount of CPU time spent in the 1-second task over total CPU time; it is not applicable to the 0-second tasks. Each measurement is the average of 10 runs.

Figure 4 shows the task rate result for the MPI, ADLB, and Swift benchmark implementations of 0-second tasks. For each process count (in ADLB and Swift, we measure only the worker processes), `MPI_COMM_WORLD` is split into 8 equal sized communicators that are created and immediately discarded. Each implementation produces hundreds of multi-process tasks/s. The ADLB implementation seems to be slightly faster, likely because the MPI implementation launches each round of tasks in lockstep, while ADLB distributes tasks to processors as they become available.

Figure 5 shows the utilization result for the MPI, ADLB, and Swift benchmark implementations. Again,

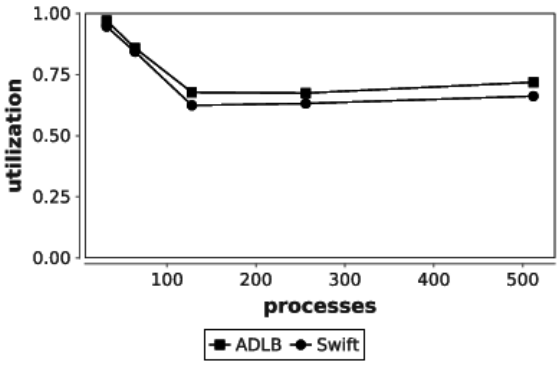

Figure 6: Parallel tasks utilization result for 1-second, varying-parallelism tasks.

`MPI_COMM_WORLD` is split into 8 equal sized communicators, but in this case the communicator is used to perform a simulated 1-second computation. The total time spent in computation is divided by the total process time to obtain a utilization result. Utilization is high for each system. This shows that the Swift model is usable for application tasks in the seconds timescale, as it does not degrade performance.

Figure 6 shows the utilization result for the ADLB, and Swift benchmark implementations. No hand-coded MPI implementation was produced (expressing "run whatever fits" would essentially be a rewrite of the ADLB parallel tasks feature). This time, `MPI_COMM_WORLD` is split into *varying*-sized communicators; for N processes total, task communicator sizes were 1, 2, 4, ..., N. Again, the communicator is used to perform a simulated 1-second computation. The total time spent in computation is divided by the total process time to obtain a utilization result. Again, utilization is acceptable for tasks in the seconds timescale and above, although there is room for improvement.

4.2 Particle Tracing Parameter Exploration

This section demonstrates the utility of our model by presenting a complete application: an ensemble of parallel particle tracing computations making up a parameter exploration. The OSUFlow code visualizes streamlines of advecting particles in a 3D vector field. For example, Figure 7 is such a visualization of the coolant flow in the plenum of a nuclear reactor [9]. Our objective is to search the parameter space and arrive as quickly as possible at the configuration that produces the shortest run time for the parallel particle advection problem.

OSUFlow is a data-parallel program with some number of MPI processes (np) and each process having some number of blocks (bp). For a given problem and machine, the shortest run time will be found somewhere within ($np \times bp$), usually not the minimum or maximum of either parameter (those locations produce over- or under-decomposed configurations with too much or too little computation per block with respect to the communication overhead between blocks). Even if an analytical model exists for predicting the optimum configuration, an exploration of the parameter space is usually required in order to create or tune such a model. Such parameter exploration problems can be performed with the hybrid data-parallel dataflow model we have developed.

Figure 8 illustrates how data-parallel libraries can be com-

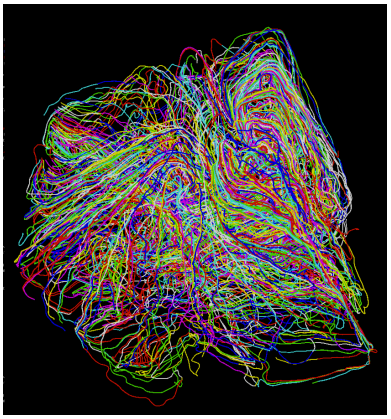

Figure 7: Streamlines representing particles in simulated fluid flow.

