Accelerated solution of a moral hazard problem with Swift

Tiberiu Stef-Praun[1], Gabriel A. Madeira[2], Ian Foster[1], Robert Townsend[2]

[1] Computation Institute, University of Chicago, USA

[2] Economics Department, University of Chicago, USA

tiberius@ci.uchicago.edu

Abstract. The study of human interactions and the modeling of social structures often require large-scale computational experiments involving complex models and substantial computational resources. The formulation of such models and computations can be taxing for researchers. We introduce the *Swift* parallel programming system and the *SwiftScript* scripting language, which together enable the rapid specification and execution of such experiments. The researcher specifies computation in SwiftScript, via a set of procedures with input-output (file) dependencies; the Swift system can then execute various sub-computations on parallel and/or distributed computing systems, ensuring reliable execution and respecting data dependencies. We present Swift features by describing their use in a computational economics application, a moral hazard problem. The benefits obtained suggest that Swift may also have utility in other social sciences.

Introduction

The study of human interactions, as addressed in various fields of the social sciences, frequently involves large sets of unstructured input data and complex models of both individual actors and inter-actor interactions. The development of such models and their efficient execution on high performance computing platforms can be challenging tasks. A frequent consequence is the use of restricted-scale models and a limited validation of results.

Swift represents a potential solution to these problems. Its high-level parallel scripting language, SwiftScript, provides for the concise specification of applications involving many loosely coupled computational tasks. The Swift runtime system provides for the efficient execution of SwiftScript programs on both parallel and distributed computing platforms.

Swift (Zhao et al., 2007) is a successor to the GriPhyN virtual data system (VDS) (Zhao et al 2000), which was originally designed to automate the processing of large data sets from high energy physics experiments. Swift, like the Pegasus planner (Deelman et al 2005) used in VDS, can map program components onto individual workstations, clusters, and/or collections of Grid resources. Taverna (Oinn et al 2005) and Kepler (Ludascher et al 2005) also have similarities, but emphasize the invocation of remote services. GenePattern (Reich et al 2006) implements a graphical approach to expressing computations, and MapReduce (Dean and Ghemawat 2004) is optimized for high volume parallel data processing on Google's dedicated infrastructure. Swift is distinguished by its explicit treatment of file system data and its support for strong typing and both task-parallel and data-parallel execution.

We describe here the use of Swift to implement a model of dynamic *optimal choice of organization* under moral hazard assumptions. In this application, worker agents (entities) can be organized in groups, with good internal information, or in a relative performance individualistic regime, which is informationally more opaque. Faced with various choices of production and organization, and with various payoffs from the entrepreneur agent, the worker agents use linear programming over a discrete grid of parameters to compute their optimal behavior. The fragmented parameter space allows us to exploit parallelism when solving the problem.

The rest of the paper is organized as follows. We first introduce SwiftScript and Swift; then describe the economic problem that we use to illustrate Swift features; and finally describe the SwiftScript implementation details of the economic problem and its execution.

Swift and SwiftScript

We introduce the language constructs used for expressing application programs and present the execution infrastructure that enables distributed execution of program components.

The SwiftScript language

SwiftScript is a simple high-level scripting language with a functional, data-driven execution model. It is particularly well suited for specifying computations involving loosely coupled collections of executable programs that communicate by reading and writing files. Dependencies between program components are expressed via dataflow constraints: when one component outputs a component that is input to one or more other components, the first component must complete execution before the other can commence. The Swift engine uses various strategies to achieve high performance when executing SwiftScript programs on parallel and distributed computing systems, while taking into account all such dependencies.

A distinctive feature of SwiftScript is its treatment of file system data as first class objects. The XDTM (XML Data Type and Mapping) specification of Moreau et al (2005) is used to describe mappings between typed SwiftScript variables and file system variables. For example, in a biomedical application, the contents of directories containing image files may be mapped to SwiftScript arrays of objects with type "Image." The SwiftScript "foreach" operator can then be used to specify that a particular operation be applied to each element of such arrays. Thus, it becomes easy to specify, compose, and type check, applications that operate on data maintained in complex file system structures.