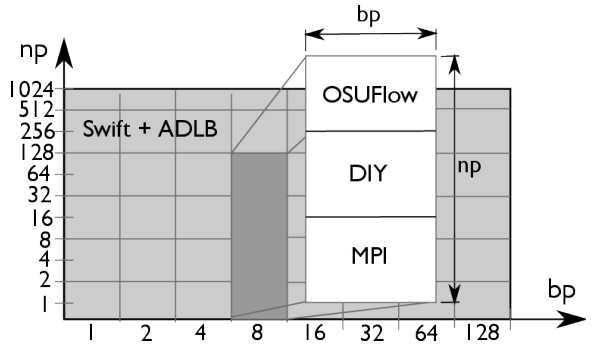

Figure 8: Exploration of the two-dimensional search space of $(np \times bp)$ (number of processes $\times$ blocks per process). Each task in that space consists of a data-parallel program: the OSUFlow parallel particle tracing and flow visualization library, which is built on top of the DIY parallel analysis library, which in turn is built on top of MPI. Our system makes such composition of libraries possible and executes many such tasks in parallel. One task, 128 processes and 8 blocks per process, is highlighted.

posed into a parameter optimization problem for parallel particle advection. Swift with ADLB allows the parallel exploration of many configurations in the parameter space in parallel. Each task, such as the highlighted one, is a data-parallel program that can be composed of numerous other libraries. The parameter space can be explored in many different ways, depending on the search algorithm encoded in the Swift user program.

4.2.1 *Performance Results*

The implementation of the Swift script to perform the parameter sweep is shown in Figure 9. The parallel `foreach` loops construct the configuration parameters. The syntax `[i:j]` constructs the array of integers between `i` and `j` for iteration; these are used as exponents (`2**x`) to produce the actual inputs.

Figure 10 shows the utilization of the OSUFlow script running on 518 processors. Black regions indicate OSUFlow

```
1  // Define call to OSUFlow feature MpiDraw
2  @par (float t) mpidraw(int bf) "mpidraw";
3
4  main {
5    foreach b in [0:7] {
6      // Block factor: 1-128
7      int bf = round(2**b);
8      foreach n in [4:9] {
9        // Number of processes/task: 16-512
10       int np = round(2**n);
11       float t = @par=np mpidraw(bf);
12       printf("RESULT: bf=%i np=%i -> time=%0.3f",
13                       bf,   np,      t);
14     }}}
```

Figure 9: Swift code for parameter sweep over OSUFlow configurations.

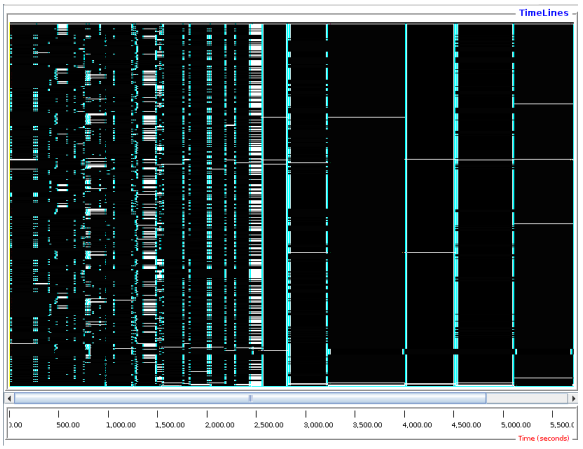

Figure 10: Jumpshot view of OSUFlow utilization.

native code executing, blue indicates Swift/ADLB control communication, and white regions are gaps.

The desired result of the script is to explore the parameter space in order to pick good parameters for parallel particle tracing with OSUFlow. Thus, the performance results are depicted by a contour plot in Figure 11. Blue regions are the shortest runtimes and best performance. This confirms the finding in [13] that, in general, higher values of *bp* lead to shorter run times, but in this work, the new dataflow programming model enabled the testing of many more configurations much faster than in that earlier work.

5. SUMMARY

In this work, we presented a motivating case for the use of dataflow programming to control executions of parallel tasks constructed with new MPI 3.0 techniques. We presented background on the technologies used and how they were integrated into the high-level Swift model. We presented synthetic performance results as an initial investigation into the performance impact of the Swift model. Additionally, we demonstrated a complete application consisting of many data-parallel visualization processing tasks of varying size.