Calls to executable programs are specified via *atomic procedures*. An atomic procedure encapsulates a software component that has been installed on one or compute systems, as documented in a *site catalog*. The atomic procedure defines the data passed to and from the executable program, the file(s) and other data that must be passed to and from a site to enable execution, and the format of the call to the executable program. At run-time, Swift chooses from the catalog a resource that has the specified application installed, and executes it.

A SwiftScript *compound procedure* can be used to specify program logic involving multiple calls to either atomic procedures or other compound procedures. For example, we may use a foreach statement to apply a particular atomic procedure to each element of an array.

A SwiftScript program typically comprises a set of type definitions, one or more atomic and compound procedure definitions, and finally a set of data definition statements (defining connections between SwiftScript variables and file system structures) and calls to atomic and

compound procedures that operate on those variables. The latter components can be viewed as equivalent to the main() procedure of a C program.

Distributed computing with Swift

SwiftScript mapper constructs connect a logical (programmatic) data representation to a physical (file) entity containing data. This information allows Swift to virtualize data resources, in the sense that both their physical location and physical format is abstracted away from the SwiftScript programmer.

SwiftScript program execution is data driven. The Swift engine identifies as executable any procedure for which all inputs are available. As execution proceeds and outputs are generated, components depending on these generated objects also become executable.

The dispatch of individual components to computers for execution is handled by *provider* extensions, which address both data (file) transfer to and from remote execution sites, and also remote execution invocation and status management. Using the Karajan (von Laszewski et al 2007) just-in-time execution engine, Swift masks much of the complexity of parallel and distributed computing from the user, through its reliance on Globus (Foster 2006) Grid middleware and CogKit (von Laszewski et al 2001) client libraries. At run time, the mapping of executable application components onto local and/or remote computers is performed by an internal resource scheduler. This scheduler uses a *site catalog* listing sites at which application components are installed. The installation of an application at a site is a one-time process; after that any application invoking that specific application can be mapped to that site.

Economic application

Following Madeira and Townsend (2007), we introduce a model of a social interaction based on economic principles: consider one entity being in control of some resources (the entrepreneur), and entering in a business contract with some other entities that will use these resources to produce outputs (the workers). There are two organizational forms available, one in which the workers cooperate on their efforts and divide up their income (thus sharing risks), and another in which the workers are independent of each other, and are rewarded based on their relative performance. Both are stylized versions of what is observed in tenancy data in villages such as in Maharastra, India (Townsend and Mueller 1998, Mueller et al 2002).

The organizational regime in these communities can change over time. Cooperative regimes sometimes break down into individualistic arrangements, and cooperation and risk sharing may emerge from initially competitive communities. This organizational instability is a key element that Madeira and Townsend incorporate in their model.

The model of Madeira and Townsend formulates the tradeoff between individualistic versus cooperative regimes as a choice of alternative incentive structures under moral hazard: production depends on unobserved effort, and incentives for effort may be provided under both individualistic and cooperative arrangements. They show numerically that these two types of arrangements may be coexisting and interchangeable.

Moral hazard problem

We consider two agents and a principal. The agents' preferences are described by discounted expected utilities over consumption *c* and effort *e*. The utility of agent *i* at period *t* is

$$w_i^t = E\{\sum_{s=1}^{\infty} \beta^{s-t}[U(c_i^s) + V(e_i^s)]\}$$

The parameter β represents a subjective discount factor, *U* is the utility from consumption, and *V* the utility of effort, which is decreasing. There is a production function, which maps the agents' effort into an output, and the model represents this as a probability distribution of outputs given the efforts of both agents:

$$p(q_1^t, q_2^t \mid e_1^t, e_2^t) \geq 0$$

The entrepreneur's share (or profit) is given by the surplus of production over consumption:

$$S^t \equiv \sum_{s=t}^{\infty} \left(\frac{1}{1+r}\right)^s [q_i^s + q_2^s - c_1^s - c_2^s],$$

where the parameter *r* is an exogenous interest rate.

Model variables are discretized, allowing the use of linear programming in the solutions. Thus, we can use lotteries as optimal policies and obtain a reliable solution (conditional on the grid). The problem's dimensionality depends on the size of these grids of consumption *C*, efforts *E*, and outputs *Q* of the agents, and also on the set of possible organizational forms, *O* (cooperative groups, with corresponding power balances within it, or relative performance). This is solved for a grid of current utility pairs for the agents, *W*. The elements of *W* are initial conditions for the solution but they also define the set of possible states in any future period. In practice, the future value of elements of *W* are part of current policies: promises for the future are part of the incentives given today to motivate effort, and the resulting dynamics on the elements of this set drive the organizational history generated by the model.

We solved the model with the following cardinality (parameter granularity) measures: $|Q| = 2$, $|E| = 2$, $|C| = 18$, $|W| = 30$, $|O| = 102$ (there are 101 possible values of Pareto weights defining the internal balance of power inside groups, and also the possibility of relative performance).

Moral hazard implementation

Because each agent and variable potentially introduces a dimension in the problem domain, computational requirements grow exponentially with the number of agents and the size of the grids. To avoid the resulting Curse of Dimensionality, the problem is broken into five interdependent pieces. Also, a new variable is introduced: interim utility, with summarizes the utility from both current consumption and future arrangements and belongs to a grid set *V*, which has cardinality $|V| = 45$. Each stage is solved by linear programming.

We solve the problem backward. First (last stage chronologically), the balance between promises for the future and consumption to optimally reward agents (to give them interim utilities) is defined. This linear program takes as inputs the surplus of the principal as a function of the future utilities (a 30 x 30 matrix that describes the surplus for each pair in W x W, and an initial pair of interim utilities, and determines the optimal probability distribution of elements of *C* and future utilities in *W* for each agent. This is solved for a grid of |V| x |V| elements, generating as output a matrix of |V| x |V| elements representing the surplus of the principal for each pair of interim utilities. Each grid point of this program is generated by a program with 291,600 elements and a constraint matrix with dimension 3 x 291,600.

Next, we solve two programs for groups and one for relative performance. (The group and relative performance stages can run in parallel; the two group stages must run in sequence.)

The relative performance program takes as an input the matrix determining the surplus associated with each interim utility pair (the output of the first program), and an initial utility pair conditional on the regime (that for each agent lays in a grid W_r, for which we impose a cardinality of 40). The linear program determines the optimal joint probability distribution of outputs, efforts, and interim utilities, which generate the regime-conditional utilities subject to a set of technological and incentive constraints (implied by a constraint matrix in the LP program). This program must be solved for a grid of initial values of W_r for each individual, and thus generate as an output a matrix of $| W_r |\text{x}| W_r |$ elements representing the surplus under relative performance conditional on initial promises (for the RP regime). Each grid point of this program is generated by a program with 10,816 elements and a constraint matrix with dimension 21 x 10,816.

The first group specific program (last chronologically) takes as an input the matrix determining the surplus associated with each interim utility pair (the output of the first program) and an initial Pareto weight and a surplus level.

The linear program determines the utility maximizing joint probability distribution of outputs, efforts, and interim utilities, subject to technological and incentive constraints. This program is solved for a grid of 101 values of the Pareto weights and 52 values of surplus. The outputs are two 101 x 52 matrixes determining the optimal utility of respectively agent one and two given a Pareto weight and a surplus level. Each grid point of this program is generated by a program with 10,816 elements and a constraint matrix with dimension 18 x 10,816.

The second group-specific program takes as an input the output matrixes of the first group program (the stage just described) and an initial group-specific utility pair (that for each agent lays in a grid W_g, for which we impose a cardinality of 40). It chooses surplus maximizing joint distribution of surplus and Pareto weights conditional on an initial group-specific surplus. The output is a matrix of size 40 x 40 determining the optimal surplus under groups for each initial utility pair in $| W_g | \text{x} | W_g |$. Each grid point of this program is generated by a program with 5252 elements and a constraint matrix with dimension 3 x 5252.

Finally, a fifth stage (the first chronologically) determines the choice of regime and utilities in each regime. It has as inputs the outputs of the Relative Performance program (the 40 x 40 matrix describing the surplus function under RP) and of the second group program (the 40 x 40 matrix describing the surplus function under groups), and also an initial pair of utilities. This program is run for a grid of 30 x 30 elements (corresponding to pairs of the elements in W) and generates as an output a 30 x 30 matrix determining the overall surplus function. This stage has a choice vector with 1600 elements, and a constraint matrix of dimension 3 x 1600.