In future work, we will improve our use of MPI 3.0 features at the micro-scale so that Swift may be used on applications with even finer-grained tasks. We will also investigate opportunities to apply scheduling and prioritization in order

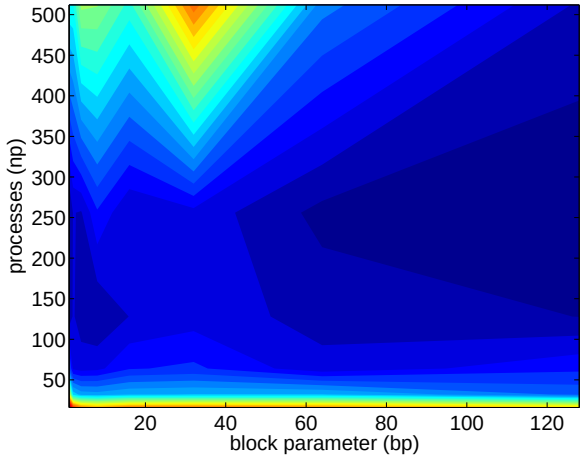

Figure 11: Performance for OSUFlow under varying configurations. The color indicates the run time: blue represents the faster cases.

to improve utilization at the macro-scale. Ultimately, we intend to use Swift as a way to construct composite applications consisting of large numbers of tasks of varying size, to support scientific computing on the largest computing systems.

Acknowledgments

This work was supported by the U.S. Department of Energy under the ASCR X-Stack program (contract DE-SC0005380) and contract DE-AC02-06CH11357. Computing resources were provided by the Mathematics and Computer Science Division and Argonne Leadership Computing Facility at Argonne National Laboratory.

6. REFERENCES

[1] M. H. Arafat, J. Dinan, S. Krishnamoorthy, T. Windus, and P. Sadayappan. Load balancing of dynamical nucleation theory Monte Carlo simulations through resource sharing barriers. In *Proc. 26th Intl. Parallel and Distributed Processing Symp.*, IPDPS '12, May 2012.

[2] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler optimization for distributed dynamic data flow programs, 2013. MCS Tech. Report: ANL/MCS-P4045-0313 *Under review for:* Int'l Conference on Parallel Architectures and Compilation Techniques.

[3] P. Balaprakash, S. M. Wild, and B. Norris. SPAPT: Search problems in automatic performance tuning. *Procedia Computer Science*, 9(0):1959 – 1968, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[4] J. Dinan, D. Goodell, W. Gropp, R. Thakur, and P. Balaji. Efficient multithreaded context ID allocation in MPI. In *Proc. Recent Adv. in MPI - 19th European MPI Users' Group Meeting*, EuroMPI '12, September 2012.

[5] J. Dinan, S. Krishnamoorthy, P. Balaji, J. R. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu. Noncollective communicator creation in MPI. In *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting*, EuroMPI '11, 2011.

[6] M. Hategan, J. Wozniak, and K. Maheshwari. Coasters: uniform resource provisioning and access for scientific computing on clouds and grids. In *Proc. Utility and Cloud Computing*, 2011.

[7] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[8] E. L. Lusk, S. C. Pieper, and R. M. Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, 17:30–37, January 2010.

[9] E. Merzari, W. Pointer, A. Obabko, and P. Fischer. On the numerical simulation of thermal striping in the upper plenum of a fast reactor. In *Proceedings of ICAPP 2010*, San Diego, CA, 2010.

[10] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. A Configurable Algorithm for Parallel Image-Compositing Applications. In *Proceedings of SC 09*, Portland OR, 2009.

[11] T. Peterka and R. Ross. Versatile communication algorithms for data analysis. In *EuroMPI Special Session on Improving MPI User and Developer Interaction IMUDI'12*, Vienna, Austria, 2012.

[12] T. Peterka, R. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri. Scalable Parallel Building Blocks for Custom Data Analysis. In *Proceedings of the 2011 IEEE Large Data Analysis and Visualization Symposium LDAV'11*, Providence, RI, 2011.

[13] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. In *Proceedings of IPDPS 11*, Anchorage AK, 2011.

[14] The MPI Forum. MPI: A message-passing interface standard, 2012.

[15] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37:633–652, 2011.

[16] J. M. Wozniak, T. G. Armstrong, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster. Turbine: A distributed memory data flow engine for many-task applications. In *Int'l Workshop Scalable Workflow Enactment Engines and Technologies (SWEET) 2012*, 2012.

[17] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Scalable data flow programming for many-task applications. In *Proc. CCGrid*, 2013.

[18] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Proc. Workshop on Scientific Workflows*, 2007.