The output of this last stage can be used, iteratively as an input in the first stage. This program is run until this last surplus function converges, with a solution that represents the infinite-period solution of the model.

Implementation and measurements

We introduce here the practical aspects of parallelizing applications by referring to the moral hazard model presented above. We describe the individual components of the parallel implementation, and then we show how to use Swift to connect these components.

Problem structure and implementation details

We first implemented the application in Matlab and used CPLEX to solve linear problem instances. To avoid licensing costs when running on many processors, we then moved to open source alternatives: Octave and CLP (from the COIN-OR optimization libraries), respectively.

The structure of the problem at each stage is similar: the Matlab/Octave code sets up the matrices representing the linear programming parameters, and the linear solver uses these matrices to generate an optimal solution. This procedure is replicated for all points in the parameter grids mentioned previously, and because the problems are independent for each set of parameters, the parallelization procedure is straightforward: the instances of linear problems from each grid point can be solved in parallel on different machines.

We next define the canonical component that is the unit for parallelization: it is the piece of the Matlab code that sets up the linear problem and the associated linear solver that produces the solution that maximizes the objective function. This functionality is encoded in our program in the *moralhazard_solver* atomic procedure. In the scenario of the parallel execution on multiple machines, the prerequisite is that we have both the supporting software (Matlab/Octave and CPLEX/CLP) installed on those machines and the problem instance copied (staged in) to that machine for execution.

Another atomic function is needed due to the fact that a previously monolithic application will be solved on pieces in a distributed environment. The *merging* function assembles the partial solutions from the distributed solvers into a form that the remaining code can import.

The rest of the parallelization process consists of expressing the general logic of the MoralHazard problem in Swift, and of determining which parts of the original Matlab code go into each distributed code component. Each Matlab code component takes, as input, parameters that specify the grid point to be solved by the current component instance. This part is where we implement the looping over the grid of parameters, and invoking the atomic procedure that represents the remote solvers.

The complete Moral Hazard program (Figure 1) consists of the five stages connected as described by the logic of the solution, with dependencies between the stages being implemented, as explained above, through files generated by one stage and consumed by the next. Each stage is represented by a compound fan-out procedure that loops over the parameters grid and solves the individual linear optimization problems, followed by an atomic fan-in procedure that merges results for that stage. We show in Figure 2 SwiftScript fragments that specify atomic procedures, the first stage compound procedure, and the invocations of stages one and two.

Program execution

The Swift execution engine, when passed a SwiftScript program, attempts to execute any procedure call that has all inputs defined. A runnable atomic procedure is executed by choosing a site on which the corresponding code will be executed, then staging in the required input data files, and after the execution ends staging out the results. Runnable compound procedures are executed directly. Completion of a procedure's execution and delivery of its output files can enable the execution engine to submit other tasks that were dependent on these as inputs. When all files have been generated, the computation is complete. If there is an error, the execution engine resubmits the erroneous task(s) automatically.

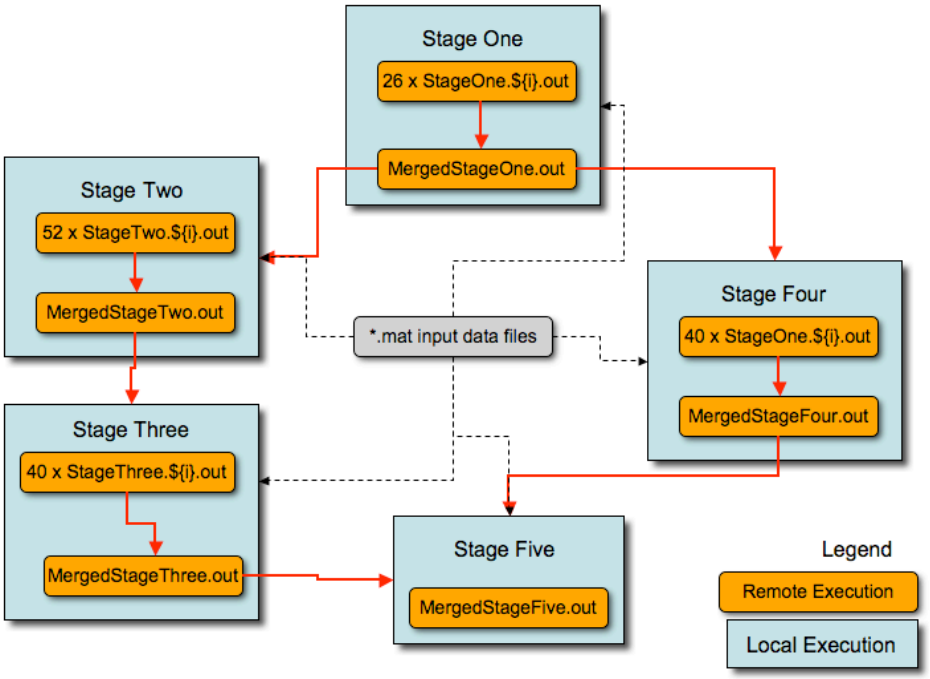

Figure 1. The five stages of the moral hazard problem

Figure 3 illustrates the execution of a small moral hazard problem on 40 processors of a cluster at the University of Chicago. In this figure, each horizontal line represents the time that elapses from when a task first becomes executable to when its execution completes; the color indicates whether the task is blocked waiting for a processor (red) or executing (green).

We see in the figure how the various phases execute with varying degrees of parallelism. During most phases, all tasks can execute immediately they become executable, due to the large number of processors allocated relative to the problem size. The one exception is the large red area, which corresponds to a time when more tasks are executable than can be dispatched immediately for execution.

Total execution time is 1,672 seconds, for a speedup of about 19 relative to the sum of the sequential execution times of the various components (32,130 seconds). Figure 3 suggests some ways in which we can improve performance: address load imbalances between different processors executing tasks from the same phase; accelerate transitions between phases; and increase parallelism in Phase 5.

Summary

We have used an example from computational economics to illustrate the capabilities of Swift, a parallel programming system designed to enable the high-performance execution of loosely coupled computations. The moral hazard problem that we consider involves a modest number of atomic procedure calls coordinated by the reading and writing of files. The SwiftScript program that expresses this coordination is concise and can be executed automatically on a range of parallel and distributed computing platforms. Performance results with a small problem are promising. Other Swift users have run far larger problems (10,000 or more tasks on many hundreds of processors) in such fields as bioengineering and physics.

```
// We define an atomic procedure: the linear solver. First we specify function arguments:
(file solutions) moralhazard_solver (file scriptfile, int batch_size, int batch_no, string inputName,
                                     string outputName, file inputData[], file prevResults[]) {
    app{
        // And then we construct the call to the linear solver program:
        moralhazard @filename(scriptfile) batch_size batch_no inputName outputName;
    }
}

// A second atomic procedure: merge
(file mergeSolutions[]) econMerge (file merging[]) {
    app{
        econMerge @filenames(mergeSolutions)  @filenames(merging);
    }
}

// We define the stage one procedure—a compound procedure
(file solutions[]) stageOne (file inputData[], file prevResults[]) {
    file script<"scripts/interim.m">;
    int batch_size = 26;
    int batch_range = [0:25];
    string inputName = "IRRELEVANT";
    string outputName = "stageOneSolverOutput";
    // The foreach statement specifies that the calls can be performed concurrently
    foreach i in batch_range {
        int position = i*batch_size;
        solutions[i] = moralhazard_solver(script,batch_size,position,
                                          inputName, outputName, inputData,  prevResults);
    }
}

// The main program. We first define mappings between three (arrays of) SwiftScript
// variables and the files on which our program will operate. Each declaration specifies
// a type, a variable name, and (as <...>) the mapper routine that links variable and files.
file stageOneSolutions[]<fixed_array_mapper; files = "stageOneSolverOutput.0,
   stageOneSolverOutput.26, ..., stageOneSolverOutput.650">; // A total of 26 output files
file stageOneInputFiles[]<fixed_array_mapper; files = "inputs/iteration.mat,
   inputs/param.mat, inputs/limits.mat, inputs/Surplus.mat">;
file stageOnePrevFiles[]<fixed_array_mapper; files = "inputs/dummy.txt">;

// We then invoke the first stage of the computation: the "stageOne" procedure
stageOneSolutions = stageOne (stageOneInputFiles,stageOnePrevFiles);

// And then merge the results from the "stageOne" compound procedure
file stageOneOutputs[]<fixed_array_mapper; files = "stage_1_opt.csv
   stage_1_index.csv stage_1_prob.csv">;
stageOneOutputs = econMerge(stageOneSolutions);
```

Figure 2. SwiftScript code fragments (see text for details)

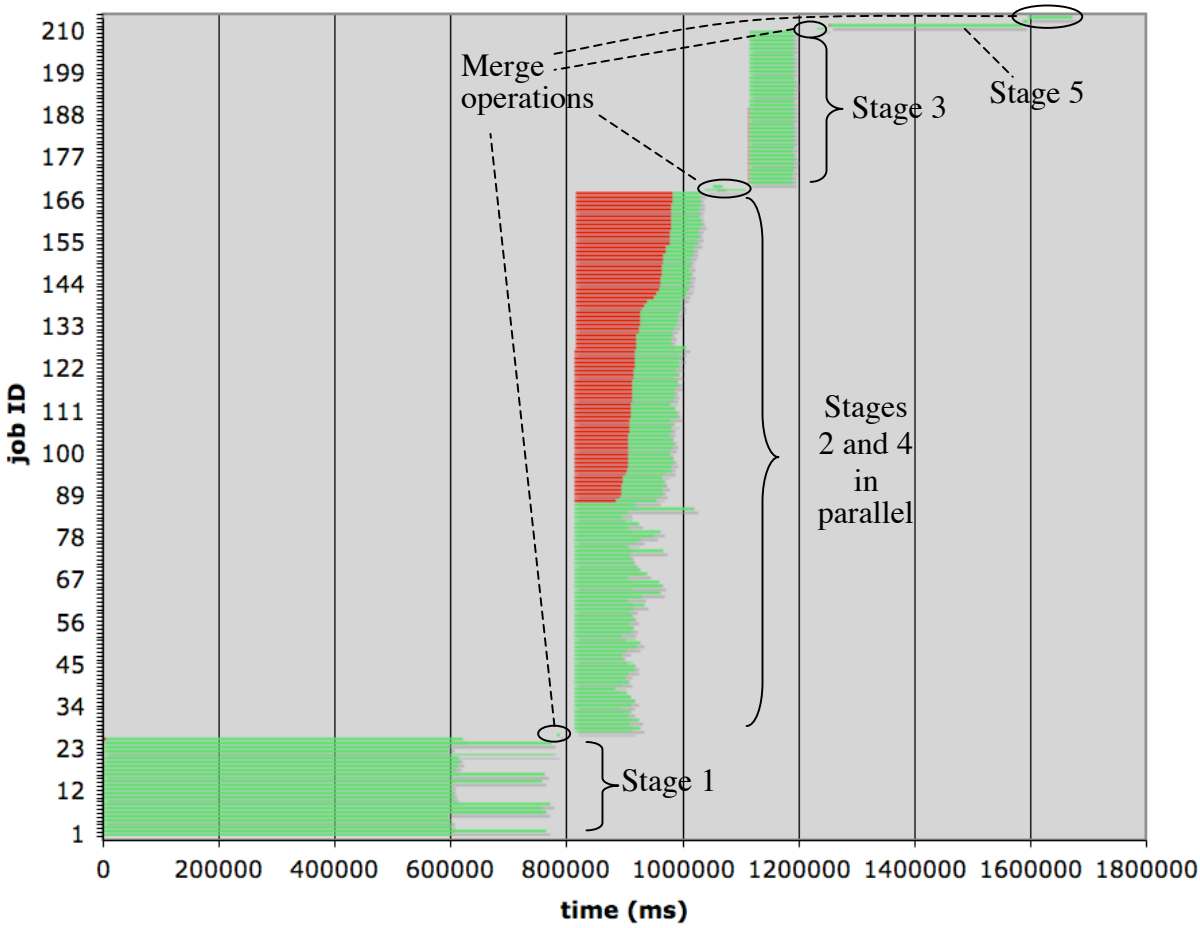

Figure 3. Execution of the moral hazard computation (see text for details)

We have shown how Swift allows the programmer to focus on describing the executable programs that must be invoked and the flow of data between file system and various invocations of those programs. The Swift engine then works to send (stage out) input files to the sites (clusters) where execution will occur, manage the execution of relevant programs, and finally retrieve (stage out) any outputs needed by the researcher. Thus, all the researcher needs to worry about is providing input files and declaring the sites that are capable of executing the program components. Our experience is that this separation of concerns between programmer and runtime system encourages more frequent and innovative uses of high-end computing, and also the reuse of program components within research teams.

In current work, the Swift team is working to improve the strategies that Swift uses to acquire resources (Raicu et al 2007) and to extend SwiftScript with features such as macros for basic data structure (e.g., string) manipulations. We are also working to integrate virtual data catalog support previously developed for the GriPhyN virtual data system (Zhao et al 2000), with the goal of facilitating documentation of data provenance. Our goal is to encourage innovative uses of large-scale computing within the sciences by facilitating the specification, execution, and sharing of programs involving many loosely coupled computations.

Acknowledgments

This work was supported by the Computation Institute of the University of Chicago and Argonne National Laboratory. We are grateful to members of the Swift team for their help.

References

Dean, J. and Ghemawat, S. (2004): 'MapReduce: Simplified data processing on large clusters', *6th ACM Conference on Operating System Design and Implementation*, San Francisco, CA.

Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G., Good, J., Laity, A., Jacob, J., and Katz, D. (2005): 'Pegasus: A framework for mapping complex scientific workflows onto distributed systems,' *Scientific Programming*, vol. 13, no. 3, pp. 219-237.

Foster, I. (2006): 'Globus Toolkit Version 4: Software for service-oriented systems', *Journal of Computational Science and Technology*, vol. 21, no. 4, pp. 523-530.

Ludaescher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., and Jones, M. (2005): 'Scientific workflow management and the Kepler system', *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039-1065.

Madeira, G. and Townsend, R. (2007): 'Endogenous groups and dynamic selection in mechanism design', forthcomming, *Journal of Economic Theory*.

Moreau, L., Zhao, Y., Foster, I., Voeckler, J., and Wilde, M. (2005): 'XDTM: XML data type and mapping for specifying datasets', in *European Grid Conference*, Amsterdam.

Mueller, R., Prescott, E., and Sumner, D. (2002): 'Hired hooves: Transactions in a south indian village factor market', *The Australian Journal of Agricultural and Resource Economics*, vol. 46, pp. 235-255.

Oinn, T., Greenwood, M., Addis, M., Alpdemir, M., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M., Senger, M., Stevens, R., Wipat, A., and Wroe, C. (2005): 'Taverna: lessons in creating a workflow environment for the life sciences', *Concurrency and Computation: Practice and Experience*, vol. 8, no. 10, pp. 1067-1100.

Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I., and Wilde, M. (2007): 'Falkon: a fast and lightweight task execution framework', in *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'07)*, ACM Press.

Reich, M., Liefeld, T., Gould, J., Lerner, J., Tamayo, P., and Mesirov, J. (2006): 'Genepattern 2.0', *Nature Genetics*, vol. 38, no. 5, pp. 500–501.

Townsend, R. and Mueller, R. (1998): 'Mechanism design and village economies: From credit, to tenancy, to cropping group', *Review of Economic Dynamics*, vol. 1, no. 1., pp. 119-172.

von Laszewski, G., Foster, I., Gawor, J., and Lane, P. (2001): 'A Java Commodity Grid Toolkit'. *Concurrency: Practice and Experience*, vol. 13, no. 8-9, pp. 643-662.

von Laszewski, G., Hategan, M., and Kodeboyina, D. (2007): 'Java CoG kit workflow', in Taylor, I., Deelman, E., Gannon, D., and Shields, M. (eds.): *Workflows for e-Science*, Springer, pp. 340-356.

Zhao, Y., Wilde, M., Foster, I., Voeckler, J., Dobson, J., Gilbertand, E., Jordan, T., and Quigg, E. (2000): 'Virtual data grid middleware services for data-intensive science', *Concurrency and Computation: Practice and Experience*, vol. 18, no. 6, pp. 595-608.

Zhao, Y., Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Raicu, I., Stef-Praun, T., and Wilde, M. (2007): 'Swift: Fast, reliable, loosely coupled parallel computation', In *IEEE International Workshop on Scientific Workflows*, Salt Lake City, Utah.